CONTENT-BASED DISSEMINATION OF XML DATA

Ni Yuan

NATIONAL UNIVERSITY OF SINGAPORE

2007

CONTENT-BASED DISSEMINATION OF XML DATA

NI YUAN

(B.Sc. Fudan University)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2007

Acknowledgement

I would like to take this section to express my sincere thanks to many people without whom this dissertation would not be possible.

My foremost thank goes to my supervisor, Professor Chan Chee-Yong, for his continued guidance and support during my entire graduate study. He taught me many things about how to become a good researcher and he provided me numerous fruitful discussions to develop my work. When I got some achievements, his encouragement drives me to go further; when I encountered some difficulties, his patience and profound knowledge help me overcome these obstacles. I appreciate the countless hours that he spent to discuss with me, to modify my writings, to improve my presentations, and even to stay up together with me before conference deadlines. I also thank him for his consideration. When my father was in hospital, he allowed me to go back to home several times to take care of my family.

My gratitude also goes to Professor Tan Kian-Lee and Professor Lee Mong Li, who are members of my evaluation committees. They provided me valuable feedback to refine my research work. I also want to thank Professor Zhou Aoying who recommended me to National University of Singapore and Professor Ooi Beng Chin who provided me the opportunity to study here.

I would like to sincerely thank many friends in NUS for the inspiring discussions

contributing to my research work and many enjoyable hours we spent together for the leisure time. They are Cheng Weiwei, Chen Su, Wang Xianjun, Gu Yan, Xiang Shili, Yang Xiaoyan, Xia Chenyi, Yu Bei, Chen Ding, Li Yingguang, Xu Linhao, Chen Yueguo, Sun Chong, Zhang Zhenjie, Ghinita Gabriel, Ni Wei, He Qi, Cao Yu, Wu Sai, Sheng Chang, Liu Bin and many others not appearing here. I also want to thank my previous and current housemates : Guo Shuqiao, Liu Chengliang, Huang Yicheng, Yu Jie and Xiao Lei. They provide me a happy and warm home. Special thanks to my friends Dai Siwen, Li Xiang, Gao Ying, Zhang Xinyi, Zhuang Lei, Xiao Da and Huang Yinyan. The cares from them, the chats with them and the warm words in their emails accompany me through the deepest mourning time.

Last but not least, I feel deeply indebted to my parents. They are always trusting me, supporting me and missing me. When my father was fighting against the terrible cancer, he still cared about me and encouraged me to be strong. He left me at last and it is my greatest regret that he can not attend my commencement. I dedicate this dissertation to him. May he rest in peace.

Contents

Acknowledgement **i**

Summary **vii**

1 Introduction **1**

1.1 Content-based XML Dissemination 4

1.2 Motivation 6

1.2.1 Global Optimization for XML Data Dissemination 8

1.2.2 Handling Fragmented XML Data 10

1.2.3 Handling Heterogeneous XML Data 11

1.3 Contributions 12

1.4 Organization 14

2 Preliminaries **15**

2.1 Extensible Markup Language (XML) 15

2.2 XPath Expressions 17

2.3 Content-based Routing of XML Data 18

2.4 Document Dissemination and Subscription Aggregation 24

3 Related Work **28**

3.1 Improving the Matching Efficiency in Dissemination Systems 29

3.1.1 Approaches to Share Processing 32

3.1.2 Approaches to Reduce the Number of Queries 39

3.1.3 Approaches to Reduce the Matching Complexity 41

3.2 Extending the Functionalities of Dissemination Systems 44

3.3 Query Processing Using Annotations 48

3.4 Query Processing on Fragmented XML Data 50

3.5 Query Processing on Heterogeneous Data 53

3.6 Summary 56

4 Global Optimization for XML Data Dissemination **57**

4.1 Introduction 57

4.2 Overview of Piggyback Optimization 61

4.3 Types of Annotations 62

4.3.1 Positive Annotations 63

4.3.2 Negative Annotations 66

4.3.3 Impact on Matching Protocol 67

4.4 Generating Annotations 69

4.4.1 Positive Subscription Annotation (PS) 71

4.4.2 Positive Data Annotation (PD) 73

4.4.3 Negative Subscription Annotation (NS) 74

4.4.4 Negative Data Annotation (ND) 74

4.4.5 Annotation Selection 75

4.5 Processing Annotated Documents 79

4.5.1 Processing Annotations $A_{i,j}$ 80

4.5.2 Processing Document D 81

4.5.3 Deriving Negative Annotations 82
4.6 Experimental Study 83
4.6.1 Experimental Testbed 83
4.6.2 Experimental Results 85
4.7 Summary 92

5 Handling Fragmented XML Data 94

5.1 Introduction 94
5.2 Preliminaries and Definitions 96
5.3 Overview of Disseminating Fragmented XML Data 98
5.4 Algorithm for Processing XML Fragments 100
5.4.1 XML Fragmentation Model 100
5.4.2 Fragment Header Information 101
5.4.3 Identifying Relevant Fragments 104
5.4.4 Scheduling Fragment Query Evaluations 106
5.4.5 Evaluating Queries in Fragments 109
5.4.6 Dynamic Optimizations 119
5.5 Experimental Study 122
5.5.1 Experimental Testbed and Methodology 122
5.5.2 Experimental Results 124
5.6 Summary 132

6 Handling Heterogeneous XML Data 133

6.1 Introduction 133
6.1.1 Data Integration Problem 134
6.1.2 Query Relaxation Problem 137
6.2 Data Rewriting Framework 138

6.2.1 System Architecture 139
6.2.2 Data Rewriting Approaches 140
6.2.3 Schema Mapping 145
6.2.4 Data Rewriting Operators 147
6.2.5 Deriving Data Rewriting Operators 150
6.3 Implementation Issues 151
6.3.1 Non-intrusive Dynamic Data Rewriting 151
6.3.2 Intrusive Dynamic Data Rewriting 156
6.4 Experimental Study 160
6.4.1 Experimental Testbed 161
6.4.2 Experimental Results 162
6.5 Summary 169

7 Conclusions **171**

7.1 Summary 171
7.2 Contributions 173
7.3 Future Work 173

Summary

The Internet has considerably increased the scale of distributed information systems, where information is published on the Internet anywhere at anytime by anybody. To avoid overwhelming users with such huge amount of information, content-based dissemination systems have emerged, where users subscribe a set of queries to the system to express the kinds of information they are interested in and the dissemination system will automatically deliver newly published information to the proper users. With the emergence of XML, it quickly becomes the standard for data exchange on the Internet. There is a new trend to publish the data contents in XML format and to provide users with a more expressive subscription language as such XPath to address both the content and the structure of the data, which makes the content-based dissemination of XML data increasingly important.

This dissertation focuses on content-based dissemination of XML data systems. The effectiveness of such dissemination systems involves two aspects, i.e. the efficiency of the system and the functionalities that they provided. The adoption of XML data in the system increases the complexity of subscription matching at each router. While various approaches have been proposed to improve filtering efficiency, these approaches focus on optimizing the filtering locally at each individual router. In this dissertation, a global optimization approach is proposed that uses

the piggybacked annotations to enable collaborative filtering among routers.

With respect to the functionalities provided by the system, this dissertation focuses on resolving two limitations of existing dissemination systems. Firstly, due to the limitation that only complete XML documents are handled in current dissemination systems, this thesis presents a three-step approach to match a set of XPath-based subscriptions on fragmented XML data in content-based dissemination, which is to satisfy the requirements for the resource-constrained mobile devices or sensors for accessing data in terms of XML fragments. Secondly, due to the implicit assumption that all published information within the same domain conforms to the same DTD in current dissemination systems, this thesis introduces a *data-rewriting* architecture to resolve the heterogeneous schema problem in the content-based dissemination of XML data.

We have implemented these approaches, and conducted extensive experimental studies to demonstrate the efficiency and effectiveness of these approaches. We believe that our research helps to significantly improve the efficiency and to effectively extend the functionalities of the content-based XML data dissemination system, which makes this system more practical and useful.

List of Figures

1.1 The Architecture for Content-based XML Dissemination 5
1.2 Motivations for the Proposed Approaches 8
1.3 Two Sample XML Documents 11

2.1 An Example XML Document 16
2.2 The Tree Structure for XML Document in Figure 2.1 16
2.3 Content-based routing of XML data 19
2.4 An Example for SAX Parser 20
2.5 The Example for XTrie 23
2.6 Data Dissemination Example 25

3.1 The Design Space of Our Works 29
3.2 XFilter and YFilter Example 31

4.1 Types of Annotations 63
4.2 XPath Subscriptions, XML Document, and Routing Tables 65
4.3 Generating & Processing Annotations 70
4.4 Experimental results for different dissemination approaches 85
4.5 Experimental results for different DTD 88

4.6 Effect of bandwidth & number of subscriptions 89
4.7 Effect of data size & subscription complexity 90
4.8 Effect of k & θ 91

5.1 Fragmentation and query models 97
5.2 Overview of processing XML fragments 99
5.3 Fragment Header Information (a) `Edge` (b) `Prefix` (c) Additional column for `Prefix+Level` 102
5.4 Relevant Fragment-Query Node Information 103
5.5 Example queries for maximum-matching 110
5.6 Algorithm for query evaluation on fragments 114
5.7 Algorithm for propagation 115
5.8 Tree patterns and their sharing prefix tree 117
5.9 Comparison of fragmentation header schemas 124
5.10 Comparison of fragmentation with non-fragmentation 125
5.11 Comparison of scheduling policies 126
5.12 Effect of dynamic optimizations, document size, D_{XMark} 127
5.13 Performance for multiple queries, D_{XMark} 129
5.14 Effect of Scheduling Window Size and Transmission Delay, D_{XMark} 131

6.1 Query rewriting approach (QRA) 135
6.2 Data Rewriting Approaches 140
6.3 Example Schema Mapping $M_{\ell,g}$ 146
6.4 Rewriting D_ℓ to D_g with Exchange(article,author) 149
6.5 The Example for Exchange Operation 154
6.6 IDDR Example 158

6.7 Comparison of different schema mechanisms & data rewriting approaches 163
6.8 Effect of document size and number of subscriptions per router . . 165
6.9 Effect of network topology 166
6.10 Experimental Results 168

Chapter 1

Introduction

Distribution is the natural character of the Internet or intranets. Participants at different locations can join the distributed systems to provide data or consume data from the system, which is called *distributed information system*. In this distributed information system, participants need some communication mechanism to interact. Traditional communication mechanism leverages a kind of *pull-based* technique, in which the data consumer actively sends a request to the data resource to get the information from the data producer and the data producer responses the consumer by sending back the information after processing the request, such as the communication through remote procedure calls (RPC) [25, 108]. There are several limitations for this kind of communication mechanisms :

- The pull-based communication involves *synchronous* communication among the data consumers and data producers. For example, RPC requires that the data producers and consumers are active synchronously, and the consumers have to wait for the response from producers after sending the requests. Such kind of communication mechanisms incurs the inflexibility of the distributed information system, and limits the scalability of the distributed applications.

- In the pull-based model, the data consumer has to continually poll the server to obtain the up-to-date information. It may not only incur huge spikes of the load at the server, but also overwhelm the data consumers in the large amount of the information due to the information exploding nowadays.

The proliferation of the Internet has considerably increased the scale of the distributed information system. Currently, it is not uncommon that the distributed information system is at the level of thousands of participants which may be distributed worldwide and be on-and-off the distributed system asynchronously. Clearly, the pure pull-based communication model is inappropriate to satisfy the trends of the Internet. Therefore, there is a profound change for the communication to move from the pure pull-based model to a push-based model [29], which is also mentioned as *dissemination-based model*. The dissemination-based communication model leverages the publish/subscribe mechanism [86]. In publish/subscribe architecture, publishers (i.e. data producers) generate the information to the system without knowing the destination of such information; subscribers (i.e. data consumers) express their interests to the system, and then the information from various publishers that matches their interests will be delivered to them by the system. The data producers and data consumers in the dissemination-based communication is *loosely-coupled*, *asynchronous* and *anonymous*, which makes it more suitable for the modern internet application.

Based on the different ways to specify the interests of subscribers, the dissemination systems are typically classified into two categories, i.e. *topic-based dissemination* and *content-based dissemination*.

- **Topic-based dissemination :** this is the earlier version of dissemination system, and has been implemented by many industrial solutions, such as VITRIA [103], TIB/Rendezvous [109], JEDI [44]. Publishers associate some

keywords with each message to indicate the topic the message belongs to; subscribers express their interests using *keywords*. Then all messages belonging to a topic will be delivered to the users who subscribe to this topic.

- **Content-based dissemination :** the topic-based dissemination only offers a coarse-grained dissemination schema. The content-based dissemination improves the expressiveness by allowing the subscribers to use some subscription language to address the content of the information in which they are interested. In topic-based dissemination, the information is delivered towards a group of users; while in content-based dissemination, the information is delivered towards each individual user. The content-based dissemination guarantees the users to receive accurate information they are interested in, which makes it more attractive than the topic-based dissemination. A variety of content-based dissemination systems are implemented by academic or industry, such as Gryphon [24], Siena [37], Elvin [100] and ONYX [50].

The initial content-based dissemination leverages a predicate-based format for the content of the information and the subscriptions, such as *Le Subscribe* [54], Gryphon [24] and Siena [37]. Specifically, the content of the information is a set of attribute-value pairs and the subscriptions are a set of predicates to specify the constraints over values of the attributes. Recently, with the emergence of XML [12], it quickly becomes the *de facto* standard for data exchange on the Internet. There is an increasing interest to publish the information in the format of XML and use a more expressive subscription language such as XPath [11] that can address both the contents and structure of the published XML document. Various approaches using different techniques have been proposed to handle the efficient matching problem in content-based XML dissemination. For example, XFilter [20], YFilter [49], YFilter* [117] and *XMILK* [63, 60] convert the set of queries to automata;

WebFilter [88], XTrie [39], Predicate-based [69] and AFilter [36] index the common parts in different queries; BloomFilter [59] makes use of the properties of Bloom filter, and FiST [71] and BoXFilter [83] converts XPath to sequences to simplify matching. There also exists some commercial products of XML routers, such as XmlBlaster [17], DataPower [2] and Sarvega [8]. Due to the advantages of content-based dissemination for modern distributed information systems and as XML becoming the universal language for data exchange on the web, it becomes clear that the content-based dissemination of XML data will attract increasing interests from both research and industry. This thesis focuses on the content-based dissemination of XML data, and proposes approaches to optimize and extend the content-based dissemination of XML data.

1.1 Content-based XML Dissemination

In the content-based XML dissemination, the information is published as the XML documents and the subscriptions are expressed using some XML query language such as XPath or XQuery. Figure 1.1 illustrates the architecture for a content-based XML dissemination system. There are three components in the system :

- **Publishers :** The left part in Figure 1.1 shows the data publishers, which are also called the data producers for the system. They generate the information and encode it as XML documents, and send the XML documents to the system. Many applications can work as publishers, such as newspapers, databases, libraries, mobile sensors, etc. Various publishers generate the XML documents independently, thus XML documents for the same domain by different publishers may conform to different schemas. The publishers can also associate headers with the XML documents to provide additional

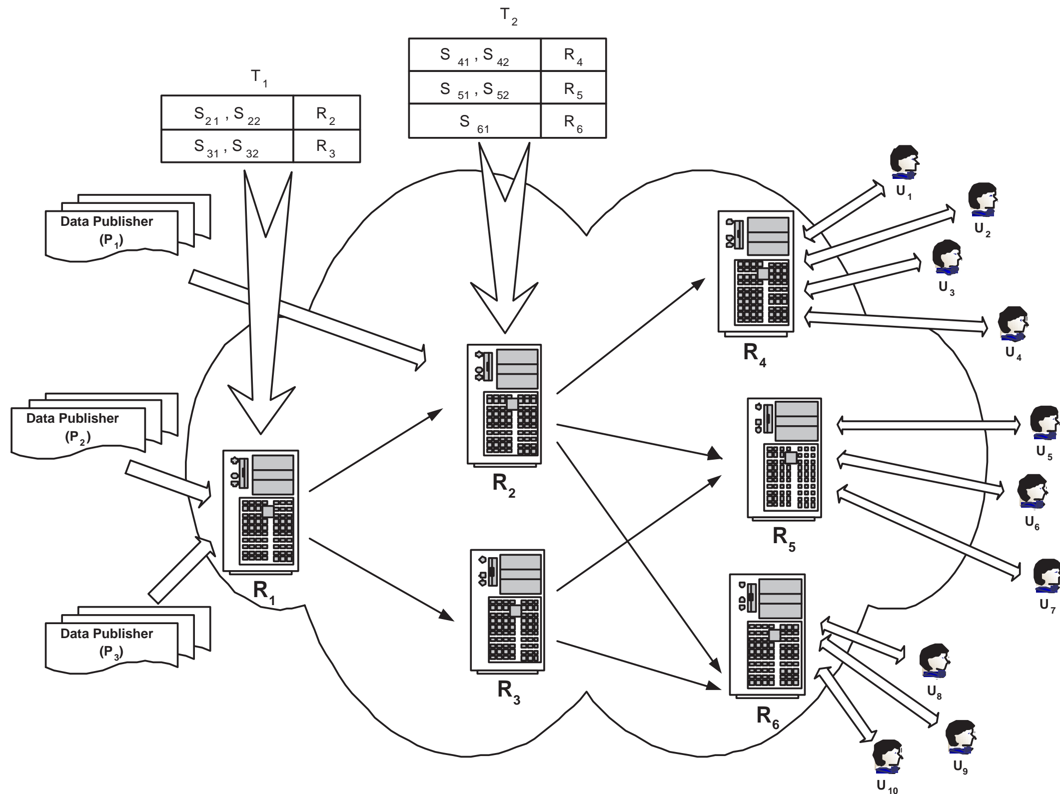


Figure 1.1: The Architecture for Content-based XML Dissemination

information for authentication, to improve the processing on the routers, etc.

- **Subscribers :** The right part in Figure 1.1 gives the subscribers which are also called the data consumers, who receive the information from the data publishers. The subscribers register their interests to the system by subscribing their profiles to the system. In the XML dissemination, their profiles are rewritten using some XML query language such as XPath [11] or XQuery [13]. The subscribers would receive all and exactly the information that matches their subscriptions. When the subscribers do not want the information anymore, they need to unsubscribe their queries.

- **XML Routing Network :** The central part in Figure 1.1 illustrates the XML routing network, which contains a set of XML routers that are inter-

connected. Each XML router receives the subscriptions from end-users or other XML routers; and receives the XML documents from the publishers or other XML routers. A routing table is stored at each router to store the set of queries subscribed to the router, and the routing table also maintains the information about the destination of a document if the document matches some query in the table. For each incoming document, the router parses the XML document to match all the queries. If a router R_i determines that document d matches a query q which is subscribed from router R_j, then R_i will forward d to R_j. Here R_i is considered as the *upstream router* of R_j and R_j is considered as the *downstream router* of R_i.

1.2 Motivation

Efficiency of the system. Content-based dissemination system is to update the data consumers with the newest published information. Some information is only useful for a small period. For example, in the stock market, the stock quote is changing frequently, users are only interested in the most up-to-date stock quote; also in monitoring systems, users should be alerted about abnormal events immediately so that they can response in time. Therefore, the efficiency of dissemination is critical. To disseminate XML data and to use XPath queries as the subscriptions improves the expressiveness of the dissemination. However, matching XPath queries with XML documents incurs larger processing cost than matching simple predicates with attribute-value pairs. Several approaches are proposed to handle the efficient matching problem for XPath queries [20, 39, 49, 117, 63, 60, 69, 36, 59, 71]. All these approaches exploit only the optimization of processing on each individual router. Actually, many routers collaborate to achieve the dissemination, which

motivates the investigation on the collaboration among routers to optimize the query processing globally.

Functionalities of the system. Besides the efficiency issue of the dissemination, the functionalities provided by the system is also an important aspect to consider. We have observed the following two limitations :

1. One limitation of existing dissemination systems is that they only accept the information that is published as complete XML documents. However, applications involving sensor devices typically collect and process data in fragments. This motivates the work for handling fragmented XML data in content-based dissemination.

2. Another limitation is that existing dissemination systems assume that all published XML documents for the same domain conform to the same schema [15] or DTD [12]. However, different publishers generate XML documents individually such that it is not uncommon that there exists the heterogeneity in both the structure and content of XML documents. The router has to handle the matching of queries on heterogeneous data.

Figure 1.2 illustrates the relationship of the work in this thesis with existing approaches. This thesis investigates the global optimization to further improve the dissemination efficiency. Additionally, this thesis extends the functionality of the dissemination system by handling the dissemination of the fragmented XML data and heterogeneous XML data. The following sections elaborate the motivations for each work in detail.

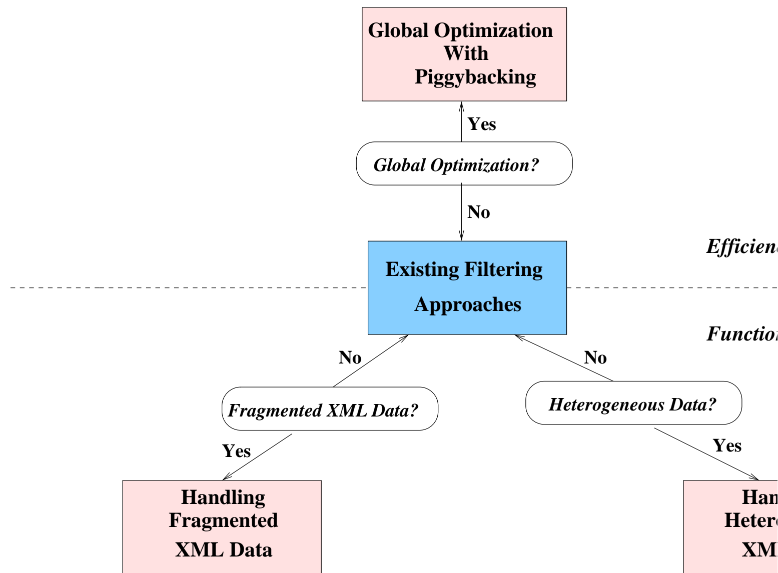


Figure 1.2: Motivations for the Proposed Approaches

1.2.1 Global Optimization for XML Data Dissemination

As aforementioned, the effectiveness of existing approaches for matching subscriptions are limited to only locally improving the performance of each individual router. Specifically, the fact that routers are interconnected and related are not being fully exploited to optimize the subscription matching.

Consider how an XML document D is being routed from an upstream router R_i to a downstream router R_j in a typical content-based XML dissemination system. On receiving D, R_i parses and processes D against the set of subscriptions S_i stored in its routing table. Once a matching subscription $s \in S_i$ (that is maintained on R_j) is detected, R_i then forwards D to R_j. A similar processing of D is then repeated at R_j but with the matching now being done against a different set of subscriptions S_j in R_j's routing table.

Two observations can be obtained on the matching and routing process.

- Firstly, the overall processing being done at different routers during the dissemination of a document can be viewed as essentially processing the same

data (i.e., XML document) against a sequence of collections of queries (i.e., sets of subscriptions along each path of forwarding routers).

- Secondly, the sequence of collections of queries being processed are not independent as they are partially related by a "containment property" that determines whether or not a document is to be forwarded to a downstream router. Specifically, the set of subscriptions S_i and S_j are related in that the subscriptions S_j in the downstream router are being aggregated (or summarized) into a smaller set of subscriptions S'_j that is stored in the upstream router R_i's routing table (i.e., $S'_j \subseteq S_i$) such that if a document D does not match any of the subscriptions in S'_j, then D will certainly not match any of the subscriptions in S_j (i.e., S'_j is "contained by" by S_j). Consequently, R_i needs to forward D to R_j only if D matches some subscription in S'_j.

Thus, given that the same document D is being processed against related sets of subscriptions, each upstream router R_i can help to optimize the performance of its downstream router R_j (and thereby reduce the overall processing time to deliver D to relevant subscribers) by passing along some useful information to R_j (about D as well as the about related queries that R_i has processed) when it forwards D to R_j. R_j can then try to exploit the hints that it receives from R_i to optimize its own processing of D. The first work in this thesis optimizes the dissemination by piggybacking annotations (i.e. hints) with the XML documents. This work exploits the collaboration among different routers, which can be considered as *global optimization*.

1.2.2 Handling Fragmented XML Data

The popularity of the mobile devices, such as mobile phones, laptops and personal digital assistants, and the advance of the wireless networks has fostered the increasing use of mobile devices in current distributed systems. Some work have addressed the dissemination in a mobile environment [45, 70]. To employ the resource-constrained mobile devices for accessing and monitoring data requires a memory-efficient technique to process queries on *fragmented* data. Furthermore, the data collected by sensor devices is often in fragments such that the querying should be performed on the fragmented data. For example, in a military battlefield, many mobile sensors are equipped to report the fragment of information for their monitored locations. The information from various sensors forms the complete information for the battlefield. Besides the above scenarios that the data is fragmented by nature, disseminating XML data in fragments is also motivated by the efficiency to propagate updated data without resending the entire document.

The size of the collection of queries being matched can vary depending on the application context. A small-scale deployment can arise in specialized monitoring applications that run on mobile devices, while a large-scale scenario can arise in middleware-based applications that disseminate data to a large number of different users based on their subscriptions. While the first scenario necessarily requires the data to be fragmented for it to be processed by resource-limited devices, the second scenario can also benefit from using fragmented data as this can enable more opportunities for query optimization by exploiting the structural relationships among the fragments to minimize unnecessary and redundant processing.

While there has been some research that addresses general query processing issues on fragmented data [97, 95, 96], we are not aware of any work that examines the problem of matching boolean XPath queries on fragmented XML data. The

more specialized nature of processing boolean queries on fragmented XML data opens up new opportunities for query optimization and processing. The second work in the thesis addresses the problem of matching XPath-based subscriptions on *fragmented* XML data, where the published XML data is being disseminated in terms of a collection of disjoint fragments.

1.2.3 Handling Heterogeneous XML Data

In content-based dissemination , data publishers and data consumers are *loosely-coupled*, *anonymous*, and do not necessarily agree on the same schema. Data consumers may have no knowledge about the schemas from data publishers, and various data publishers generate and publish their data independently. Therefore, publications from different publishers may conform to heterogeneous schemas although they satisfy the same kinds of users' interests. Thus, although the users' subscriptions do not exactly match the publications, the publications do satisfy the users' interests.

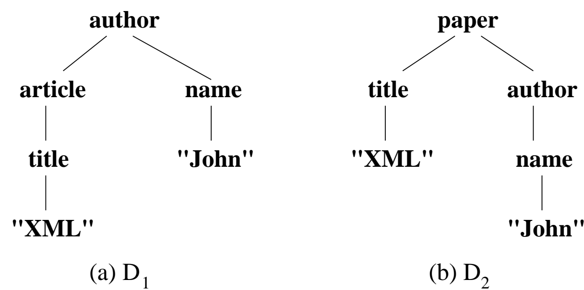


Figure 1.3: Two Sample XML Documents

For example, Figure 1.3 gives the XML documents D_1 and D_2 from two data publishers. Suppose a user is interested in the information about the papers from author "John", thus the user submits a subscription using the XPath expression like /author[name = "John"]/paper/title. We know that items *paper* and *article* have the same meaning, which makes D_1 satisfies the user's requirement; and D_2

also provides the information about the papers from author "John", thus it should also be forwarded to the user. However, the existing dissemination systems fail to forward any of these documents to the user, since none of the approaches consider the probable semantic and structural heterogeneity in schemas among data publishers and users.

In the large-scale distributed system, it is not uncommon to have heterogeneous data from various publishers who may be unaware of one another. There is indeed a requirement for the system to handle such heterogeneous data, while the supporting of the heterogeneous data should not be at the cost of the dissemination efficiency. An approach is proposed in this thesis to handle the problem of efficient dissemination of XML data while there exists heterogeneity in schemas. Besides forwarding the XML data that match the subscriptions exactly to users, the data whose semantic meanings satisfying the users' interests is also forwarded to the users.

1.3 Contributions

The major contributions of this dissertation are three-fold :

1. A novel, holistic optimization technique for XML data dissemination called *piggyback optimization* is proposed. This approach enables upstream routers to pass useful hints in the form of document header annotations to optimize the performance of downstream routers. This new optimization is orthogonal to the existing approaches for matching queries efficiently on each individual router. Two types of annotations are proposed in this approach, i.e. *positive annotations* and *negative annotations*. Various annotations for each type are provided and studied. These annotations help to improve the filtering

efficiency on the downstream router either by detecting a matching query earlier to forward the document without being parsed or by eliminating the non-matching queries to reduce the number of processed queries. A comprehensive experimental study is provided to demonstrate the efficiency of piggyback optimization. This work has been published in SIGMOD 2007 Conference [41].

2. A comprehensive study on matching XPath-based subscriptions *directly* on fragmented XML documents without reconstructing the original documents is presented. The approach extends the functionality of the content-based dissemination system to handle data that is disseminated in fragments. Additionally, by exploiting the optimization to process only the relative fragments for query evaluations, the filtering efficiency on each router is improved. The optimizations based on the dynamic query processing results are proposed to further improve the filtering performance. The experimental results using both synthetic and real-life datasets show that the fragmented approach outperforms the traditional non-fragmented approach by up to a significant margin. This work has been published in ICDCS 2006 Conference [40].

3. A novel framework leveraging dynamic data rewriting is proposed to handle the efficient dissemination of heterogeneous XML data. Existing approaches for query processing on heterogeneous data use the query rewritten mechanism, which is not suitable for the dissemination scenario where a large number of queries are evaluated simultaneously. Eight operators for performing data rewriting are proposed, which cover a reasonable set of semantic and structural heterogeneity in XML schemas. The algorithm to perform these data rewriting operators dynamically during the parsing of the document to evaluate queries is provided. Besides the dynamic data rewriting approach,

other alternative approaches in terms of when and how to perform the data rewriting are also exploited. An extensive performance study is conducted to compare the dynamic data rewriting approach with other approaches. The results on both simulation and real network verify the effectiveness of the dynamic data rewriting approach. This work has been submitted for publication [85].

1.4 Organization

The rest of the thesis is organized as follows. Chapter 2 provides background knowledge for the work conducted in this thesis. A survey of related work for various approaches in content-based XML dissemination is presented in Chapter 3. The related work for each particular work in the thesis is also discussed in Chapter 3. Chapter 4 presents the piggyback optimization for content-based XML dissemination. Chapter 5 introduces the approach of matching XPath-based subscriptions when the published XML data is being disseminated in terms of a collection of disjoint fragments. Chapter 6 introduces the dynamic data rewriting approach to handle the efficient dissemination of heterogeneous XML data. Finally, Chapter 7 concludes this thesis and points out some directions for future work.

Chapter 2

Preliminaries

This chapter presents further background information for the work in this thesis. Firstly, this chapter introduces the XML, which is the data format to publish information in content-based XML dissemination. Secondly, the subscription language used in the thesis, i.e. XPath, is presented. After that, the matching approaches performed by the router to detect the matched subscriptions are introduced. Finally, this chapter also introduces how the subscriptions are aggregated and propagated in the dissemination system.

2.1 Extensible Markup Language (XML)

XML (stands for *eXtensible Markup Language*) is a markup language. Instead of focusing on how to display the data as in HTML, XML is designed to describe data and focus on what data is. XML is *self-describing*, *machine-readable* and *extensible*, which makes it quickly become a *de facto* for data exchange on the Internet.

XML documents are composed of markup and content. The most common markup is the element. An element begins with a start-tag $< \mathit{element_name} >$, and ends with an end-tag $< /\mathit{element_name} >$. Attributes, which are name-pairs, can

occur inside start-tags after the element name, e.g. $< book, class = \text{“}whodunit\text{”} >$. Content, which is text data, can be enclosed between tag-pairs. Elements can be nested in any depth in XML documents, but they must be well-nested, i.e. if the start-tag of an element n_i occurs in the tag-pair of another element n_j, the end-tag of n_i should also occur in the tag-pair of n_j. Every XML document has a root element, and it can not be contained in any other element. Figure 2.1 gives an example XML document providing the course information.

```
<?xml version = “1.0”? >
<Courses>
        <Course Code = “CS3230”>
                <Title> Database Management < /Title>
                <Instructor>
                        <Name> Jim < /Name>
                        <Email> jim@comp.nus.edu.sg < /Email>
                < /Instructor>
                <Time> Wed, 16:00 - 18:00 < /Time>
                <Location> LT33 < /Location>
        < /Course>
< /Courses>
```

Figure 2.1: An Example XML Document

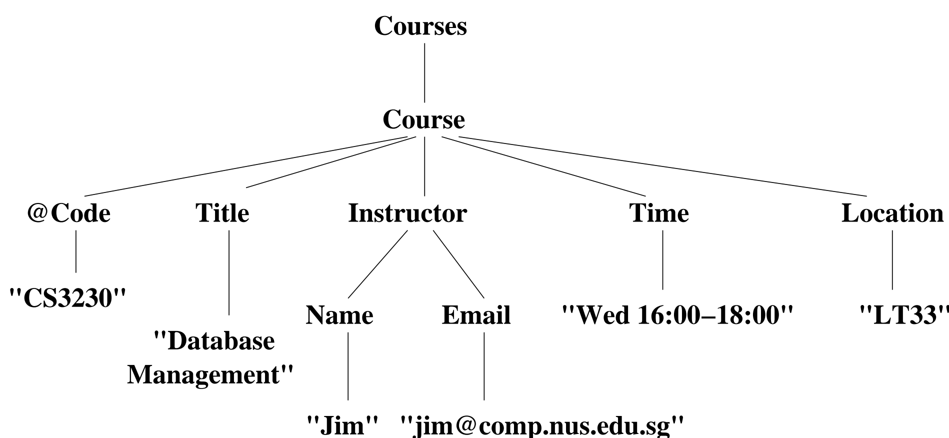


Figure 2.2: The Tree Structure for XML Document in Figure 2.1

The XML document can be modeled by a tree structure due to its hierarchical

structure. Figure 2.2 shows the tree structure for the XML document in Figure 2.1. The root of the document is the root of the tree. The element/subelement relationship in the document is modeled as the parent/child relationship in the tree. Attributes are represented as children of their associated elements and contents are represented as children of their associated elements or attributes.

2.2 XPath Expressions

XQuery [13] and XPath [11] are the query languages provided to address on the XML document. The core component of XQuery is the XPath expressions and most existing filtering approaches [20, 39, 49, 117, 63, 60, 69, 36, 59, 71] handle a fragment of XPath expressions. Thus this thesis focuses on the XPath expressions. The XPath language treats XML documents as a tree of nodes (corresponding to elements) and offers an expressive way to specify and select parts of this tree. XPath expressions are structural patterns that can be matched to nodes in the XML data tree. The evaluation of an XPath expression yields an object whose type can be a node-set, a boolean, a number, or a string. For subscription matching purpose in content-based dissemination, an XML document matches an XPath expression when the evaluation result is a non-empty node set.

The simplest form of an XPath expression specifies a single-path pattern, which can be either an absolute path from the root of the document or a relative path from some known location (i.e., context node). An XPath expression is composed of one or more location steps. A location step has three parts: an axis, a node test, and zero or more predicates. A node test specifies the node types and node names selected by the location step. The wildcard "*" can be used as the node test to match any node names. An axis specifies the hierarchical relationships between

the nodes selected by the location step and the context node.

This thesis focuses on two main axis operators in XPath: parent-child operator "/" specifies the nodes at the adjacent level of the context node; ancestor-descendant operator "//" specifies the nodes separated by any number of levels from the context node. Considering the XPath expression Q_1.

$$Q_1 : /Courses/Course//Title$$

It addresses all *Title* elements descendant of element *Course* which is the child of element *Courses*.

Each location step can also include one or more predicates to further refine the selected set of nodes. Predicate expressions are enclosed by "[" and "]" symbols. The predicates can be applied to the text or the attributes of the addressed elements. The predicates may also include other path expressions, which makes the XPath expression to be a *tree pattern query*. Any relative paths in a predicate expression are evaluated in the context of the element nodes addressed in the location step at which they appear. Considering the XPath expression Q_2 shown as follows.

$$Q_2 : /Courses/Course[@Code = \text{``}CS3230\text{''}][Instructor/Name]/Title$$

It specifies a tree-structured pattern starting at the root element *Courses* with two children "branches" *Course/Title* and *Course/Instructor/Name* such that the element *Course* has an attribute *Code* with the value to be "CS3230".

2.3 Content-based Routing of XML Data

In content-based dissemination, the routers take charge of matching a collection of XPath expressions on them with each incoming XML document. There are a batch of approaches proposed to efficiently match the set of XPath expressions [20, 39, 49,

117, 63, 60, 69, 36, 59, 71]. In traditional query processing, the XML documents are stored statically in the database and some kinds of indexes for the documents may be provided. The indexes are exploited or the documents are navigated to process each query. While in content-based dissemination, a large number of subscriptions are relatively static on the routers and these subscriptions are indexed for efficient evaluation. The XML documents continuously arrive the routers as streams from publishers or other routers, and these documents are parsed to match the set of subscriptions on the routers. Figure 2.3 shows a schematic diagram of the key components in a typical content-based router. An incoming XML document D is first parsed by an event-based XML document parser. The parsed events are used to drive the matching engine which relies on some efficient index on the subscriptions to quickly detect matching subscriptions in its routing table; D is then forwarded to neighboring routers and local subscribers with matching subscriptions.

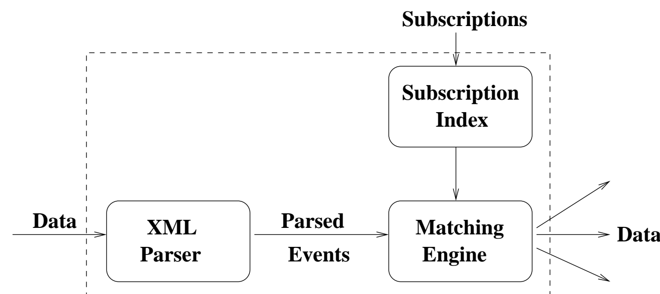


Figure 2.3: Content-based routing of XML data

The SAX API [9] is used to parse the XML document on-the-fly in the dissemination. The SAX API brings the following two advantages :

- The query processing is started immediately once the XML document arrives. There is no need to wait for the receiving of the complete document, which improves the response time considerably.

- There only incurs small memory usage in SAX API, which makes the router

be able to handle large XML documents.

start_document	
start_element	Courses
start_element	Course Code = “CS3230”
start_element	Title
characters	Database Management
end_element	Title
start_element	Instructor
...	
end_element	Instructor
start_element	Time
characters	Wed, 16:00 - 18:00
end_element	Time
...	
end_element	Course
end_element	Courses
end_document	

Figure 2.4: An Example for SAX Parser

SAX provides a mechanism for reading data incrementally from an XML document. The XML stream is accessed unidirectionally such that the previously accessed data can not be re-read unless re-parsing the document. The SAX parser is implemented using an event-driven model in which the developer provides the callback methods with respect to events which are invoked by the parser as it serially traverses the document. There exists several SAX API implementations, such as Apache Xerces [1] and Libxml [4]. Figure 2.4 illustrates the sequence of events by the SAX parser for the document in Figure 2.1. There are three main kinds of events reported by the SAX API.

- *start_document/end_document* : the start_document event reports the beginning of an XML document, and the end_document event reports the end of the document.

- *start_element/end_element* : the start_element event indicates the start tag of an element, it carries the information for the name of the element, the attributes associated with the element and their values. And the end_element event indicates the enclose of the element, which corresponds to the previously nearest start_element event.
- *characters* : the characters event contains the text information between two XML tags.

All existing matching approaches utilize the SAX API to parse the XML document. As aforementioned, these approaches focus on improving the matching efficiency on each individual router. The work in this thesis focuses on the global optimization of the efficiency or the extension of the functionality of the dissemination system, thus this thesis is orthogonal with the existing approaches. To implement the approaches proposed in this thesis, the XTrie [39] filtering approach by Chan et al is used at each router. A brief introduction for the XTrie method is presented in the following.

The XTrie approach exploits the shared processing for the common substrings in the collection of XPath expressions. The sequence of XPath expressions is first decomposed into substrings. It requires that each pair of consecutive elements in substrings must be separated by a parent-child ("/") operator, and each substring has the maximal length. A substring-table (ST) is used to store these substrings. Each row in ST corresponds to one substring from some XPath expression. Physically, the substrings from the same XPath expression are clustered together and are ordered by the simple decomposition of the expression. Logically, the same substring from different XPath expressions are chained together using a linked list to facilitate the following matching process. Each substring (denoted as s_i) in ST has five attributes :

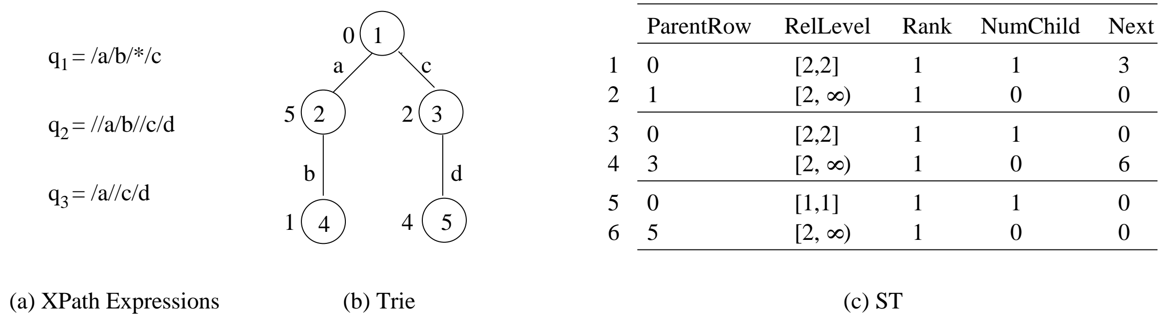
- ParentRow : specifies the row number of the substring in ST corresponding to the parent substring of s_i (If s_i is the root substring, ParentRow = 0).
- RelLevel : is the relative level of s_i with its parent substring. Let x denote the distance in document level between the last element in s_i and the the last element in s_i's parent substring, if there are "//" between s_i and its parent substring, then the RelLevel of s_i is $[x, \infty)$; otherwise RelLevel = $[x, x]$.
- Rank : the substring s_i having rank k means that s_i is the kth child of its parent substring.
- NumChild : indicates the total number of children of s_i.
- Next : is an integer indicates the row number of the substring s_j such that s_j is the first substring behind s_i satisfying the requirement that s_j is the same with s_i. *Next* is used to logically group the substrings with same labels. Actually, a linked list is formed, and the head of the linked list is substring with the smallest row number in ST.

The above five attributes are used to check the matching of substring and further the matching of XPath expressions with the XML document.

The set of decomposed substrings is indexed by a *trie* structure T. For the substrings with the same label, only the substring with the smallest row number is indexed in the *trie* T, and other substrings can be looked up using the *Next* attribute in ST. The *trie* T is a rooted tree. Each edge of T is associated with an element name, and each node N of T is labelled with a string formed by concatenating the edge labels along the path from the root node of T, which is denoted as $label(N)$. Each node N in T is also associated a value, denoted as $\alpha(N)$, which is determined as follows : if $label(N)$ corresponds to a decomposed substring, then

$\alpha(N)$ is the row number of this substring in ST; otherwise $\alpha(N) = 0$. The *trie* T is used to check whether the substring parsed from the XML document has some matchings in XPath expressions.

XTrie method needs to construct another table called substring-table (ST). When a start-element event e is encountered, the algorithm searches in the *trie* T. If there is an edge label e from the current node to a node N, the search continues on node N. For each node N visited, if $\alpha(N) \neq 0$, a matching algorithm will be invoked to check the matching of all substrings in the linked list pointed by the substring at row $\alpha(N)$ in ST. The matching algorithm uses the attributes in the ST table to check if the constraints are satisfied and return the XPath expressions that matched. On the other hand, if there is no edge out the current node labelled e, the search in the trie T will backtrack to the node that is the longest suffix of the current node to check for other potential matchings.



	ParentRow	RelLevel	Rank	NumChild	Next
1	0	[2,2]	1	1	3
2	1	[2, ∞)	1	0	0
3	0	[2,2]	1	1	0
4	3	[2, ∞)	1	0	6
5	0	[1,1]	1	1	0
6	5	[2, ∞)	1	0	0

(a) XPath Expressions (b) Trie (c) ST

Figure 2.5: The Example for XTrie

Example 2.1 Considering the collection of XPath expressions in Figure 2.5(a), the Trie structure and ST table for them are shown in Figure 2.5(b) and (c) respectively. The numbers at the left of nodes in the Trie structure point to the first rows of the substrings with the same labels in ST. Given a data path $/a/b/c/d$ in some document, when the start_elmement of a is reported, the Trie moves the current node from node 1 to node 2. Then the number 5 is used to find the substring, i.e. a,

in row 5 of the ST. Comparing the element a in the document with the information in ST, the method detects substring a is matched. When the start_element of b is reported, the node in the Trie further moves to node 4, which corresponds to row 1, i.e. substring $/a/b$, in ST. The Next attribute of row 1 is used to find the substring $/a/b$ in row 3. The processor detects that both two substrings are matched. Similarly, when the substrings c in row 2 and c/d in row 4 and 6 are matched. The matching of substrings are propagated from the child substring to its parent substring, and the RelLevel is used to check the level requirement. Finally, the processor detects are q_2 and q_3 are matched. □

2.4 Document Dissemination and Subscription Aggregation

In content-based dissemination environment, each data consumer registers his subscription to his local router. In order for a router to know about subscriptions that have been registered with other routers, a routing protocol is used by the routers in the overlay network to exchange subscription information such that their subscription tables are set up correctly to establish routing paths for forwarding documents.

As previously mentioned, the content-based dissemination system consists of three components, i.e. publishers, subscribers and a routing network. A collection of subscriptions are stored at routers to be matched with the incoming documents. We use R_i to denote a router, and T_i to denote the set of subscription entries in its routing table. Figure 2.6(b) illustrates a simple routing network with three routers R_1, R_2 and R_3. The rectangles in each router show the routing tables maintained on the router. Conceptually, each entry in T_i is of the form (S_j, p_j), where S_j denotes a set of subscriptions and p_j denotes a unique identifier that refers to either a local

subscriber of R_i or a neighboring router of R_i. For a given document D, we use $S_j^+(D)$ and $S_j^-(D)$ to denote, respectively, the subset of subscriptions in S_j that matched and did not match D (i.e., $S_j = S_j^+(D) \cup S_j^-(D)$). For each incoming document D to R_i, R_i will forward D to p_j if and only if $S_j^+(D)$ is non-empty.

If a router R_i forwards some document to a neighboring router R_j, we call R_i as an *upstream router* and R_j a *downstream router*. In order for any document to be forwarded from an upstream router R_i to a *downstream router* R_j, R_j needs to have advertised (via some routing policy) its collection of subscriptions (i.e., $U_j = \bigcup_{(S,p)\in T_j} S$) to R_i so that an entry (U_j, R_j) can be recorded in T_i.

Example 2.2 Considering the routing network in Figure 2.6(b), R_1 is the *upstream router* of both R_2 and R_3, and consequently, R_2 and R_3 are the *downstream routers* with respect to R_1. R_2 needs to advertise its collection of subscriptions to R_1, which incurs a tuple (S_2, R_2)(i.e. $\{s_5\}, R_2$ in Figure 2.6(b)) in the routing table T_1 on R_1. The document D is published to R_1 first. If R_1 detects some subscription $s_i \in S_2$ that matches D, R_1 forwards D to R_2. □

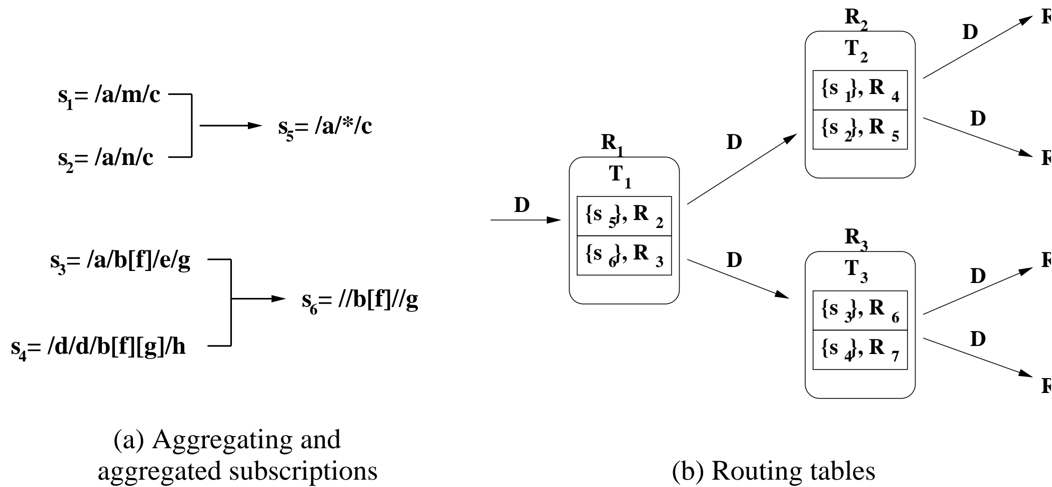


Figure 2.6: Data Dissemination Example

Since the entire collection of subscriptions in R_i (i.e., $U_i = \bigcup_{(S,p)\in T_i} S$) is gener-

ally large, R_i needs to summarize (or aggregate) U_i to a smaller set $\mathcal{S}'_i$ of aggregated subscriptions before advertising it to its neighboring routers. To preserve forwarding correctness, $\mathcal{S}'_i$ needs to satisfy the following *containment property* w.r.t. U_i: for every document D, if D matches some subscription $s \in U_i$, then there must exist some subscription $s' \in \mathcal{S}'_i$ such that D also matches s'. We say that $\mathcal{S}'_i$ *contains* U_i (or U_i is *contained by* $\mathcal{S}'_i$), denoted by $U_i \sqsubseteq \mathcal{S}'_i$. Similarly, we say that a subscription s' contains another subscription s, denoted by $s \sqsubseteq s'$, if $\{s\} \sqsubseteq \{s'\}$. The importance of the containment property (i.e., $U_i \sqsubseteq \mathcal{S}'_i$) is that using $\mathcal{S}'_i$ in place of U_i for document matching will guarantee that there are no *false negatives* (i.e., documents not being forwarded when they should); however, *false positives* can arise (i.e., documents being forwarded when they need not) which are tolerable and do not compromise correctness.

Several algorithms (e.g., [38, 117]) have been developed to aggregate a set of subscriptions S into a smaller set S' such that $S \sqsubseteq S'$, and they are all formulated (at a high level) in terms of the following two steps: first, partition S into a collection of disjoint subsets $S_1, \cdots, S_m$, where $m < |S|$; next, aggregate each S_i into a single subscription s'_i (i.e., $S_i \sqsubseteq \{s'_i\}$) to obtain $S' = \{s'_1, \cdots, s'_m\}$ with the properties that $S \sqsubseteq S'$ and $|S'| < |S|$. In addition, to ensure that the aggregated subscriptions are space-efficient, a space bound is generally impose on S' to limit the total number of query steps among all the queries in S'.

For each of the subscriptions $s \in S_i$, $S_i \sqsubseteq S'$, that becomes aggregated to $s'_i \in S'$ (i.e., $s \sqsubseteq s'_i$), we refer to s as an *aggregating subscription*, and refer to s'_i as an *aggregated subscription* of s.

Example 2.3 Consider the set of XPath expressions $S = \{s_1, s_2, s_3, s_4\}$ in Figure 2.6(a). One way to aggregate S into a smaller set is to first partition S into two subsets $S_1 = \{s_1, s_2\}$ and $S_2 = \{s_3, s_4\}$; followed by aggregating S_1 and S_2,

respectively, into s_5 and s_6 as shown in Figure 2.6(a). It can be verified that $S_1 \sqsubseteq \{s_5\}$ and $S_2 \sqsubseteq \{s_6\}$. We say that s_5 and s_6 are, respectively, the aggregated subscriptions of S_1 and S_2; and the subscriptions in S_1 and S_2 are aggregating subscriptions. □

Chapter 3

Related Work

As mentioned in Chapter 1, the content-based dissemination system has two important issues, i.e. filtering efficiency and functionality. Figure 3.1 illustrates the design space of the work in this dissertation. All existing filtering approaches [20, 39, 38, 49, 117, 63, 60, 69, 36, 59, 71] focus on optimizing the matching efficiency on each individual router, and they can only handle the XML data that are published as complete documents, and they have the constraints that all data from the same domain conform to the same XML schema. The work on global optimization of XML data dissemination exploits the collaboration among routers to optimize the filtering efficiency in a global manner; the work to disseminate the fragmented XML data extends the functionality to handle the information published in fragments; and the work to handle the heterogeneous XML data allows the publishing of data in heterogenous structure. This chapter first classifies and introduces the existing matching approaches that intend to improve the efficiency of the dissemination system. Secondly, this chapter reviews some approaches proposed to extend the functionality of the dissemination system. After that, this chapter reviews the techniques that are related with each particular work in this thesis. The related techniques are constrained in the content-based dissemination scenario.

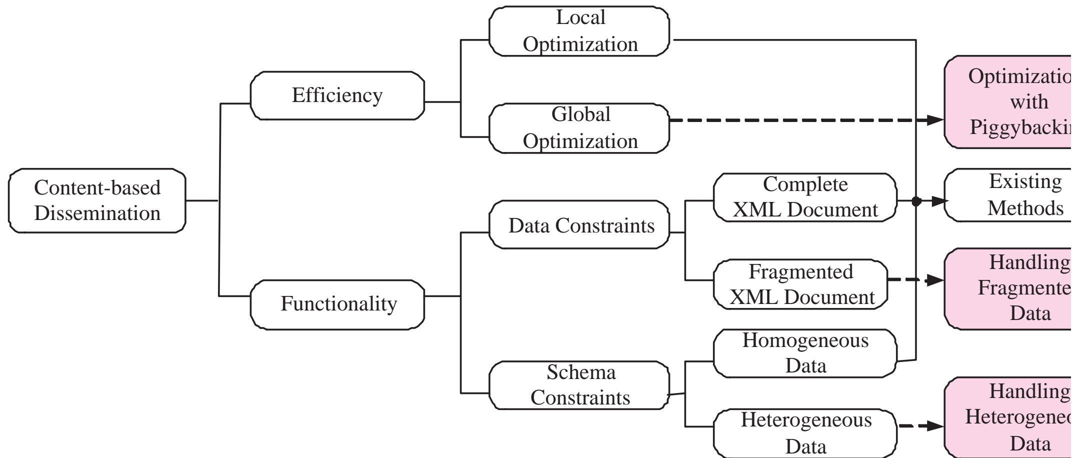


Figure 3.1: The Design Space of Our Works

3.1 Improving the Matching Efficiency in Dissemination Systems

XFilter [20] is the first approach proposed in content-based XML dissemination to handle the efficient matching of a large collection of XPath expressions. XFilter converts every query to a Finite State Machine (FSM) and uses the events reported by the SAX-based parser to drive the execution of the FSM for queries. Each state in the FSM is represented by a *path node* which corresponds to one step in the XPath expression except the "*" step. The path node contains four types of information as follows :

- QueryId : the identity of the query that this path node belongs to.
- Position : the location of this path node in the order of the path nodes in the query. The first path node is numbered as 1, and the rest of the path nodes are numbered sequentially.
- RelativePos : the level distance of this path node with its previous path node. If the axis of the current path node is "/", this value is set to be 1 plus the

number of wildcards (i.e. "*") between this path node with its predecessor path node; otherwise this value is set to be -1.

- Level : it tells that at which level in the XML document this path node should be checked. This value will be updated dynamically during the evaluation of queries.

These path nodes are indexed using a hash table based on the element names that correspond to the path nodes. Each element name is associated with two lists : the *Candidate list* and *Wait list*. For the path nodes that are the next-to-be-processed node for the query are to be placed on the candidate list of the index entry for its respective element name; and the path nodes that are to be processed in the future are placed on the corresponding wait list. The lower right part of Figure 3.2 gives seven XPath expressions. Figure 3.2(a) and Figure 3.2(b) show the decomposed path nodes and the hash table index for these queries respectively.

The state transition of FSM is driven by the events from the SAX parser. For the *start_element* event, the processor looks up the element name in the hash table and checks each path node in the candidate list of the entry. There are two kinds of checks performed on the path nodes : level check, which is to make sure that the element occurs in the level that satisfies the requirement in the path expression, and predicate check, which is to satisfy the requirement on the attributes of the element. If the path node passes these two checkings, the FSM of the query moves to a new state by copying the next path node from the wait list to the candidate list. If the current path node is the last path node in the query, this query is matched by the document. For the *end_element*, the corresponding path nodes are deleted from the candidate list.

XFilter is a basic matching approach that does not exploit the common parts in the queries such that XFilter is not very efficient to handle larger number of

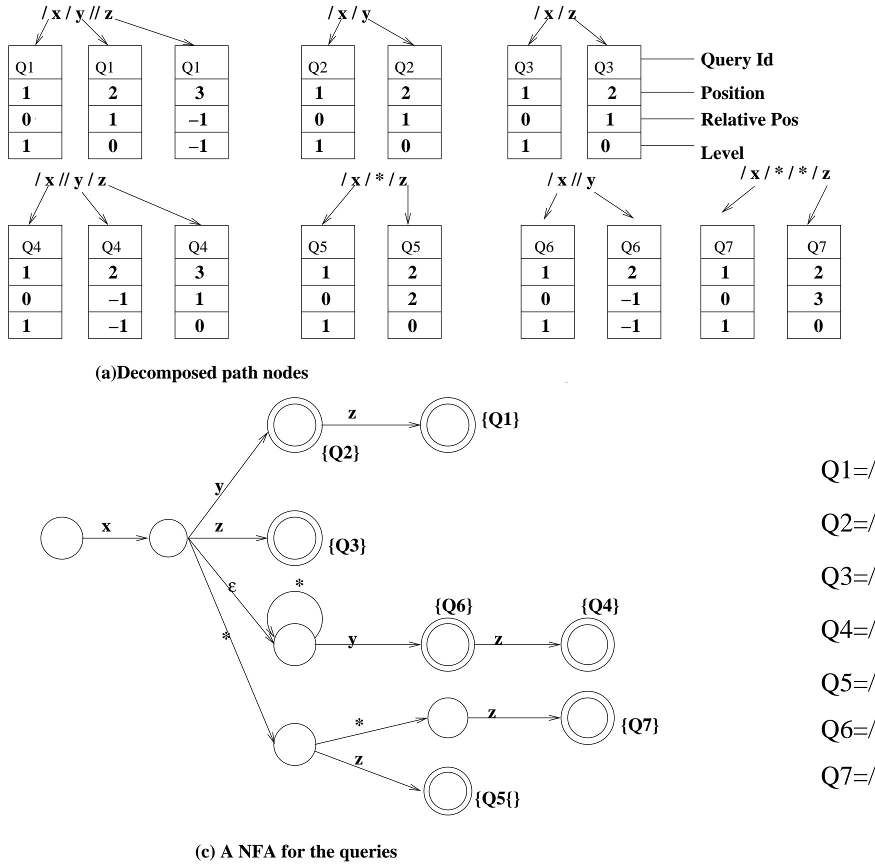


Figure 3.2: XFilter and YFilter Example

queries. Many approaches are proposed to improve the matching efficiency. These approaches can be classified into three categories based on the mechanism to optimize the matching.

- Approaches to share the processing. Each router stores a large number of XPath expressions, thus there is a high probability that the common parts exist in XPath queries. To share the processing of these common parts could improve the matching efficiency. YFilter [49], YFilter* [117] and *Lazy*DFA [60] share the processing of common prefixes; XTrie [39], Predicate-based [69] share the processing of common substrings; AFilter [36] shares the processing of both the common prefixes and suffixes; and XPush machine [63] shares the processing of common predicates. Besides the common parts in queries, the

shared processing for incoming XML documents is explored in [112].

- Approaches to reduce the number of processed queries. The matching time increases as the number of processed XPath queries grows. This kind of approaches intends to reduce the number of processed queries during the parsing of the document. The approach to use precomputed views [61, 62] and the tree pattern aggregation approach [38] belong to this category.

- Approaches to reduce the matching complexity. To match the path expressions especially the tree pattern queries incurs larger cost. Some approaches are proposed to convert the tree pattern matching to some simple matchings. The BloomFilter [59] uses the Bloom to hash the path expression and converts the matching of tree patterns to be the matching of corresponding bits. FiST [71] converts both the XPath queries and XML document to be sequence, such that the matchings can be detected using string matching.

In the following, we introduce these approaches in details.

3.1.1 Approaches to Share Processing

Shared processing of common prefixes. Diao et al [49] have proposed an automata-based approach called YFilter, which converts the collection of XPath expressions on a router to a single *non-deterministic finite state automata*(NFA). The common prefixes in different XPath expressions are combined into one path in NFA such that the processing of these common prefixes are shared, thus the filtering efficiency can be improved. Figure 3.2(c) shows the NFA in YFilter for the set of XPath expressions in the lower right part of this figure. A circle stands for a state in NFA and two concentric circles stand for a final state, which is associated with the identities of the queries it accepts. The symbol " $*$ " matches

any element in the document. The "//" axis in XPath expressions is converted to an ε transition followed by a "$*$" loop in the NFA. This figure clearly illustrates the shared processing for the common prefix $/x$ in all queries. The state transitions of the NFA in YFilter are driven by the events generated by the SAX parser. When the final state is reached, the queries associated with the final state are found to be matched. Since YFilter eliminates the redundant processing on common prefixes of the queries, it is more efficient than XFilter which shares nothing during processing.

YFilter* [117] is proposed by Zhang et al to further enhance the performance of YFilter on the processing of tree pattern queries. The NFA in YFilter can only handle the matching of single path expressions. To process the tree pattern queries, YFilter conducts an expensive post-processing to join the matching of single paths which are decomposed from the tree pattern queries. Then a large number of temporary matching results for the single paths have to be maintained in the memory, and may be discarded finally in the join procedure. YFilter* is to optimize the processing by either detecting a matching tree pattern query or discarding an unnecessary partial matching as early as possible. YFilter* additionally maintains the information for *branch point nodes* of the tree pattern queries. The checking of whether the tree pattern queries are matched is performed during the evaluation of single paths using NFA. If some tree pattern query is determined to be matched, the processing of the single paths decomposed from it can be short-circuited in the following processing. The maintained *branch point nodes* can also help to determine that a matched single path cannot generate the matching of the tree pattern query such that the matching of the single path can be discarded immediately.

Green et al [60] have proposed a method to use the *Deterministic Finite Automata*(DFA) to process the large number of XPath expressions on the routers. The DFA is built in two steps : the collection of XPath expressions is converted into

Nondeterministic Finite Automata(NFA) at first and then the NFA is converted into a DFA. The DFA approach has two variants : *eager* DFA and *lazy* DFA. The *eager* DFA method completely computes all the states before the query evaluation; while the *lazy* DFA computes the states on demand during the query evaluation. It shows that the wildcard (i.e. "*") and the descendant axis (i.e. "*") incur the significant exponential growth in the number of states, which makes the *eager* DFA prohibitive in practice. However, for the *lazy* DFA, it is theoretically proved that an upper bound of the number of states exists depending on the character of the XML data, thus the number of states in *lazy* DFA is manageable.

All the above approaches use some kind of automata. The large number of states maintained in the memory may cause the data miss in the cache when the state transitions are performed. A cache-conscious automata is proposed in [66] to improve the locality of the automation state transition. The overall performance is improved due to the higher cache hit rate.

Shared processing of substrings. The approaches in the previous section only exploit the common parts in the prefixes. The XPath expressions, however, can contain the common parts in other places. For example, the two XPath queries $/a/*/b/c$ and $/e/*/b/c$ have the commons parts $/b/c$ which is not the prefix, and such common parts will not be shared using the approaches in the previous section. The approaches XTrie [39] and Predicate-based Filter [69] can share the processing of common substrings in the XPath queries.

XTrie achieves the shared processing for common substrings by decomposing the XPath expressions into substrings. The detailed algorithm for XTrie is already introduced in Section 2.3.

The Predicate-based Filter proposes a mechanism to encode the XPath expressions using ordered sets of predicates, such that the common parts in the XPath

expressions are encoded by the same predicates, and these predicates would be stored and processed only once. Specifically, an XPath expression is represented by a sequence of predicates : $(a_1, o_1, v_1) \rightarrow (a_2, o_2, v_2) \rightarrow \ldots \rightarrow (a_n, o_n, v_n)$, where a_i represents either a tag name or a pair of tag names; o_i represents some relational operator; and v_i is the value constraints. There are four types of predicates :

- Absolute predicate (a_{n_i}, o, v) : specifies the position of the tag n_i in the XPath expression. This predicate is used to encode the first tag name in the XPath expression. The operator o can be either $\geq$ or $=$. The $=$ is used to encode the absolute position of the tag n_i. For example, $/*/n_i$ is encoded as $(a_{n_i}, =, 2)$. The $\geq$ is used to encode the descendant operator or the relative expression. For example, $/*//n_i$ or $*/n_i$ is encode as $(a_{n_i}, \geq, 2)$.

- Relative predicate $(d(a_{n_i}, a_{n_j}), o, v)$: represents the constraints on the relative position of two tags n_i and n_j. This predicate is used to encode the two tag names in consecutive location steps. The operator o can be either $\geq$ or $=$. The $=$ is used to encode the parent/child relationship between the two tag names; and the $\geq$ is used to encode the ancestor/descendant relationship. For example, the expression $n_i/*/n_j$ is represented as $(d(a_{n_i}, a_{n_j}), =, 2)$ and the expression $n_i//*/n_j$ is represented as $(d(a_{n_i}, a_{n_j}), \geq, 2)$.

- End-of-path predicate $(a_{n_i}^{-1}, \geq, v)$: specifies the position of a tag name n_i relative to the end of an XPath expression. This predicate is used to encode the tag names that are followed by wildcards only. For example, for the tag name n_i in the expression $/*/n_i/*/*$, the predicate $(a_{n_i}^{-1}, \geq, 2)$ is used to specify that there should have at least two tags following n_i in a matching XML path with this expression.

- Length predicate $(length, \geq, v)$: specifies the constraints on the length of the

XPath expression. This predicate is used to encode the path with wildcards only. For example, the XPath expression $/*/*/*$ is represented using the predicate $(length, \geq, 3)$.

An XPath expression is encoded using a combination of the above predicates in terms of the type of the XPath expression. In Predicate-based Filter approach, all XPath expressions are translated to a set of predicates. A multiple stages of hashtable is used to index all distinct predicates in the set of predicates. An XML document is a collection of paths, denoted as $(e_1, e_2, \ldots, e_n)$, from the root of the document to each leaf element. The path is then translated to a set of tuples as follows :

$$\begin{gathered}
(length, n), \\
(e_1, 1), (e_2, 2), \ldots, (e_n, n), \\
(e_1, e_2, 1), (e_1, e_3, 2), \ldots, (e_1, e_n, n-1), \\
(e_2, e_3, 1), (e_2, e_4, 2), \ldots, (e_2, e_n, n-2), \\
\ldots \\
(e_{n-1}, e_n, 1)
\end{gathered}$$

The matching algorithm consists of two phrases. Firstly, the above tuples from XML document paths are used to retrieve the matched predicates from the hashtable. Secondly, the matched predicates are combined to check the matching of the complete XPath expressions.

Shared processing of both common prefixes and suffixes. AFilter [36] leverages the shared processing in both common prefixes and suffixes. An *AxisView* is created to capture the information of all nodes and all axes in the collection of XPath expressions. Specifically, the *AxisView* (denoted A_{view}) is a directed graph such that : (1) for an element name e_i in some XPath expression, there exists

a unique node with name e_i in A_{view}; (2) if there exists an axis e_i/e_j or $e_i//e_j$ in some XPath expression, then A_{view} has an edge from node n_i to n_j; (3) if the XPath expression with identity k has the axis e_i/e_j (resp. $e_i//e_j$) at the rth step, then the edge from n_i to n_j is annotated with an assertion $(q_k, r)|$ (resp. $(q_k, r)||$). Each node in A_{view} is associated with a stack. During the parsing of the XML document, these stacks maintain the currently active data nodes. The data nodes in stacks have the pointers to the data nodes in other stacks to maintain the ancestor/descendant and parent/child relationships for them. When the data node that matches a leaf node in some XPath expression is encountered, a backward checking procedure is triggered to determine whether the XPath expression is matched.

A *trie* structure is used to index the XPath expressions based on the commonalities in their prefixes. The backward checking procedure would cache the matching results for some expressions. Thus, next time when the same checking from the same pointer needs to be validated, the results in the cached can be returned immediately. Similarly, a *trie* structure is also used to cluster the XPath expressions based on their overlapping suffixes. The matchings of XPath expressions are triggered by the matchings of some leaf nodes. To cluster the common suffixes can reduce the number of assertions to consider. It shows that by combining the shared processing on both common prefixes and suffixes, AFilter achieves better performance than YFilter.

Shared processing of common predicates. The above approaches only intend to share the processing of the common parts in the *axis navigation*. However, for the XPath expressions with value predicates, it is not uncommon to have some common parts in the predicate parts. For example considering the two XPath queries $/b/a[@c = 2]$ and $/e//a[@c = 2]$, the fact that the predicate $[@c = 2]$ is common in both queries can be exploited to speed up the filtering further. The

XPush machine approach [63] can eliminate the redundant processing caused by such common predicates.

XPush machine uses a *deterministic pushdown automata* to index the queries. Besides the element names, *XPush machine* also converts the predicates to be the states of the automata. The XPush Machine uses a modified pushdown automata in which the states in XPush Machine have two components: a top-down state and a bottom-up state. The top-down phrase builds the stack for the automata, and the bottom-up phrase is the matching of the automata with the document. The bottom-up state in the top of the stack stands for the total subquery that has been evaluated to be true so far. The states transition is driven by the events reported from the SAX parser. When an XML document is exhausted, the current state will return a set of XPath expression identities that are matched with the document.

To construct an XPush machine for a collection of XPath expressions, the algorithm first converts each of the XPath expression to an *Alternating Finite Automaton* (AFA). The construction of AFA is like the construction of NFA in YFilter, and it only needs to add an AND, OR, or NOT label to a state after constructing the NFA. The construction of the bottom-up XPush machine follows the construction of AFA. Each bottom-up state in XPush Machine stands for one common part (including both the navigation part and the predicate part) in every AFA. Therefore, the XPush machine can eliminate the redundant work in both structural navigation part and predicate part.

When queries have multiple predicates, the query evaluation time may be dominated by the predicate evaluation. To eliminate redundant processing in predicate evaluation further improves the matching efficiency. In this case, XPush Machine is more efficient. However since XPush machine incorporates the predicate information in the states of automata, it may need more states than other approaches,

especially when the number of predicates in the XPath expression is large. The construction of XPush Machine may also take more time.

Shared processing of XML documents. Different XML documents conforming to the same DTD may also contain similar parts in both structure and contents. The RoXSum approach [112] designs a new structure (i.e. RoXSum) to aggregate the content of multiple XML messages. The query matching is performed on the RoXSum itself instead of original documents. This approach improves the efficiency by batched processing of XML documents. Similarly, B-BoXFilter [83] also explores the batched processing mechanism.

3.1.2 Approaches to Reduce the Number of Queries

The filtering time increases as the number of XPath expressions increases. This section introduces the approaches aiming to improve the filtering efficiency by reducing the number of subscriptions to be processed.

The methods in [38, 117] use query aggregations to reduce the number of subscriptions. One disadvantage of these approaches is that the aggregation may cause the irrelevant information to be forwarded to some users.

The problem of subscription aggregation is defined as follows. Given a set of tree pattern subscriptions S and a space constraint k on the total size of aggregated subscriptions, the aggregation computes a set of subscriptions S' which satisfies the following conditions:

- All the documents that are matched with S are also matched with S'.
- The number of nodes in S' is not greater than the constraints k.
- The error of the matching results between the two sets is minimized.

To solve the above subscription aggregation, an algorithm to compute the aggregation of two XPath expressions is required, which is addressed in [38]. Given two tree patterns p and q, the algorithm iteratively traverses the subtree of p and q. "$*$" is the aggregation of the nodes at the same level with different name, and "//" is the aggregation for the root nodes of the common sub-pattern at different level of p and q.

Then the aggregation of a set of XPath expressions S to another set of XPath expressions S' can be handled. The approach in [38] iteratively selects the pair of XPath expressions to aggregate such that the aggregation maximizes the gain in space while minimizes the loss in selectivity. The approaches in [117] defines a similarity function to measure the similarity between two tree pattern queries and the clustering mechanism is adopted to cluster and aggregation the similar queries. Chand et al. [42] also proposed an approach to estimate the tree-patten similarity based on the observed document stream. The effective clustering of similar queries helps to increase the accuracy of the aggregation.

The methods in [61, 62] uses the attached additional information (i.e. *precomputed-view*) with the published documents to detect matching subscriptions before evaluations. This approach assumes that the publishers have some knowledge about the queries issued by the subscribers such that the publishers can pre-compute some information to benefit the matching on each router. Such information is attached with the document as the *header* to be forwarded together. Specifically, the views include a set of XPath expressions and their matching results. When a router receives an XML document with the header, it first tries to answer the XPath expressions using the header. For the XPath expressions that match the header, their evaluation can be skipped during the parsing of the document. Furthermore if all XPath expressions are matched, the parsing of the XML document can be skipped.

To attached views in this approach is similar with the work to piggyback additional information proposed in this thesis. However, the views in this approach are pre-computed by the publishers; while the piggybacked information in this thesis is dynamically computed by the upstream router based on its matching results.

3.1.3 Approaches to Reduce the Matching Complexity

The tree pattern matching performed between the XPath expressions and XML documents dominates the filtering time. This category of approaches try to convert the time-consuming tree pattern matching on each router to other simple matchings.

The BloomFilter [59] approach uses the Bloom filter [26] to convert the tree pattern matching to the matchings of bit-vectors, which can be performed more efficient. A Bloom filter is a bit-vector of length m used to efficiently test whether an element y is a member of a set $S = \{x_1, x_2, \ldots, x_n\}$. To store an element x_i into S, a set of hash function $h_1(x_i)$, $h_2(x_i)$, ..., $h_k(x_i)$ are computed such that the bits at the positions corresponding to the computed values are set to be 1. To check whether an element $y \in S$, the bits at positions $h_1(y)$, $h_2(y)$, ..., $h_k(y)$ are checked. If any position has the value 0, y is guaranteed to not belong to S; otherwise y is considered to be in S. However, the *false positive* does exists although the probability is relative low for many applications.

The BloomFilter approach uses one Bloom filter to store all queries for each user. Given a set of users $U = \{u_1, u_2, \ldots, u_n\}$, their corresponding Bloom filters are denoted as $B = \{b_1, b_2, \ldots, b_n\}$. During the parsing of the XML document, let n denote the currently parsed element, then all possible paths (including the paths with wildcard "*" and axis "//") are enumerated. For a certain path p enumerated, if there exists a bloom filter b_i such that $p \in b_i$, then the document is considered

to be matched u_i's interest and will be forwarded to u_i. The efficient matching of bit-vectors leads to the better performance of the BloomFilter approach. However, as the depth of the XML document increases, the number of enumerated paths increases exponentially which may diminish the improvement by the BloomFilter and the BloomFilter may cause the false positive such that some irrelevant information may be disseminated to the users.

FiST [71] converts the tree pattern matching to be the sequence matching. The XML documents and XPath expressions are encoded using the Prüfer sequences. Then the matching of an XPath expression with an XML document can be determined by the matching of prüfer sequence for the query with the the prüfer sequence for the XML document. The set of XPath expressions on each router will be converted to a collection of sequences, and these sequences are organized using a dynamic hash based index. The matching is conducted in two phrases. Firstly, a progressive subsequence matching procedure is used to obtain a set of XPath expressions whose prüfer sequences match some prüfer sequences generated during the parsing of the XML document. This phrase identifies a superset of XPath expressions that potentially match the incoming XML document. The second phrase is conducted to refine the results to eliminate the *false positive* matchings. By converting the tree pattern matching to a holistic sequence matching, FiST avoids the join operations for tree pattern queries in other approaches, which makes FiST better for the XPath expressions with more branch nodes.

BoXFilter [83] also makes use of the Prüfer sequences for matching XPath expressions. A early pruning mechanism is used to eliminate unnecessary XPath expressions as early as possible.

The *Branch Sequencing* approach [94] also converts the complicated tree pattern matching to the sequence matching, while the *branch sequencing* is proposed in

the paper and used to encode both XML documents and XPath queries. Besides determining whether an XPath query is matched, the *branch sequencing* approach can also output the matched nodes for the query.

The SemCast [87] approach makes use of the multiple channels in the interior dissemination network such that the time-consuming content-based filtering at the interior routers are eliminated. In the SemCast, a number of multicast channels are created for disseminating information. Each channel consists of several routers which form a dissemination tree. The channel is represented by some predicates expressions to specify the content. Data sources publish the information to one or more channels in terms of the overlapping between the content of the data source and the content of the channel. Data consumers subscribe to one or more channels whose content collectively cover their interests.

There are five kinds of routers in SemCast system :

- *Coordinator :* is responsible for creating and managing channels. It communicates with both the source routers and gateway routers to inform them about the content of the existing channels.

- *Source Routers :* receive the information from the publishers. By comparing the content of the information with the content of channels from the coordinator, the source routers determine the set of dissemination trees for forwarding the information.

- *Rendezvous routers :* take charge of the dissemination trees and serve as the roots of the dissemination trees.

- *Internal Routers :* forwards the information by identifying the identity of the channels that correspond to this information.

- *Gateway Routers :* take charge of forwarding the information to the end users.

The SemCast improves the filtering efficiency by eliminating the content-based matching at internal routers. However, SemCast may incur a large number of channels to be maintained.

3.2 Extending the Functionalities of Dissemination Systems

The basic content-based dissemination system only supports the filtering of the information. Once the information matches the subscription from some user, the complete piece of information will be forwarded to the user. Some approaches are proposed to provide more functionalities of the dissemination system.

To detect all matchings for queries. Besides the information that matches the user's interests, one user may also want to know all the matching positions for the subscribed query such that a quick browsing for the interested parts can be performed. MatchMaker [72] and Index-Filter [33] are proposed to find all matches for subscriptions during the filtering of documents.

The MatchMaker indexes all the parent/child and ancestor/descendant relationships in the XPath expressions. For each event reported from the SAX parser, the MatchMaker uses the index to find the relevant queries to process. The matching status for each query is recorded using some auxiliary lists. Once a matching for a query is detected, the matching position is recorded. Thus when the document is completely parsed, all matchings for queries are detected.

The *Index-Filter* provides the path queries with the position information of the

elements in XML documents. The position of an element in the XML document is represented by $(L : R, D)$ where L and R are the offset of the start and the end of the element from the beginning of the document by counting the numbers of the words and D is the level of the element in the document tree. *Index-Filter* algorithm is conducted in two phases. The first phase is to find the candidate nodes that may participate in a new match using the position information of the elements that associated with each node in the prefix tree of all the queries. The representation of the position information makes it easy to determine the structural relationship between two nodes. For example, given two nodes n_1, n_2 with the position information as $(L_1 : R_1, D_1)$ and $(L_2 : R_2, D_2)$, if $L_1 < L_2$ and $R_1 > R_2$, we can conclude that n_2 is the descendant of n_1. Further if $D_2 = D_1 + 1$, we can conclude that n_2 is the child of n_1. The second phase of the algorithm decides whether it is time to output the matching results and output the results once the matching is finished. In the dissemination scenario, the position information of elements is computed on-the-fly by parsing the document once. If the document is large and the number of queries is small, the time to compute the position information will delay the response time.

To customize the information with respect to specific users. ONYX [50] extends the conventional content-based dissemination by supporting the information transformation such that the information disseminated to users is customized in terms of the users' requirement. In ONYX, queries are written using a richer subset of XQuery [13], which is the set of *for-where-return* expressions. The *for* clause binds the elements that match a path expression with a variable name; the *where* clause further filters the binding elements using a set of conjunctive predicates; and the *return* clause retrieves the fragments of XML document requested by users. Given an incoming XML document, the ONYX system returns the users

the transformed XML document which is a set of tuples retrieved by the *return* clause.

To handle a large set of queries, the shared processing for common parts in the queries helps to improve the efficiency significantly. In ONYX, the shared processing is exploited to handle the *for*-clause and the *where*-clause which can make use of the shared matching engine in YFilter. Furthermore, the shared processing is also exploited to handle the *return*-clause in some degree. Finally, a post-processing procedure is performed to customize the information with respect to specific users.

To support stateful subscriptions. The traditional content-based dissemination system only allows the subscriptions to address an individual event, which is called *stateless subscriptions*. However, some users may be interested in the trends of a sequence of events happened in a period of time. For example, in the stock ticker system, some user may have the request for a stock whose price consistently increases in an hour. There are some systems proposed to support such kind of request, which is constructed using the *stateful subscriptions*.

A state-persistent publish/subscribe system is proposed in [74]. The system stores the states between the publications and subscriptions, which are the relationships about whether a publication matches some subscriptions or whether a subscription is matched by some publications. Then a publication is only sent to a user when some subscription from this user undergoes a state transition with respect to this publication.

PADRES [77] is another system to support the stateful subscriptions, in which the *composite subscriptions* are proposed to address a sequence of events. The *composite subscription* is formed by linking the traditional subscriptions using some logical or temporal operators. Each *composite subscriptions* is matched by a set of events that satisfy its correlation requirements. The PADRES supports four kinds

of composition operators on subscriptions, i.e. *parallelization*,*alternation*,*sequence* and *repetition*. The *parallelization* operator is to address the set of events that occur together; the *alternation* operator on two subscriptions s_1 and s_2 is matched if an event matches either s_1 or s_2; the *sequence* operator is to address a set of events that occur in a sequence, and the attribute *time-span* can be used to specify the time interval between two events; and the *repetition* operator matches the events that occur in aperiodically or periodically. The PADRES proposed a mechanism to decompose the composite subscriptions and allocate the parts of subscriptions in the distributed environment. A rule-based matching engine is extended to support the matching of composite subscriptions.

The Cayuga [47] system also allows users to express subscriptions that span a set of events. Four kinds of binary operators, addressing on subscriptions s_1 and s_2, are proposed to support the stateful subscriptions. The first operator is *union*, which is similar with *alternation* operator in PADRES. The second operator is called conditional sequence, which requires that s_1 and s_2 are the sequence of two consecutive and non-overlapping events, and s_2 satisfies the condition with s_1. The third operator is *iteration* operator, which is the repeated application of the conditional sequence operator. The *iteration* iteratively applies the *conditional sequence* operator on a sequence of events. This operator enables the powerful parameterized subscriptions. The last operator is *aggregate*, which occurs over a sequence of events. For example, to compute the summarization or average for some attribute over a sequence of events needs the *aggregate* operator.

The queries supported in Cayuga are only simple attribute-value predicates. In [67], Hong et al. further proposed an approach to process a large number of XML queries involving joins over multiple XML documents.

To provide the QoS property for dissemination. Many existing dissemination systems are based on the best effort principle. Subscribers are only allowed to issue their interests for the kind of XML documents, but no parameters addressing the quality of the dissemination service are provided to them. The router sends matched XML documents to subscribers as soon as the processing of XML documents has been finished. Schmidt et al [99] extended such kind of dissemination systems through implementing a prototype of a Quality-of-Service-based dissemination system, which is based on a state-of-the-art real-time operating system that provides the native streaming support with QoS. The system gathers the QoS parameters from the DTD [12], the document and the XPath queries to estimate the resource to be consumed and determine whether the queries can be accepted. If the queries can be accepted, the operating system reserves the resource for the queries; otherwise, the queries are rejected.

3.3 Query Processing Using Annotations

This section introduces some related work with the annotation mechanism used in global optimization of XML dissemination. The annotation mechanism is exploited in some other works to either improve the query processing efficiency or to reduce the memory usage in the streaming processing.

Tucker et al. [110] proposed to use the annotations in processing continuous queries in streaming data. Due to the infinite property of the continuous stream data, the query processing encounters the *blocking operators* and *unbounded stateful operators* problems. The annotations are leveraged to specify the end of a subset of the attribute tuples in the stream with the format $< attribute_1, attribute_2, \ldots, attribute_n >$, which indicates that no more attributes following such annotation

will match it. Then the operators that can not evaluated due to these attributes can be processed. Ding and Rundensteiner [52] also made use of the annotations in the continuous data stream to evaluate window join queries. These works are different with the work in this thesis to optimize the dissemination with piggybacked annotations in two aspects : (1) the above works concentrated on the streaming data in the attribute-value format; while this thesis focused on the XML data; (2) the annotations in the above works are inserted before transmitting the data; while the annotations used in this thesis are inserted dynamically during the transmission of the data.

Shen and Tirthapura [101] introduced the Lookup Reuse approach in the content-based dissemination. Their approach based on the observation that the neighboring routers are likely to have common subscriptions such that an event that matches some subscription on a router is likely to match some subscription on its neighboring routers. The Lookup Reuse approach allows the upstream router to insert the identities of a list of matching subscriptions as the annotations, and these annotations are forwarded to the downstream router together with the data. Then the downstream router would perform a hash lookup to detect the matching subscriptions using the annotations which avoids the processing on the content of the data. The Lookup Reuse approach utilizes a similar mechanism to exploit the collaboration among routers. However, firstly this approach does not consider the subscription aggregation during the propagation of subscriptions. With the aggregation, the subscriptions on the neighboring routers may not be exactly the same such that a simple lookup may not work well. Secondly, only the correlation among subscriptions are exploited in the Lookup Reuse approach; while the work in this thesis additionally considered the correlation between the XML data and subscriptions.

3.4 Query Processing on Fragmented XML Data

In this section, we review some approaches that are related with the query processing on fragmented XML data, while they are not applied in the dissemination environment.

Bose and Fegaras proposed several approaches to process queries on fragmented streaming XML data based on the *hole-filler* model [95, 97]. In the *hole-filler* model, the XML stream data is transmitted in the unit of fragments, each of which is associated with an unique ID. The fragments are related using the concepts of *holes* and *fillers*. Each fragment is considered as a filler which contains holes in it, and each hole specifies the ID of the filler that can be positioned to complete the tree. The early works by Bose and Fegaras [55] processed the fragments based on the navigation among fragments. Then the processing of some fragments has to be suspended due to the lacking of their ancestor fragments. This method not only incurs challenge requirements on the memory, but also delays the query response time due to the waiting for a fragment that is necessary for the information completeness to be processed.

With respect to such problem, another approach, called XFrag [96], was proposed. The XFrag makes use of the structural summary of the XML data, which indicates the structure of tags in the document and provides the information of fragmentation. The structural summary is used to generate the evaluation plan for an XQuery expression, and during the processing time, it can be used to decide whether a fragment is needed to be kept in memory. In XFrag, the fragment is processed upon arriving, and the fragments that are guaranteed not to contribute to the results would be discarded as soon as possible. Thus, the XFrag is optimized for memory usage. The goal of XFrag is to evaluate a single continuous query on the streaming XML data to output all the selection results, which is different with

our work that is to match a batch of boolean XPath expression in the dissemination environment. Therefore, our work utilizes a different evaluation and optimization mechanism. Furthermore, our work exploits scheduling strategies of fragments to improve filtering efficiency.

The Active XML [18] is a framework for managing the distributed information over the internet using a peer-to-peer architecture. The information is stored as the Active XML document in the system. In the Active XML document, some parts of the data are stored explicitly, while other parts contain the Web Service calls to retrieve the fragment of XML document when required. To evaluate a query on the Active XML document, the processor navigates the document to match the query. When a Web service called is encountered, the called is activated to fetch the corresponding data, and the returned data is inserted into the document for the query evaluation. The returned fragment of XML data may also contain the Web service calls, and they may be activated for the further processing if necessary. To materialize all the service calls is not practical and necessary. Thus the peer hosting the Active XML document decides when to activate a particular service call. In [19], Abiteboul et al. proposed an approach to activate the Web service call lazily, and the irrelevant Web service calls would not be invoked.

In Active XML, the XML fragments are stored distributed and statically on the peers. The XML fragments would be sent to other peers by the service calls from other peers. This is different with our work on disseminating the fragmented XML data, in which environment the fragmented XML data is transmitted as a stream in the publish/subscribe architecture. In the Active XML, the query processing is initialized on one peer, and the relevant service alls are activated sequentially when navigating in the document; while in our fragmentation work, each fragment is identical and can be processed in any sequence.

Some researchers have studied the distributed query processing on fragmented XML data. Dan Suciu [105] addressed the problem of query evaluation on distributed semistructured database. In his setting, fragments of databases are stored at a fix number of sites, and the cross links between two fragments are used to connect the related information at different sites. The goal of the work is to minimize the communication among sites during the query evaluation. Deshpande et al. [48] studied the query processing of XML data on a distributed sensor network, where the fragments of XML data are collected by sensors. The data collected by a sensor are stored close to this sensor, but can be cached elsewhere in the network as required by some queries. Deshpande et al proposed the approach to efficiently route the queries to the proper site for the evaluation and to efficiently collect the missing data for processing the queries.

The above two works handled the query processing on the scenario that the fragments of XML data are statically stored on the distributed sites. Either the queries are routed to the proper sites or the missing data are fetched from other sites to evaluate the queries. They tried to minimize the communication among sites for query processing. Thus the query evaluation in their work is optimized at different sites, which is different with our fragmentation work that the query evaluation is optimized at one site, since in our scenario, the fragments of XML documents are transmitted as streaming in the dissemination network and all fragments of an XML document would arrive on the same router for matching the subscriptions.

Recently, a new technique for decomposing an XML document into a set of vectors was proposed by Buneman et al. in [34]. Each fragment is identified by a path and it contains all leaf text nodes for that path. The query processing makes use of a compressed skeleton that describes the structure of the XML data, with the necessary scanning of data vectors and decompression of skeletons. Their goal

is to query a large XML repositories statically stored, which is different from the dissemination environment that small documents are published and transmitted.

To disseminate and process the XML fragments, we need an approach to partition the XML document. A batch of such approaches are proposed. Gertz and Bremer [58, 32] developed a fragment specification to decompose an XML document into a disjoint and complete set of tree-structured data fragments. The fragmentation can be classified into two types: vertical fragmentation and horizontal fragmentation, in which vertical fragmentation is based on the structure of data and horizontal fragmentation is based on the attribute values. Bordawekar and Shmueli [28] incorporated the workload during the partition of XML documents. They assigned the weight, which measures whether two nodes should be put in the same cluster, to each edge of XML tree and partitioned the tree based on the weight such that intra-cluster weight is maximized and inter-cluster weight is minimized. Wong et al. [113] introduced the concept of the semantic-based fragmentation of the XML documents, where the structure and organization of the contents are taken into consideration. W3C provided a recommendation called XML Fragment Interchange [14], which considers the XML document as a logical document composed of possibly several entities. The above works examined how to fragment XML documents, which are the complementary of our work on disseminating the fragmented work that focused on matching queries directly on the fragmented data.

3.5 Query Processing on Heterogeneous Data

In this section, we discuss some related works for querying heterogeneous data in both dissemination and non-dissemination environment.

Petrovic et al. [89] proposed a semantic publish/subscribe system, i.e. S-ToPSS,

to deal with the semantic filtering problem. The matching algorithm was extended by adding three kinds of semantic capabilities. Firstly, the synonyms are considered to match the events and subscriptions that use semantically equivalent attributes; secondly, a concept hierarchy is exploited to match the events and subscriptions that have specialization or generalization relationships; thirdly, the mapping functions are allowed to define arbitrary relationships between schema and attribute values, and these functions are exploited to match the events and subscriptions. The S-ToPSS system handles the heterogeneous data which are published as the attribute-value tuples. The XML data also incur the structural heterogeneity which is not considered in S-ToPSS.

Uschold et al. [111] developed Xinfosphere system that makes use of the ontology to provide the semantic matching in the dissemination. The ontology is created using the DAML+OIL [68] syntax to represent the key concepts and relationships in the domain of interests. The published data is associated with the semantic annotations to represent the content of the data using ontology language. The subscriptions in Xinfosphere are also represented using DAML+OIL to address a set of interested objects. The building of ontology involves more effort and the query matching using ontology is inefficient. In Xinfosphere, both the publishers and subscribers should be aware the semantic matching, and perform the corresponding operations. However, the work in this thesis keeps the dissemination of the heterogeneous XML data transparent to both the data publishers and data consumers.

The problem of querying heterogeneous data also occurs in the data integration scenario, and the approaches based on query rewritten are proposed to address this problem in data integration. A global schema exists for the heterogeneous data. Queries are expressed in terms of the global schema, and will be reformulated to

be a set of queries over the various data sources using the proper approach based on how the correspondence between the data sources and the global schema is specified. The query rewritten approaches are classified into three categories :

- *local-as-view*(LAV) [76, 80] : In LAV, the local data sources are represented as a set of views over the global schema. The advantage of LAV is that new data sources can be added without affecting the global schema, which makes it more scalable. However, the query rewritten is known as a hard problem [75], since there is no mediator to align the users' query with a simple expansion strategy as in GAV.

- *global-as-view*(GAV) [35, 79, 115, 116]: In GAV, the global schema is represented as a set of views over the local source schemas. Then the query rewritten in GAV is straightforward since the associations between the global schema and the local schemas are well-defined. However, the global schema has to be modified once some new data source is added or some local schema is modified. Thus, GAV prefers the scenario that the data sources are relatively static.

- *global-and-local-as-view*(GLAV) [57] : it combines the data expressive power of both LAV and GAV. It is shown that the data complexity of answering queries in GLAV is no harder than in LAV.

These query rewritten approaches to query heterogeneous data are suitable for the environment that a single query is evaluated on huge amount of data that is stored on the disk. However, the work in this thesis dealt with the dissemination of heterogeneous data, where a small document is parsed to match a large number of queries simultaneously, which makes the query rewritten approach not scalable.

There also exist some works [98, 22, 81, 21] to solve the querying of structural

heterogeneous data problem by relaxing one query to a set of queries for evaluation and generating a ranked matching result. Three operations are used to relax the queries, i.e. edge generalization, leaf deletion and subtree promotion. The relaxed queries are combined into a DAG, and would be evaluated together. These approaches have to relax one query to several candidate queries for evaluation, which is not suitable for the dissemination environment where the number of queries is usual large. The reason is that to relax these queries will significantly increase the number of processed queries, which further reduces the filtering efficiency.

3.6 Summary

The chapter first surveys a batch of works that are related in dissemination environment. After that some related works with each particular problem in this thesis are studied. The works in the dissemination scenario is classified into two categories. Section 3.1 introduced the set of approaches that intend to improve the filtering efficiency. It is observed that all existing works in content-based dissemination of XML data try to improve the efficiency on each individual routers, which is called *local optimization*. This is different with the work in this thesis that optimize the filtering efficiency by exploiting the collaboration among routers, which can be considered as the *global optimization*. Section 3.2 studied the approaches that focus on extending the functionality of the dissemination system. It can be observed that none of the existing works in content-based dissemination of XML data addressing the problems to handle the data that are fragmented or heterogeneity in both the content and the structure. These two problems are handled in this thesis.

Chapter 4

Global Optimization for XML Data Dissemination

4.1 Introduction

To publish data content in XML format and to use XPath as subscription language provides a more expressive content-based dissemination mechanism. However, it also increases the complexity of subscription matching at routers. Thus, there is an even greater need for effective optimization techniques to meet the performance challenge of content-based dissemination of XML data. The existing research efforts have focused on two key optimizations to minimize the number of subscription matchings.

1. The first optimization is to exploit efficient index structures (e.g.,[39, 49, 20, 33, 59, 60, 63, 69, 71, 117]) to perform selective matching with only a small subset of potentially matching subscriptions.

2. The second optimization uses aggregation algorithms to summarize an initial set of subscriptions into a smaller set of generalized subscriptions (based on

subscription containment properties) to reduce the number of subscriptions and the matching overhead [38, 117].

One limitation of the existing approaches is that they only consider the opportunity to improve the performance on each individual router. However, as mentioned in Section 1.2.1, in the content-based XML data dissemination system, the same document D is being repeatedly processed against related sets of subscriptions on the upstream and downstream routers. This opens up a new opportunity to improve the filtering efficiency by exploring the collaboration among various routers. To facilitate such "collaborative" processing, the upstream router R_i is allowed to pass along some hint information (which is referred to as annotations) to its downstream router R_j when it forwards D to R_j. Such hint information is generated based on the matching of document D with the set of subscriptions on R_i. On receiving the document D with annotations, the downstream router R_j tries to exploit the hint information to optimize its own processing of D. There are two key ways that a downstream router R_j can optimize its processing and matching of D by exploiting additional hint information from its upstream router R_i :

- The hint could enable R_j to quickly determine that D is to be forwarded to a downstream router R_k without requiring R_j to parse and process D.
- The hint could enable R_j to quickly detect that a portion $S'_j \subseteq S_j$ of the subscriptions in R_j's routing table are guaranteed not to match D, and R_j can therefore speed up its matching of D against the smaller set $(S_j - S'_j)$ instead of S_j.

Let us use the following two examples to illustrate the above optimization opportunities.

Example 4.1 Consider the following routing of a document D among three routers (R_i, R_j, and R_k), where D is first forwarded from R_i to R_j due to a matching subscription $/a//d \in S_i$ (that is associated with R_j); and then D is then forwarded from R_j to R_k due to a matching subscription $/a/b/c/d \in S_j$ (that is associated with R_k). Observe that if R_i had forwarded to R_j (along with D) the additional information on the data bindings for the matching subscription $/a//d \in S_i$; i.e., that the wildcard "//" in $/a//d$ actually matches the data path "b/c" in D, then R_j could have very efficiently determined that D matches the subscription $/a/b/c/d \in S_j$ without actually having to parse and process D against the subscriptions in S_j. In this way, R_j is able to speed up the forwarding of D to R_k and thereby reduce the overall processing time to disseminate D to relevant subscribers. □

Example 4.2 Consider the scenario where a router R_i needs to forward a document D to its downstream router R_j, and that after having parsed and processed D against S_i, R_i has obtained the following information about D and S_i: (H1) D does not match some subscription $s \in S_i$ that is associated with R_j; (H2) the data pattern "x/y/z" occurs in D with its last occurrence located at some position p within D; and (H3) the data pattern "a/b/c" does not occur at all in D. Observe that each of these three pieces of information could be forwarded to R_j as hints to optimize the performance of R_j. For (H1), R_j can use the non-matching subscription $s \in S_i$ to identify the subset of subscriptions $S'_j \subseteq S_j$ in R_j that were aggregated to s (i.e., subscriptions that are guaranteed to not match D), and exclude matching D against such subscriptions to improve the matching performance. For (H2), once R_j has parsed D beyond position p, R_j can conclude that there will not be any new matches of subscriptions that contain the data pattern $x/y/z$, and therefore such subscriptions can be excluded from further matching and processing. Finally, (H3) can be treated as a special case of (H2) with p being at the starting position

of the document D. Thus, R_j can ignore matching D against the subscriptions in S_j that contain the data pattern $a/b/c/$ right at the beginning of D. □

This chapter presents our proposed *piggyback optimization* approach, which is an orthogonal and holistic optimization that enables a downstream router to leverage the subscription matching work completed by upstream routers to optimize its own performance. This optimization could be used in combination with the existing optimizations proposed for pub/sub systems (i.e., subscription indexing and aggregation).

There are three key design issues to be addressed for our piggyback optimization approach:

1. What type of information is useful to piggyback?
2. How can such information be efficiently computed by a forwarding router and exploited by a receiving router?
3. How does this optimization impact the data matching protocol (i.e., when a router R_i detects that some subscription corresponding to a downstream router R_j matches a document D, should R_i forward D immediately to R_j? And should R_i continue matching D against other subscriptions related to R_j?)

As there are many possible types of hints that could be forwarded along with a document, forwarding too much hints could increase both the transmission cost as well as the overhead of pre-processing the hints and thereby possibly negating the potential performance improvements. Thus, the hints need to be selected judiciously to balance these tradeoffs. Intuitively, a hint is preferred if it is more likely to be beneficial and can be efficiently computed by the upstream router and exploited by the downstream router.

In this chapter, we examine and evaluate several design options for piggyback optimization. Our experimental study demonstrates that our proposed piggyback optimization is indeed a feasible and effective technique to improve the performance of content-based dissemination of XML data.

The rest of this chapter is organized as follows. Section 4.2 gives an overview of our piggyback optimization technique. Section 4.3 presents each type of annotation in detail. We discuss the mechanism to generate the annotations in Section 4.4 and the approach to process the annotated document in Section 4.5. Section 4.6 presents experimental results, and Section 4.7 concludes this chapter.

4.2 Overview of Piggyback Optimization

This section presents an overview of the novel approach, which is termed *piggyback optimization*, to optimize the subscription matching by exploring the collaboration among routers. The proposed technique is orthogonal to the two existing optimizations, namely, indexing techniques and subscription aggregation algorithms, that are also targeted at improving the routers' performance.

The central idea behind piggyback optimization is to optimize the performance of a router by leveraging information from the work done by its upstream router. This is possible because both the upstream and downstream routers are processing the same document against subscriptions that are partially related (due to subscription containment relationships). Thus, an upstream router could pass to its downstream router some useful hints (along with the document being forwarded) about properties of the document and/or matching/non-matching subscriptions that it has encountered to enable the downstream router to optimize its performance by expediting the forwarding of the document (without processing the doc-

ument) and/or speeding up its subscription matching process.

In our proposed piggyback optimization, the hints from an upstream router are disseminated to its downstream router in the form of header annotations in the document. On receiving an annotated document, a router will first pre-process the header annotations to optimize the subsequent processing of the document.

In the following, we use $A_{i,j}$ to denote the header annotations that an upstream router R_i adds to a document D before forwarding it to a downstream router R_j. The annotated document that R_j receives from R_i is denoted by $(\mathrm{D}, A_{i,j})$.

4.3 Types of Annotations

The first key issue for the piggyback optimization technique is to decide on what types of information to include in the header annotation of a document to optimize performance. Our design of the annotated information is guided by three performance-related requirements.

1. It should be concise so that it incurs minimal processing overhead in terms of parsing and transmitting the additional header information.

2. It should be efficiently generated so that the computation overhead incurred by the upstream router does not offset any performance gains of its downstream routers.

3. It should be effective in that a downstream router can efficiently preprocess the annotations to optimize its subscription matching performance.

Let us consider the possible sources of useful information that an upstream router R_i can pass on to a downstream router R_j along with a document D that needs to be forwarded to R_j.

	Subscription	**Data**
Positive	Data bindings for matching subscriptions (PS)	Positions of last occurrences of data patterns (PD)
Negative	Non-matching subscriptions (NS)	Non-occurring data patterns (ND)

Figure 4.1: Types of Annotations

After having matched its own subscriptions against D, R_i has acquired additional information about D and how its subscriptions are related to D. We can classify this knowledge into *positive* and *negative* information:

- *Positive information* refers to information about (a) subscriptions in T_i that matched D, and (b) patterns / properties that occur in D.
- *Negative information* refers to information about (a) subscriptions in T_i that did not match D, and (b) patterns/properties that did not occur in D.

We shall refer to the annotations of these two types of information as *positive annotations* and *negative annotations*.

In the following, we identify two types of positive annotations (PS and PD) and two types of negative annotations (NS and ND), which are classified in Figure 4.1. The emphasis of the discussion in this section is on the ideas; we address the implementation issues in Section 4.4.

4.3.1 Positive Annotations

A positive annotation specifies information related to either (1) a matching subscription or (2) a data pattern that occurs in the document. Subscription-related information could be used to expedite a document forwarding decision without having to process the document itself, while data-related information could be used to reduce the effective number of subscriptions that need to be matched.

Positive subscription (PS). To understand the positive subscription annotation, we need to first define the *simple aggregated subscription* as follows.

Definition 4.3.1 (Simple aggregated subscription) *A subscription s' is said to be a* simple aggregated subscription *of another subscription s (or s is a* simple aggregating subscription *of s') if the following two requirements are satisfied :*

- *$s \sqsubseteq s'$*

- *s' can be made to match s by simply substituting each wildcard (i.e., $*$ and //) in s' with some path of data element names*

A PS annotation is of the form (s_x, B_x), where s_x is some subscription detected by R_i to match D (i.e., $s_x \in S_j$, $(S_j, R_j) \in T_i$), such that s_x is a simple aggregated subscription of some subscription in T_j; and B_x is the set of pairs (l, p) such that l specifies the position of a wildcard (i.e., * and //) in s_x and the p is the binding of that wildcard corresponding to the detected matching. Thus, a PS annotation essentially specifies a data pattern in D that matches some subscription in T_i. Such an annotation can benefit R_j if the specified data pattern also matches some subscription s_y in T_j that aggregates to s_x. When this happens, R_j can very quickly detect that D matches s_y (from processing $A_{i,j}$) without having to actually process D (which is more costly to process than $A_{i,j}$). R_j can then immediately forward D to the relevant downstream router.

Example 4.3 Let us use the set of subscriptions in Figure 4.2(a) and the dissemination system in Figure 4.2(c) to elaborate the PS. s_5 on R_1 is the aggregated subscription of s_1 and s_2 on R_2; and s_6 on R_1 is the aggregated subscription of s_3 and s_4 on R_3. Consider the processing of the document D in Figure 4.2(b) by the router R_1, which has two immediate downstream routers R_2 and R_3. R_1 first detects that D matches the subscription s_6 with the wildcard * and // in s_6

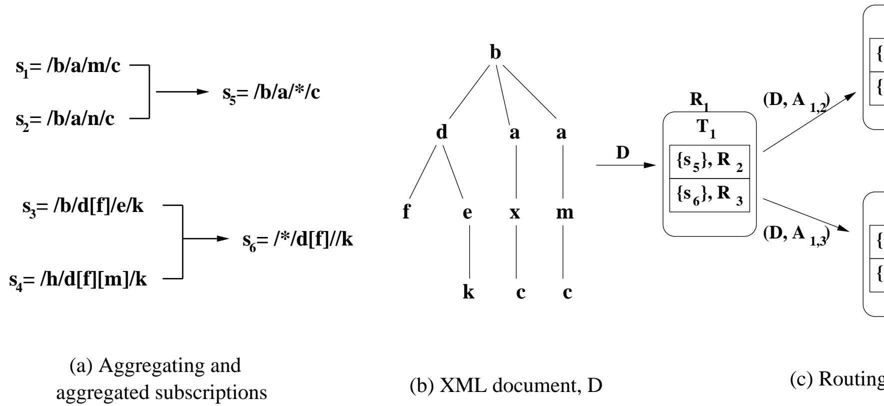


Figure 4.2: XPath Subscriptions, XML Document, and Routing Tables

matching data elements b and e in D, respectively. R_1 then adds the PS annotation $(s_6, \{(1,b),(2,e)\})$ into $A_{1,3}$ and forwards the annotated document (D, $A_{1,3}$) to R_3. Next, R_1 detects that D also matches the subscription s_5 with the * in s_5 matching the data element x. R_1 then adds the PS annotation $(s_5, \{(1,x)\})$ into $A_{1,2}$ and forwards (D, $A_{1,2}$) to R_2. On receiving (D, $A_{1,3}$), R_3 will first check whether any of the aggregating subscriptions for s_6 (i.e., s_3 and s_4) matches s_6 with the bindings $\{(1,b),(2,e)\}$. In this case, there is indeed a matching for the subscription s_3. Thus, R_3 can very quickly forward D to the router R_6 without having to scan and process D. On the hand, when R_2 receives (D, $A_{1,2}$), none of the subscriptions in R_2 (i.e., s_1 and s_2) matches the PS $(s_5, \{(1,x)\})$ in (D, $A_{1,2}$). In this case, R_2 needs to scan and process D before detecting that s_1 matches D. Note that if R_1 had created a PS for the second matching of s_5 in D as well, R_2 would have been able to detect the matching of s_1 earlier without having to process D. □

Positive data (PD). The purpose of a PD annotation is to specify some useful property about the data D that can potentially be exploited by a downstream router R_j to skip the matching of some of its subscriptions in T_j thereby reducing R_j's processing overhead.

In this dissertation, we use a simple PD of the form (p, l), where $p = e_1/e_2/\cdots/e_m$ refers to a path of element names that exists in D, and l refers to the position of the last occurrence of p in D. Here, the position information l means that the end-tag of the element e_1 in the last occurrence of p is l^{th} end-tag in D. To see how a PD (p, l) can be exploited, suppose R_i has just completed parsing the subtree of data elements rooted at the lth element in D, then R_i can safely ignore all of the subscriptions that contain the pattern p from further processing since such subscriptions are guaranteed not to match the remaining yet-to-be-processed portion of D. This subscription pruning optimization can improve performance particularly if the location l is early or if there are many subscriptions in T_i that contain p.

4.3.2 Negative Annotations

The main idea behind negative annotations is to identify the set of subscriptions in the downstream router that are guaranteed not to match the document D being forwarded. In this way, the downstream router can optimize its performance by eliminating the need to compare against such subscriptions against D.

Negative subscription (NS). The NS annotation for a downstream router R_j (w.r.t. D) is a list of the identities of all the non-matching subscriptions in S_j (i.e., $S_j^-(D)$). This information can be exploited by R_j to skip the matching of all the aggregating subscriptions that were aggregated to $S_j^-(D)$. Specifically, for each subscription s in T_j, if s is an aggregating subscription of an aggregated subscription $s' \in S_j^-(D)$ (i.e., $s \sqsubseteq s'$), then by the containment property, the fact that D did not match s' at R_i necessarily implies that D will not match s at R_j. Thus, the matching of s against D at R_j is redundant and can be skipped without affecting correctness.

Negative data (ND). Besides using non-matching aggregated subscriptions to skip the matching of corresponding aggregating subscriptions, a more general approach to enable a similar optimization is to exploit the absence of certain data patterns in D to skip the matching of all subscriptions in R_j that contain such patterns. As an example, if R_i knows that D does not contain the path of elements $p = A/B$, this negative information can be beneficial to R_j if there is a large collection C_p of subscriptions in T_j that contain such a pattern p. By similar reasoning using the containment property, R_j can safely skip the matching of the subscriptions in C_p. In this dissemination, we use simple data patterns in the form of linear paths of element names for ND annotations.

4.3.3 Impact on Matching Protocol

The *matching protocol* of a router R_i refers to the two key decisions that R_i makes when it detects that some subscription corresponding to a downstream router R_j matches D. The first deals with whether R_i should forward D immediately to R_j; and the second deals with whether R_i should continue matching D against other subscriptions related to R_j. This section discusses the impact of piggyback optimization on the options for the matching protocol.

Eager forwarding with skipping. For routers in conventional pub/sub systems (without piggyback optimization), the matching protocol adopted is that when a document D is detected at an upstream router R_i to match some subscription associated with some downstream router R_j, R_i will immediately forward D to R_j and skips the matching of subscriptions associated with R_j. Forwarding a document as soon as possible helps to improve response time, while skipping unnecessary subscription matchings helps to reduce the processing overhead. We refer to this conventional protocol as *eager forwarding with skipping* (denoted by ES).

Lazy forwarding without skipping. The conventional *ES* approach of forwarding D to a downstream router R_j as soon as a subscription matching for R_j is detected generally occurs when D has not been completely processed. The eager forwarding protocol has two implications with regards to the use of annotations. First, negative annotations cannot be included in the forwarded document; and second, only limited PS annotations (derived from the processed portion of D) can be used. Given that negative annotations could potentially help to skip a large number of subscriptions in R_j and PS annotations could enable a document to be forwarded quickly without processing the document, it might actually be beneficial to delay the forwarding of D to R_j until D has been completely processed at R_i. Clearly, for this "lazy" forwarding protocol to generate additional annotations for a matching downstream router R_j, it is necessary for R_i to continue matching D against the subscriptions that correspond to R_j. We refer to this protocol as *lazy forwarding without skipping* (denoted by L).

There is a performance tradeoff between the *ES* and L protocols. On the one hand, by forwarding D immediately to a downstream router, *ES* can help to reduce the response time of delivering a document to matching data consumers. On the other hand, by delaying the forwarding to generate both negative as well as complete PS annotations, L can potentially minimize the matching cost at each downstream router by (1) using negative annotations to skip the processing of many subscriptions, and (2) using PS annotations to enable D to be quickly forwarded without having to first parse D. L is particularly cost-effective if at the time a subscription matching is detected at R_i, only a small proportion of D has not been processed.

Combining annotations and protocols. Based on the preceding discussion, the design space of our piggyback optimization consists of four basic annotation

types (PS, PD, NS, and ND) and two matching protocols (ES and L). A data dissemination strategy is formed by choosing a subset of annotation types together with a matching protocol. We use P^{α}_{β} to denote a dissemination strategy, where $P \in \{ES, L\}$ refers to the matching protocol used; $\alpha \subseteq \{+s, +d\}$ refers to the set of positive annotations used; and $\beta \subseteq \{-s, -d\}$ refers to the set of negative annotations used. Here, $+s$, $+d$, $-s$, and $-d$ denote, respectively, PS, PD, NS, and ND annotations. For conciseness, we use $+sd$ (resp., $-sd$) to represent $\{+s, +d\}$ (resp., $\{-s, -d\}$); moreover, an empty set value is simply represented by a blank, and a singleton value $\{x\}$ is abbreviated to x. For example, the conventional dissemination strategy is denoted by ES; and a strategy that uses lazy forwarding with PS, PD, and ND annotations is denoted by L^{+sd}_{-d}.

Note that it is not meaningful to have a dissemination strategy that involves some negative annotation type together with the ES policy. This is because negative annotations cannot be added to D if it is eagerly forwarded when the processing of D has not completed. Therefore, in this work, we do not consider dissemination strategies ES^{α}_{β} with $\beta \neq \emptyset$.

4.4 Generating Annotations

In this section, we discuss the details of how an upstream router R_i computes the various annotations (i.e., $A_{i,j}$) for a data D to be forwarded to a downstream router R_j. Except for NS annotations, all the other annotations in $A_{i,j}$ can be created more effectively if they exploit knowledge of the subscriptions in the downstream router R_j. To achieve this, our approach generates PS, PD, and ND annotations in two steps, referred to as the *offline step* and *online step*. The offline step is performed only once as part of the routing protocol to set up the routing tables in

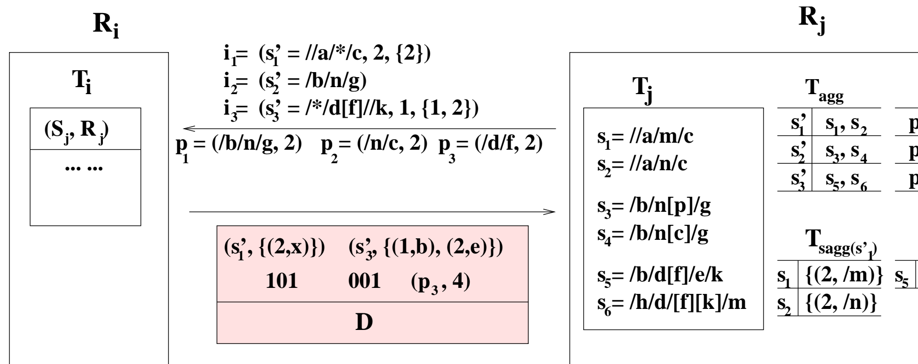


Figure 4.3: Generating & Processing Annotations

the routers. Specifically, when a downstream router R_j is advertising its aggregated subscriptions to each of its upstream routers, we also make use of this opportunity to transmit some useful information that is derived from R_j's subscriptions to the upstream routers. This derived information from R_j will be stored by the upstream routers and used to create annotations in the online step for documents that are forwarded to R_j. The online step is performed by an upstream router each time it needs to forward a document to some downstream router.

In general, since there are many possible options for each annotation type, we devise a benefit metric for each annotation type to enable the effectiveness of annotations to be compared so that a reasonably small set of beneficial annotations can be judiciously selected (for inclusion in the document header) that both maximizes the performance improvement for the downstream router as well as ensures transmission efficiency.

In the following subsections, we first describe, for each of the four annotation types, the intuitive benefit metric used to select annotations and how the annotations are created. After that, we give a comprehensive approach to select the set of annotations among all types of annotations given the size limitation for the header annotations.

4.4.1 Positive Subscription Annotation (PS)

Intuitively, a PS annotation (s, B) is beneficial for a downstream router R_j if s is a simple aggregated subscription of many subscriptions in R_j as this increases the chance that the pattern specified by (s, B) matches a subscription in R_j. On the other hand, it is also desirable for the size of the binding B to be small so that the annotation is space-efficient. We therefore define the benefit of using a subscription s for a PS annotation in $A_{i,j}$ as follows :

$$benefit(s) = \frac{|S'|}{\sum_{i \in w(s)} size(i)}$$

where S' is the set of simple aggregating subscriptions of s in R_j, $w(s)$ is the set of the wildcards in s, and $size(i)$ is size of binding values for the ith wildcard of s. Note that a subscription s has no benefit if $S' = \emptyset$. R_i will select the most beneficial PS annotations based on the above metric.

PS annotations to be included in $A_{i,j}$ are computed in two steps. In the offline step, R_j will identify a set of candidate subscriptions that can be used for PS annotations and advertise these candidates to R_i. In the online step, whenever R_i needs to forward a document to R_j, R_i will create PS annotations from a subset of these candidates by adding relevant data bindings.

More specifically, in the offline step, after R_j has aggregated its subscriptions, R_j derives the information $(s,\ |S'|,\ l)$ for each simple aggregated subscription s computed by R_j, where S' is the subset of subscriptions in R_j that was aggregated to s such that s is a simple aggregated subscription of each subscription in S' (defined in Section 4.3.1); and l is a list of wildcard positions indicating which of the wildcards (// and *) in s need to be instantiated with data path bindings to match some aggregating subscription in S'.

Example 4.4 Consider the example in Figure 4.3, where the six subscriptions in R_j are partitioned into three sets: $S_1 = \{s_1, s_2\}$, $S_2 = \{s_3, s_4\}$, and $S_3 = \{s_5, s_6\}$, which are aggregated, respectively, into s'_1, s'_2, and s'_3. And for simple aggregated subscriptions s'_1 and s'_3, $S'_1 = \{s_1, s_2\}$ and $S'_3 = \{s_5\}$. The derived information generated by R_j in the offline step for these aggregations are shown as i_1, i_2, and i_3 in Figure 4.3. Observe that $i_1 = (s'_1, |S'_1|, l_1)$, where $|S'_1| = 2$ and $l_1 = \{2\}$ indicating that only the second wildcard (i.e., *) requires a data binding; however, the first wildcard (i.e., //) does not require a data binding to transform s to any of the subscriptions in S. On the other hand, since s'_3 is the simple aggregated subscription of only $s_5 \in S'_3$, $i_3 = (s'_3, |S'_3|, l_3)$, where $|S'_3| = 1$ and $l_3 = \{1, 2\}$. □

The collection of derived information $(s, |S'|, l)$ will be passed to upstream routers of R_j when R_j advertises its aggregated subscriptions to them. In the online step, when an upstream router R_i detects a matching subscription s (associated with downstream router R_j) while matching a document D, R_i will first compare s against the derived information from R_j to determine whether s could form a candidate PS annotation.

Example 4.5 Continuing with Example 4.4, suppose that R_i in Figure 4.3 is processing the document D from Figure 4.2(b). When R_i detects that subscription s'_1 matches the data $/b/a/x/c$ in D, R_i uses the derived information $i_1 = (//a/*/c, 2, \{2\})$ to create the PS annotation $(s'_1, \{(2, x)\})$ for s'_1. Based on i_1, R_i knows that only the second wildcard in s'_1 requires a data binding. On further processing D, R_i detects that subscription s'_3 matches D and creates the PS annotation $(s'_3, \{(1, b), (2, e)\})$ for this matching. Figure 4.3 shows the scenario where R_i is forwarding all these PS annotations to R_j along with D (indicated by the shaded box). □

4.4.2 Positive Data Annotation (PD)

A PD annotation (p, l) is beneficial for R_j if p is contained by many subscriptions in T_j and the value of l is small so that R_j can skip many of its subscriptions after processing only a small portion of D. We therefore define the benefit of a pattern p to be $\frac{freq(p)}{pos(p)}$, where $freq(p)$ represents the number of subscriptions in T_j that contain p; and $pos(p)$ represents the position of the last occurrence of p in D. R_i then selects a subset of the PD annotations of the form $(p, pos(p))$ that have the highest benefit values.

PD annotations are also computed in two steps. In the offline step, R_j advertises to R_i a small set of beneficial data patterns P_j (together with their frequencies) derived from the subscriptions in T_j. based on the following benefit metric for a data pattern p: $benefit(p) = freq(p) \cdot \ln(l(p) + 1)$, where $freq(p)$ represents the number of subscriptions in T_j that contain p; and $l(p)$ represents the length of the linear pattern p. The function $\ln(l(p) + 1)$ provides an approximate measure of the probability that the last appearance of p occurs early in the document. In the online step, an upstream R_i keeps track of $pos(p)$ for each pattern $p \in P_j$ as it processes D. With the $freq(.)$ information from R_j and the $pos(.)$ information that it derives, R_i can approximately select the most beneficial PD annotations for $A_{i,j}$.

Example 4.6 Consider again the example in Figure 4.3 which shows that R_j is advertising three candidate data patterns p_1, p_2 and p_3 (along with their frequencies) to R_i for possible use as PD annotations. In the online step, after R_i has completed processing D (from Figure 4.2(b)), R_i detects that $pos(p_3) = 4$ (note that there is no occurrence of p_1 and p_2 in D). It decides to create the PD annotation $(p_3, 4)$ and forwards it to R_j. □

4.4.3 Negative Subscription Annotation (NS)

An NS annotation is a subscription $s \in T_i$ that did not match D; and it is more beneficial to R_j if there are more subscriptions in T_j that aggregate to s as it enables R_j to skip the processing of a larger number of subscriptions. The benefit of each $s \in T_i$ is therefore defined to be the number of subscriptions in T_j that aggregates to s. NS annotations are computed in two steps. In the offline step, R_j notifies to its upstream router R_i the number of its subscriptions that aggregate to each aggregated subscription. While processing a document D during the online step, R_i selects from among the subscriptions in T_i that did not match D, the subset with the highest benefit values as NS annotations. However, as the total number of subscriptions is generally not too large, all the non-matching subscriptions can be specified concisely and precisely using a bitstring with each subscription represented by a single bit such that the bit is turned on if and only if the subscription is non-matching.

4.4.4 Negative Data Annotation (ND)

An ND annotation is of the form of a linear data pattern p that is absent in D. Intuitively, a data pattern p is more beneficial to R_j if there are more subscriptions in T_j that contain p and the probability of p's occurrence in D is low. In fact, ND annotations can be viewed as a special case of PD annotations with $pos(p) = 0$. Thus, we can use the same metric $freq(p) \cdot \ln(l(p) + 1)$ (defined in Section 4.4.2 for PD annotations) to compare the benefit of different ND annotations. The generation of ND annotations in $A_{i,j}$ follows a similar two-step process, where R_i advertises to R_j a set of candidate data patterns during the offline step; and during the online step, R_i keeps track of the candidate patterns that did not occur in the document being processed, and concisely represent the non-matching data patterns

as a bitstring in the ND annotations.

Example 4.7 After R_i in Figure 4.3 has processed the document D from Figure 4.2(b), R_i detects that D did not match the subscription s'_2 and that the data patterns p_1 and p_2 did not occur in D. Thus, R_i can create the ND annotation 001 and the NS annotation 101 for R_j. □

4.4.5 Annotation Selection

The previous sections provide some intuitions to select annotations independently among each individual type. A more general problem for annotation selection is given the size limitation for the header annotations, how to select a set of annotations that can achieve largest improvement while the total size of these annotations fits the header limitation. This section presents the approach to solve such general problem.

For each annotation s (s can be PS,PD,NS, or ND), we can estimate the performance improvement by s, which is denoted as $value(s)$; and we can compute the size of s, which is denoted as $size(s)$. The annotation selection problem can be formally defined as follows.

Definition 4.4.1 (Annotation selection problem) *Given the size limitation K for the header annotations, select the set of annotations (denoted as S) such that (1) $\sum_{s \in S} size(s) < K$; (2) $\sum_{s \in S} value(s)$ is maximized.*

Value estimation. The improvement of annotations has two types : (1) to eliminate unnecessary queries, let λ denote the benefit by eliminating one query; (2) to immediately forward the data at the downstream router, let θ denote such benefit. To process each annotation also incurs some additional cost, we assume the processing cost for the annotations PS, PD, NS and ND as α_{PS}, α_{PD}, α_{NS} and α_{ND}

respectively. Then the value of each annotation s is defined by the improvement of s divided by the processing cost of s.

Given a subscription s on some router, let ρ_s denote the probability that s is matched; and given a data pattern p, let ρ_p denote the probability that p appears in the document. The following gives the value estimation for each type of annotation.

PS: PS has both the above two types of improvement. Given a subscription s_i, let $A(s_i)$ denote set of subscriptions that are aggregated to s_i and satisfy the requirements for PS, and $|A(s_i)|$ denote the number of subscriptions in $A(s_i)$. The PS is useful when the subscription is matched on the upstream router, and there exists some subscription $s_j \in A(s_i)$ that s_j matches the annotations. We assume that the probability of a subscription s_j matches the annotations is k_j. For the set of subscriptions $A(s_i)$, let $A^d(s_i)$ denote the subscriptions from the downstream router; and let $A^l(s_i)$ denote the subscriptions from local users, then $A(s_i) = A^d(s_i) \cup A^l(s_i)$. The $value(ps_i)$ can be computed as :

$$value(ps_i) = \frac{\rho_{s_i} * \sum_{j=1}^{|A(s_i)|} k_j * \lambda + \rho_{s_i} * \sum_{j=1}^{|A^d(s_i)|} k_j * \theta}{\alpha_{PS}}$$

PD: The benefit of PD is to eliminate unnecessary subscriptions. Given a data item d_i, let $l(d_i)$ denote the last occurrence of d_i in the document, and let $N(d_i)$ is the set of subscriptions that can be eliminated after position $l(d_i)$, then

$$value(pd_i) = \frac{\frac{l(d_i)}{l} * |N(d_i)| * \lambda}{\alpha_{PD}}$$

NS: The benefit of NS is to eliminate unnecessary subscriptions due to unmatched aggregated subscriptions. Given a subscription s_i on the upstream router, let $N(s_i)$ denote the set of subscriptions on the downstream router that are aggregated to s_i, then $|N(s_i)|$ is the number of subscriptions in $N(s_i)$. The

NS is useful if the query is non-matching on the upstream router, then

$$value(ns_i) = \frac{(1 - \rho_{s_i}) * |N(s_i)| * \lambda}{\alpha_{NS}}$$

ND: The benefit of ND is also to eliminate unnecessary subscriptions due to absent data patterns. Given a data pattern p, let $N(d_i)$ denote the set of subscriptions on the downstream router that contain d_i, then $|N(d_i)|$ is the number of subscriptions in $N(d_i)$. The ND is useful if the data item is absent from the XML document, then

$$value(nd_i) = \frac{(1 - \rho_{d_i}) * |N(d_i)| * \lambda}{\alpha_{ND}}$$

Size computation. The following presents the computation of size for each type of annotation.

PS: The size of PS is estimated based on the subscriptions. Given the aggregated subscription s_x and a simple aggregating subscription s'_i of s_x, we know the required bindings B^i_x such that s_x can be made to match s'_i. B^i_x is the set of pairs (l_i, p_i) such that l_i specifies the position of a wildcard in s_x and p_i is the bindings of the wildcards to make s_x match s'_i. We use the average length of p of all s_x's simple aggregating subscriptions to estimate the size of PS for s_x. Let S'_x denote the set of s_x's simple aggregating subscriptions, and let k_{s_i} denote the probability that the aggregating subscription $s_i \in S'_x$ matches the annotations of s_x. The size of PS for s_x is computed using the following function.

$$size(s_x) = \frac{\sum_{s_i \in S'_x} (\sum_{p_j \in B^i_x} |p_j|) * k_{s_i}}{\sum_{i \in S'_x} k_{s'_i}} * avg_{ElementSize}$$

PD: The size of PD is 8 bits, where 4 bits are used to specify the identity of the data item and the other 4 bits are used to specify the last occurrence of this data item.

NS: Only 1 bit is required for each NS to indicate the matching status of the subscription, where 1 represents that this subscription is matched, and 0 represents the non-matching.

ND: Similar with NS, only 1 bit is required for each ND.

Based on the computed *value* and *size* information for the annotations, we can solve the annotation selection problem. The annotation selection problem is a variant of the knapsack problem [64], since the *value* of an annotation is equivalent to the value of each item and the *size* of an annotation is equivalent to the cost of each item. It is known that the knapsack problem is an NP-complete problem, it follows that the annotation selection problem is NP-complete.

The dynamic programming approach is used to solve this problem. Given n annotations $a_1, a_2, \ldots, a_n$, let k_i denote the size for annotation a_i and v_i denote the value for a_i. The goal is to maximize total value while the constraint that the total size is less than K is satisfied. For each $i \leq K$, define $A(i)$ to be the maximal value obtained with total size less than or equal to i. Then the recursively structure is defined as follows :

$$A(0) = 0;$$

$$A(i) = max\{v_j + A(i - k_j) | k_j \leq i\};$$

Thus the solution to this problem is $A(K)$. The running time of this dynamic programming solution is $O(nK)$.

Another solution to solve this problem is to use greedy algorithm. These an-

notations are sorted based on the metric $value(s_i)/size(s_i)$. Then we select the annotations in the decreasing order of this metric until there is no longer space in the header for more annotations.

4.5 Processing Annotated Documents

This section describes the details of how a router R_j processes an annotated document $(\text{D}, A_{i,j})$ that it receives from some upstream router R_i.

To efficiently and effectively process annotations in an annotated document $(\text{D}, A_{i,j})$, each downstream router R_j maintains the following information:

Aggregation Table, T_{agg}. For each aggregated subscription s' generated by R_j, $T_{agg}(s')$ stores the set of aggregating subscriptions of s'.

Simple Aggregation Table, T_{sagg}. For each aggregated subscription s' generated by R_j, if s' is also a simple aggregated subscription, $T_{sagg}(s')$ stores the set of pairs (s, P), where s is a simple aggregating subscription of s'. P is a set of pairs (l_i, p_i), where l_i specifies the position of a wildcard in s' and p_i specifies a data pattern binding such that if for each $(l_i, p_i) \in P$, the lth wildcard in s' is replaced by the pattern p_i, then the transformed s' will match s.

Pattern Table, T_{pat}. For each data pattern p that R_j has advertised to its upstream routers during the offline step for PD or ND annotations, $T_{pat}(p)$ stores the subset of subscriptions in R_j that contains p such that if a document D does not match p, then D also does not match any of the subscriptions in $T_{pat}(p)$.

Non-Matching Array, A_{not}. A_{not} is a bit-array of size equal to the number of subscriptions in R_j such that the ith bit corresponds to the ith subscription in R_j. Each $A_{not}[i]$ is initialized to zero at the start when R_j receives D, which could be set to one as R_j processes the annotations and D. Specifically, $A_{not}[i]$ is set to one

if and only if the ith subscription in R_j is guaranteed not to match the document D being processed. R_j uses A_{not} to optimize its processing of D by skipping the processing of subscriptions that indicated by A_{not} to be guaranteed to not match D.

Figure 4.3 shows all the tables maintained at R_j for the six subscriptions. Note that all the tables T_{agg}, T_{sagg} and T_{pat} are created only once in the offline step after R_j has advertised his aggregated subscriptions and derived information to its upstream routers. These tables remained static unless there are changes to the subscriptions in R_j. The bit-array A_{not} is the only structure that needs to be initialized and updated for each document that R_j processes.

4.5.1 Processing Annotations $A_{i,j}$

We are now ready to explain how R_j processes an annotated document $(\mathrm{D}, A_{i,j})$ that it receives from R_i using the following four steps.

Step 1: Processing PS Annotations. For each PS annotation $(s', B) \in A_{i,j}$, R_j compares (s', B) against each $(s, P) \in T_{sagg}(s')$. If for each pair $(l_i, p_i) \in P$, there exists a pair $(l'_i, p'_i) \in B$ such that $l_i = l'_i$ and p_i matches p'_i, then D is detected to match subscription s and R_j can immediately forward D to the downstream router (say R_k) associated with subscription s without the need to first process D[1]. Since R_j has not yet processed D, $A_{j,k}$ needs to be derived from $A_{i,j}$; the details are described in Section 4.5.3.

Example 4.8 Suppose that R_j has just received from R_i the annotated document $(\mathrm{D}, A_{i,j})$ (indicated by the shaded box in Figure 4.3) and is processing the PS annotations. For the PS annotation $PS_1 = (s'_3, \{(1, b), (2, e)\}) \in A_{i,j}$ in which

[1]Note that the immediate forwarding due to a matching in $A_{i,j}$ is independent of the forwarding policy being used which applies when there is a matching in D.

$B = \{(1, b), (2, e)\}$, R_j will process PS_1 against each $(s, P) \in T_{sagg}(s'_3)$. For the tuple $s = s_5$ and $P = \{(1, /b), (2, /e)\} \in T_{sagg}(s'_3)$, R_j detects that it matches PS_1 since for $(1, /b) \in P$ there exists a pair $(1, b) \in B$ such that b matches $/b$; and for $(2, /e) \in P$, there exists the pair $(2, e) \in B$ and e matches $/e$. Thus R_j knows D matches s_5 without processing D. □

Step 2: Processing NS Annotations. For each NS annotation $s' \in A_{i,j}$, R_j knows that D will not match any of the subscriptions in $T_{agg}(s')$. R_j therefore updates $A_{not}[i]$ to one for each aggregating subscription $s_i \in T_{agg}(s')$.

Step 3: Processing ND Annotations. For each ND annotation $p \in A_{i,j}$, R_j knows that D will not match any of the subscriptions in $T_{pat}(p)$. R_j therefore updates $A_{not}[i]$ to one for each subscription $s_i \in T_{pat}(p)$.

Step 4: Processing PD Annotations & D. For each PD annotation $(p, \ell) \in A_{i,j}$, R_j will dynamically process each of them (in ascending order of their positions ℓ) as part of its processing of D. Specifically, if R_j has completed parsing some data element in D at position ℓ and (p, ℓ) is the next-to-be-processed PD annotation, then R_j knows that the data pattern p will not occur in the remaining portion of D, and that it is redundant to process any new matchings of subscriptions in $T_{pat}(p)$. Therefore, R_j updates $A_{not}[i]$ to one for each subscription $s_i \in T_{pat}(p)$.

4.5.2 Processing Document D

Based on the preceding discussion, R_j processes all the PS, NS, and ND annotations before it begins to process D (steps 1-3). In some situations (step 1), it is even possible for R_j to forward D to another downstream router without having to process D at all. The PD annotations are processed (step 4) along with the normal processing of D.

When R_j detects that D matches some subscription $(S_k,\ R_k) \in T_j$, there are two cases to consider. If the dissemination strategy used is the ES matching protocol, then R_j will generate the appropriate annotations for $A_{j,k}$ (depending on the annotation types being used) and forward $(\mathrm{D}, A_{j,k})$ to R_k. Moreover, R_j will also skip all subscriptions associated with R_k from further matching as it continues to process D; this is achieved by setting the appropriate bits in A_{not} to one. Otherwise, if the dissemination strategy in use is the L matching protocol, then R_j will only forward D to R_k (with appropriate $A_{j,k}$) after it has completely processed D.

4.5.3 Deriving Negative Annotations

In this section, we consider the scenario where R_j is going to forward D to a downstream router R_k without having yet completed processing the entire document D. An issue that arises from this situation is what are the possible negative annotations (if any) that R_j can include in $A_{j,k}$ given that R_j has not processed some portion of D. Observe that R_j cannot arbitrarily include a subscription s that has not matched the processed portion of D as an NS annotation since s could potentially have matched D if R_j has processed D completely.

It turns out that R_j can actually derive some limited types of negative annotations for $A_{j,k}$: (1)for each NS annotation $s' \in A_{i,j}$, R_j can identify the subset of subscriptions $S_k \subseteq T_{agg}(s')$ that are associated with R_k. Thus, the subscriptions in S_k can be included as NS annotations in $A_{j,k}$; (2)the ND annotations in $A_{i,j}$ can be directly inherited as ND annotations in $A_{j,k}$. However, since the inherited annotations are not specifically optimized for R_k using derived information from R_k, they are generally less beneficial than the customized ND annotations.

4.6 Experimental Study

This section reports the extensive experimental results on the comparison of various dissemination strategies. The results show that the dissemination strategy L_{-sd}^{+s} outperforms the conventional method ES by a factor of 2.

4.6.1 Experimental Testbed

The NS2 network simulator [5] is extended for our experiments by adding application code for content-based routing and piggyback optimization. The subscription indexing method and subscription aggregation approach implemented for each router are based on existing solutions in the literature [39, 38]. It is important to emphasize that our proposed piggyback optimization approach is orthogonal to the specific algorithms used for filtering and aggregation.

The router network topology used was a complete binary tree with fours levels and a total of 15 routers, where each router (except for the root router) has one immediate upstream router at one level above. Data is disseminated from the root router downwards to the leaf routers. Data users can subscribe to any router; and each router aggregates all its subscriptions to a size that is k% of the original set; our experiments used values of 12.5 and 25.0 (the default value) for k. The network bandwidth values used were 1MBps, 10MBps (default), and 100MBps.

Data sets. Our experiments used three synthetic data sets (1) NITF DTD [7], which has been used in previous studies [20, 49, 39]; (2) Treebank [10]; (3) DBLP [3]. For each data set, ten documents were generated using IBM's XML Generator [51]. In addition, we also used one real-life Protein data set [6] by extracting data from it to form small documents. The average sizes of the documents in each of the data sets are shown in Table 4.1, where $NITF_2$ is the default data set. Note

that the document sizes used in our experiments are typical for data dissemination settings [49, 39].

$NITF_1$	$NITF_2$	$NITF_3$	Treebank	DBLP	Protein
50	129.4	200.8	144.1	116.6	276081

Table 4.1: Average document size used (# elements)

Subscriptions. The subscriptions were generated using the XPath generator in [49], and the parameter values used are shown in Table 4.2 with the default values used indicated in bold.

Parameter	Description	Value
L	maximum number of steps	8
ρ_*	probability of "*"	0.1
$\rho_{//}$	probability of "//"	0.1
ρ_λ	probability of nested paths	0.1,**0.2**,0.4
θ	skewness of element names	0,**0.5**
P	#subscriptions per node	2500,**5000**,50000

Table 4.2: Parameter values for subscriptions

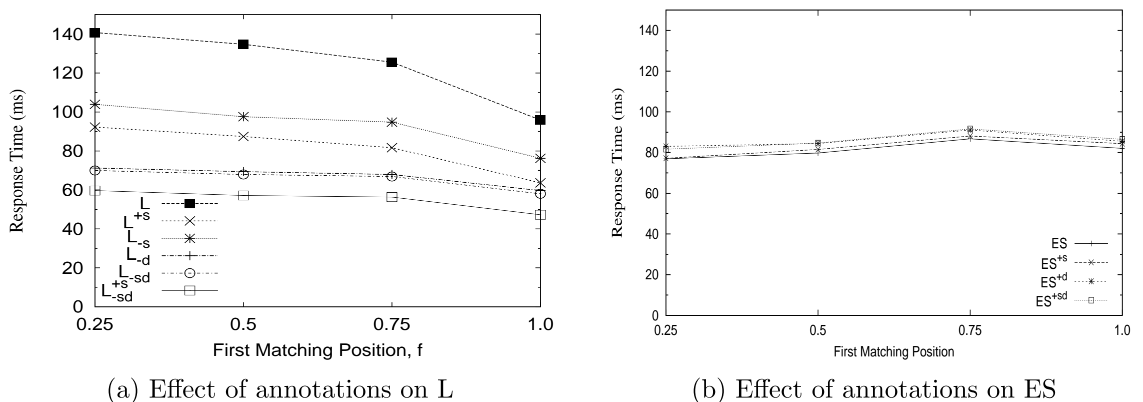
In order to study the effect of the choice of forwarding policy on performance, it is important to be able to control the position in the document at which the first subscription matching is detected. Intuitively, the benefit of early forwarding is more significant if the first subscription matching occurs early in the document; while the benefit of lazy forwarding is more significant if the first subscription matching occurs late in the document. To enable the first subscription matching position to be varied, we inserted an unique element <test> in each generated document (varying the location being inserted), and also added an additional predicate $//test$ to each generated subscription. In this way, a subscription matching can occur only after the <test> element has been parsed in the document. In our experimental graphs, the first matching position is represented as a fraction $f \in [0, 1]$ indicating the proportion of the document parsed before the occurrence

of a first subscription matching. The values used for f are 0.25, 0.5, 0.75, and 1.

Algorithms. We compared the various dissemination strategies P_{β}^{α} discussed in Section 4.3.3 by varying the annotation types (α and β) and matching protocols (P), including the conventional approach ES as a special case. The main performance metric compared is the *response time*, which is defined as the average time taken to disseminate a published document from its source to the relevant users. The response time for disseminating a document D comprises of two key components: (1) the transmission spent in in the network, and (2) the processing time incurred by the routers to process annotations against D, match subscriptions against D, and generate annotations. Each response time reported for a data set is the average response time for disseminating the ten documents in the data set.

Our experiments were conducted on a 3GHz Intel Pentium IV machine with 1GB main memory running Windows XP, and all algorithms are implemented in C++.

4.6.2 Experimental Results



(a) Effect of annotations on L

(b) Effect of annotations on ES

Figure 4.4: Experimental results for different dissemination approaches

Effect of annotation types. Figure 4.4(a) compares the performance of different

annotation types using the lazy forwarding without skipping protocol as the first matching position, f, is being varied on the x-axis. Among the four annotation types, ND improves L the most, followed by PS, NS, and PD. L^{+d} (not shown in Figure 4.4(a) turns out to have similar performance as L, since the effectiveness of PD to skip the matching operations is limited. L_{-d} performs better than L_{-s} for the reason that the NITF DTD has many optional elements such that it is likely that some elements or substrings absent from the documents which makes ND more effective to eliminate non-matching subscriptions. Specifically, for our default setting, 10% of the subscriptions on each router are matching. ND can eliminate 70% of the total subscriptions while NS can eliminate only 37% of them. Our results also show that PS is more effective than NS but less effective than ND.

Observe that when both NS and ND are combined, it improves over ND only slightly. The reason is that a number of subscriptions that can be skipped using NS can also be skipped by ND; consequently, using NS in addition to ND offers very little improvement. However, as the overhead of using NS is small, adopting both negative annotations is still better than using only a single negative annotation. The performance of L_{-sd} can be further optimized by also using PS, and L_{-sd}^{+s} is in fact the best strategy based on lazy forwarding. This is because PS enables a document to be forwarded to some routers very quickly without parsing the document; and when this is not possible, the negative annotations are effective in skipping many subscription matchings.

On the other hand, our experimental results shown in Figure 4.4(b) indicate that positive annotations[2] do not enhance the performance of the eager forwarding policy at all: ES^{+s} has similar performance as ES, while ES^{+d} actually performs worse than ES. This is because only very limited PS annotations can be used

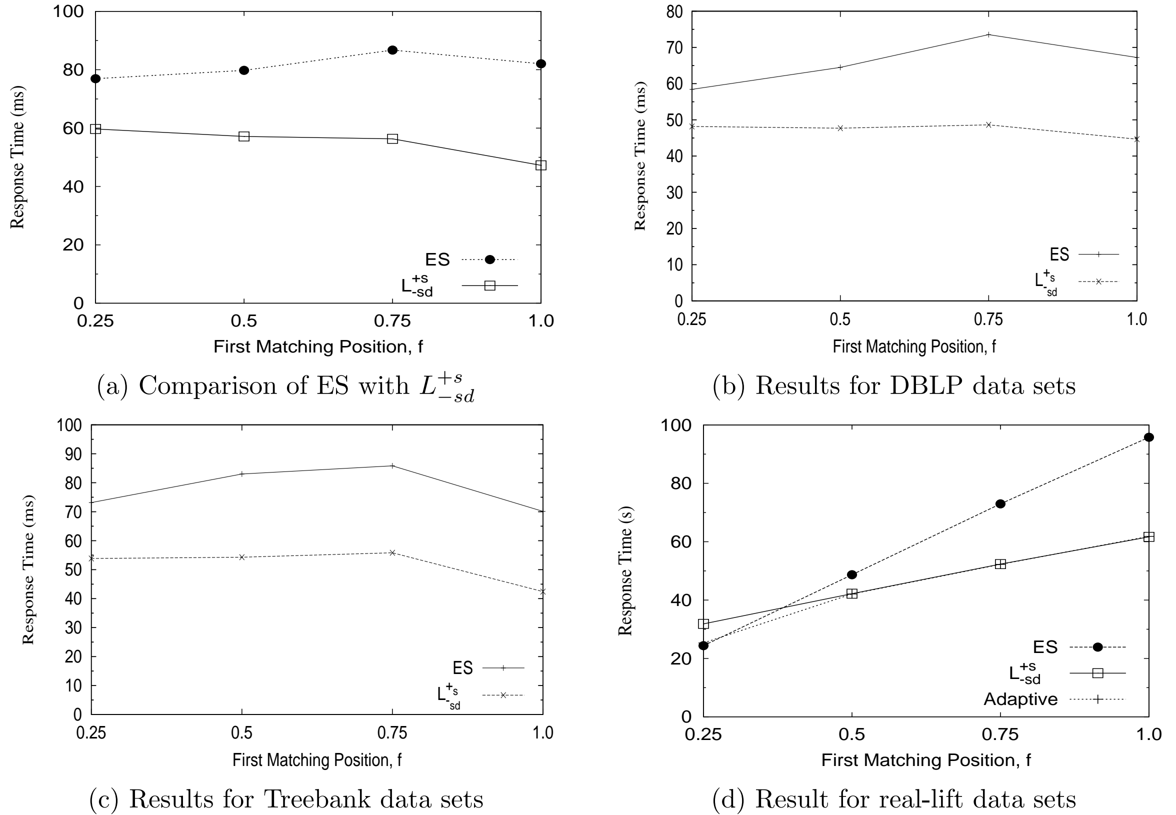
[2]Recall from Section 4.3.3 that negative annotations are not meaningful for eager forwarding.

when a document is forwarded eagerly; and similar to the case for lazy forwarding, PD turns out to be not cost-effective due to the fact that PD only enables a small number of subscriptions to be skipped and its benefit is offset by its processing overhead.

Eager vs. lazy forwarding. Figure 4.5(a) compares the performance of the best eager-forwarding strategy (ES) and the best lazy-forwarding strategy (L^{+s}_{-sd}). The results show that L^{+s}_{-sd} outperforms ES indicating that the slight delay incurred by lazy forwarding is compensated by the improvement gained by the downstream routers from exploiting a more complete set of annotations to optimize their processing. We also observe that as f increases from 0.25 to 1, the improvement by L^{+s}_{-sd} over ES also increases from 22% to 42%. The reason is that as f grows, the lazy forwarding in L^{+s}_{-sd} incurs relatively smaller delay.

Effect of other workload. Figure 4.5(b)(c) show the results on other synthetic data sets, i.e. DBLP and Treebank respectively. We observe the similar trends with the NITF data set that is L^{+s}_{-sd} obviously outperforming ES. Figure 4.5(d) shows the comparison using the real-life Protein data set. Observe that ES is actually better than L^{+s}_{-sd} when $f = 0.25$, but as f increases, L^{+s}_{-sd} outperforms ES with increasing margin. The reason that ES performs better with $f = 0.25$ is due to the large document size (about 10MB) : the benefit of the annotations is offset by the longer delay incurred by L^{+s}_{-sd} when f is very small. We also tried an adaptive approach where ES is used when the first subscription matching occurs early and L^{+s}_{-sd} is used otherwise. Our result shows that this hybrid strategy (indicated as "Adaptive" in Figure 4.5(d)) outperforms ES and L^{+s}_{-sd}.

Effect of bandwidth. Figure 4.6(a) shows the effect of the network bandwidth. With a small bandwidth of 1MBps, L^{+s}_{-sd} has a 15% improvement over ES when $f = 0.25$; which increases to over 30% when f increases to 1.0. As the bandwidth



(a) Comparison of ES with L_{-sd}^{+s}

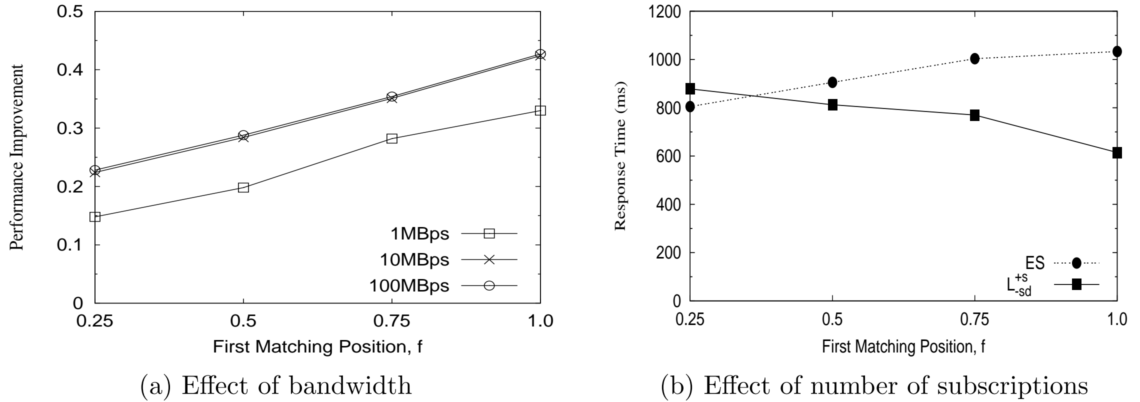
(b) Results for DBLP data sets

(c) Results for Treebank data sets

(d) Result for real-lift data sets

Figure 4.5: Experimental results for different DTD

increases to 10MBps, the performance improvement increases to 22% when $f = 0.25$ and to 42% when $f = 1.0$. This is because the disadvantage of L_{-sd}^{+s} (in terms of the delay in forwarding) becomes even less significant (relative to the processing time incurred by the routers) with a higher network bandwidth. Not surprisingly, our results for a bandwidth of 100MBps are similar to the results for a bandwidth of 10MBps. Note that, the space overhead incurred by all annotations used in L_{-sd}^{+s} is 464bytes (the size of PS, NS, and ND are 224, 160, and 80 bytes, respectively) for the default experimental setting where the size of the documents is around 7000bytes. We can see that the size occupied by the annotations is no more than 7% of the document size. Thus, transmitting the additional annotations with the document only incurs a very small overhead in the network delay, while the speedup obtained for the processing on the routers is up to a factor of 2.



(a) Effect of bandwidth

(b) Effect of number of subscriptions

Figure 4.6: Effect of bandwidth & number of subscriptions

Effect of number of subscriptions. When the number of subscriptions P is increased from 2500 to 5000, we observe that the improvement of L^{+s}_{-sd} over ES becomes larger (from 35% for P = 2500 to 42% for P = 5000). This is due to an increase in the number of subscriptions that can be skipped by NS and ND. Figure 4.6(b) shows the results when P is increased to 50000. We observe that for larger f (i.e. $f = 0.75$ and $f = 1.0$), the improvement of L^{+s}_{-sd} over ES stays the same (e.g., 41% at $f = 1.0$). However, for smaller values of f, the improvement of L^{+s}_{-sd} over ES diminishes slightly; e.g., with $f = 0.5$, the improvements decreases from 28% for P = 5000 to 11% for P = 50000. When $f = 0.25$, ES is more efficient than L^{+s}_{-sd}. The reason is because for L^{+s}_{-sd}, the root router needs to match against all the subscriptions; and when $f = 0.25$, the delay at the root router is increased more significantly for L^{+s}_{-sd} (relative to ES) such that the the overall efficiency of L^{+s}_{-sd} becomes diminished.

Effect of data size. Intuitively, a larger document has two conflicting effects on the performance of L^{+s}_{-sd}. On the one hand, the delay incurred by L^{+s}_{-sd} is expected to increase, but on the other hand, the improvement obtained from skipping subscription matchings would also become more significant with a larger document. Our experimental results shown in Figure 4.7(a) (using data sets $NITF_1$, $NITF_2$

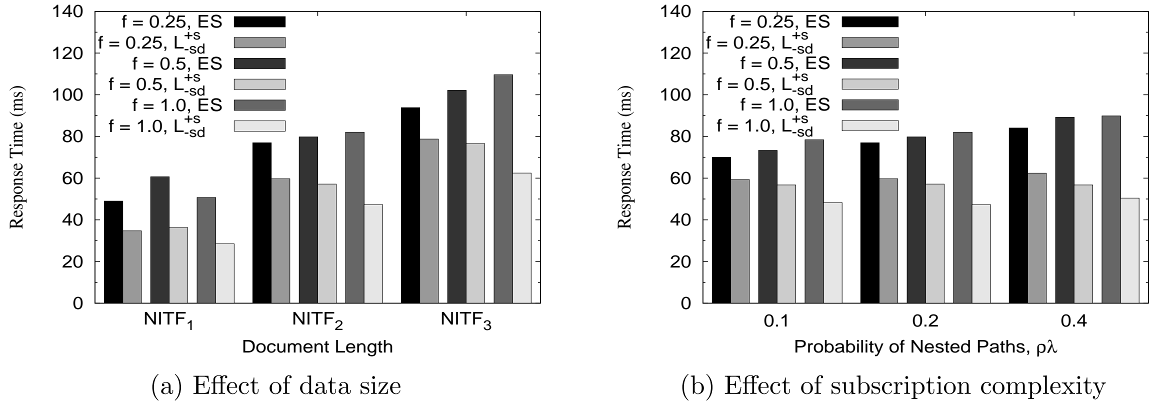
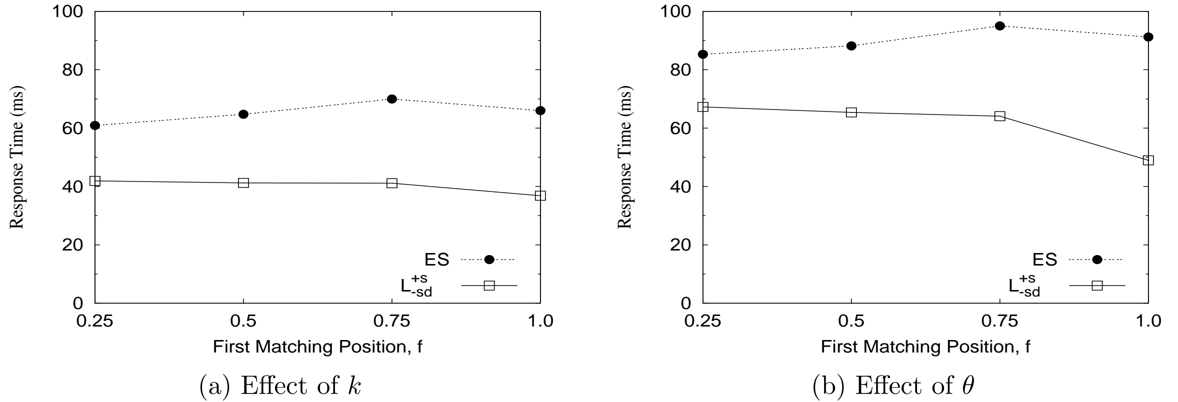


Figure 4.7: Effect of data size & subscription complexity

and NITF_3) indicate that for larger documents (e.g. NITF_3) when $f = 0.25$, the improvement of L^{+s}_{-sd} over ES becomes smaller (i.e. 16%), compared with the improvement for NITF_2 (i.e. 22%). But their performance margin widens significantly as the value of f increases. When $f = 1$, the improvement of L^{+s}_{-sd} over ES for NITF_3 is 41%, which is almost the same with NITF_2 (i.e. 42%).

Effect of subscription complexity. By varying the ρ_λ parameter to increase the complexity of the subscriptions, we observe that the performance gain of L^{+s}_{-sd} over ES, shown in Figure 4.7(b), becomes more significant with more complex subscriptions. In particular, when ρ_λ increases from 0.1 to 0.4, the improvement of L^{+s}_{-sd} over ES (with $f = 1.0$) increases from 38% to 44%. The reason is because the processing cost of the subscriptions increases with their complexity; thus, the savings from skipping subscription matchings also become more significant.

Other experiments. Figure 4.8(a) shows the results by setting the parameter k from 12.5. With a smaller value of k (compared with the default setting $k = 25$), PS becomes slightly less effective since the more brute aggregation makes the simple aggregation requirements harder to satisfy. Figure 4.8(b) shows the results by setting parameter θ to 0. The results demonstrate the similar trends with the default setting. It indicates that varying the distribution to generate the



(a) Effect of k

(b) Effect of θ

Figure 4.8: Effect of k & θ

subscriptions has little effect on the performance trend.

Throughput comparison. Although L^{+s}_{-sd} performs better in terms of the response time, the results show that the throughput of L^{+s}_{-sd} is slightly worse than ES : when $f = 0.25$, the throughput of L^{+s}_{-sd} is 72% of that of ES; when $f = 1.0$, it increases to 88% of ES's throughput. The total throughput is determined by the slowest router in the system. In L^{+s}_{-sd}, the root router is the bottleneck since there are no annotations that can be used and all subscriptions have to be evaluated, thus the throughput of L^{+s}_{-sd} loses to ES. However, at all downstream routers, the disseminated documents are processed much faster using L^{+s}_{-sd} than using ES because of the effective annotation inserted. Therefore, if we use a more powerful router at the root node, L^{+s}_{-sd} can also achieve higher throughput than ES.

Discussions. Based on our experimental results, we have the following observations on the effectiveness of the various annotation types. First, the effectiveness of PD annotations is found to be limited as only very few subscriptions can be pruned using PD annotations and the marginal saving from using PD annotations is offset by its overhead. Second, comparing ND and NS annotations, the former is generally more effective than the latter. This is due to a combination of two reasons. As the disseminated documents are small and the data schema has many optional

elements, many of the elements in the schema (and hence also appearing in many queries) do not occur in the small documents. Thus, ND annotations can help prune many queries in a downstream router. Moreover, the process of aggregating subscriptions often results in a document D matching an aggregated subscription s' in an upstream router even when all of the aggregating subscriptions of s' in a downstream router do not match D. Finally, the relative effectiveness between ND and PS annotations is less clear due to the different nature of their benefits. Recall that the benefit of ND annotations is in reducing subscription matching in a downstream router, while the benefit of PS annotations lies in enabling the forwarding of document without processing it. Neither ND nor PS annotations are found to be significantly more effective than the other in our experiments.

4.7 Summary

This chapter presented a novel approach to optimize the performance of content-based dissemination of XML data by piggybacking useful annotations to the document being forwarded so that a downstream router can leverage the processing done by its upstream router to reduce its own processing overhead. Four useful annotations are proposed, i.e. PS, PD, NS and ND, and these annotations have the benefits as follows.

- PS is useful to detect some matching subscriptions on the downstream router such that the document can be forwarded along certain outgoing links on the downstream router without parsing the document.
- PD helps to skip some processing operations on the downstream router.
- NS and ND are effective to largely reduce the number of processed subscriptions on the downstream router.

A large design space of dissemination strategies that combine four types of annotations and two forwarding policies is examined. The experimental study demonstrates both the feasibility and effectiveness of the new approach. In particular, the strategy of combining lazy forwarding with three types of annotations (i.e. PS, NS and ND) turns out to be the best option that outperforms the conventional method by a factor of 2.

Chapter 5

Handling Fragmented XML Data

5.1 Introduction

The previous chapter focused on the efficiency aspect of content-based dissemination of XML data. This chapter and the next chapter will investigate the approaches to extend the functionalities of the system. As mentioned in Chapter 1, the emergence of XML as a standard for information exchange on the Internet has led to an increased interest in using more expressive subscription/filtering mechanisms that exploit both the *structure* and the *content* of published XML documents. The existing filtering approaches in content-based dissemination of XML data require that the data is published as a complete XML document. However, the increasing use of XML in Web-based services and applications has led to the importance of processing XML data in the distributed context, where the XML data is stored, disseminated and processed in terms of fragments. Moreover, the prevalent use of resource-limited mobile device as client devices has also motivated the need to process and disseminate the fragments. And the widely deployed sensor devices for monitoring always collect data in fragments. These observations indicate that more and more data may be collected and published in terms of fragments, thus

there is indeed a need to extend the functionality of the current content-based XML dissemination system to support the XML data that is published in fragments.

Many research attentions have been paid on processing fragmented XML data that spans many diverse issues from how to fragment XML document [58, 32, 28, 113], distributed query processing on fragmented XML data [105, 48], managing the distributed XML data [18, 19], and processing XQuery queries on streaming XML fragments [96]. However, there is no work that examines the problem of efficiently matching boolean XPath queries on fragmented XML data in content-based dissemination. The matching of XPath queries with fragmented data not only provides additional service for the XML dissemination system to handle the publications in fragments, but also opens up new opportunities for optimization the query matching by routers. The challenge of evaluating boolean XPath queries on fragmented XML data is how to efficiently and effectively schedule and optimize the processing of fragments so as to "short-circuit" the query evaluation as early or as much as possible by determining the evaluation result (either finding a matching in the data if it exists or concluding that the outcome is non-matching) with minimal unnecessary/redundant fragment evaluations. To the best of our knowledge, the work in this thesis represents the first comprehensive approach to schedule and optimize the evaluation of boolean XPath queries on fragmented XML data.

This chapter presents the work on matching XPath-based subscriptions *directly* on fragmented XML documents without reconstructing the original documents. The proposed query processing strategy consists of three main steps :

- identifying the relevant subqueries to evaluate on each fragment.
- scheduling the order to evaluate the fragments.
- optimizing each fragment evaluation to minimize unnecessary and redundant

processing.

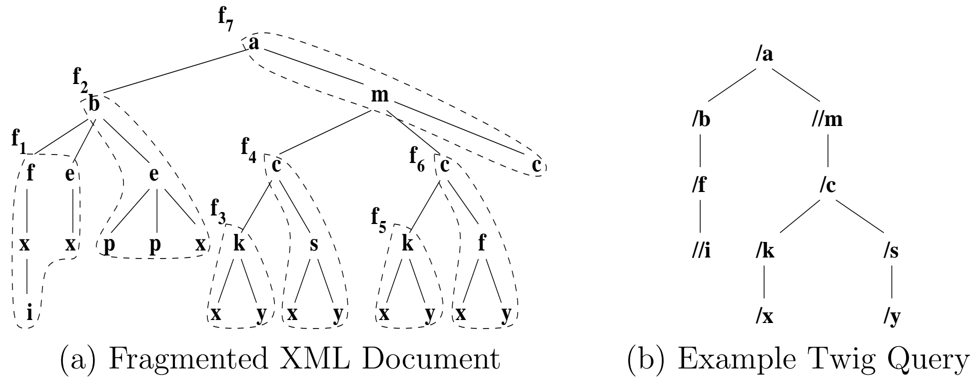
The proposed fragmented approaches are shown to outperform the traditional non-fragmented approach by up to a significant margin.

The rest of this chapter is organized as follows. Section 5.2 introduces some useful preliminaries and definitions. Section 5.3 gives an overview of our proposed approach for disseminating fragmented XML data. Section 5.4 describes the proposed approach in detail, which covers the XML fragmentation model, the steps to matching queries on fragments, and two optimization mechanisms. An extensive performance study is presented in Section 5.5. Finally, Section 5.6 concludes this chapter.

5.2 Preliminaries and Definitions

The XPath expressions used as the subscriptions in content-based dissemination of XML data has been introduced in Chapter 2. This work focused on a commonly used subclass of XPath queries called *tree pattern(or twig) queries* that essentially supports XPath's / and // location steps and predicates for path expressions as mentioned in Section 2.2. A tree pattern query is represented by an unordered rooted tree, where each node is labelled with an element name or a wildcard that is prefixed by either "/" (for a child-step) or "//" (for a descendant-step).

Given a query q and an XML document d, a *matching of* q *in* d is identified by a mapping from the nodes in q to the nodes in d such that both the following conditions are satisfied: (1) each mapped data node d_i matches its corresponding query node q_j (i.e., either d_i and q_j have the same element tag or q_j has a wildcard tag); and (2) the structural (parent-child and ancestor-descendant) relationships between query nodes are satisfied by their corresponding mapped data nodes. Thus,



(a) Fragmented XML Document (b) Example Twig Query

Figure 5.1: Fragmentation and query models

we say that q *matches* d if there exists at least one matching of q in d; otherwise, q does not match d.

Example 5.1 Consider the XML document d and twig query q in Figure 5.1, each dotted region of data nodes in d (labeled by some f_i) denotes a data fragment which can be ignored for now. For a document d' to match q, d' must have a root element a that satisfies two conditions: (1) a must have a child element b which in turn has a child element f that has a descendant element i; and (2) a must also have a descendant element m that in turn has a child element c that satisfies the two conditions: (a) c must have a child element k that in turn has a child element x; and (b) c must also have a child element s that in turn has a child element y. It can be easily verified that the document d matches the query q. □

Consider a node t_i in a (query or data) tree T. We define the *prefix of* t_i, denoted by $prefix(t_i)$, to be the path of nodes from the root node of T to t_i (inclusive). We define the *minimum (maximum) height of* t_i, denoted by $minHt(t_i)$ ($maxHt(t_i)$), to be the length of the shortest (longest) path from t_i to one of its descendant leaf nodes in T.

Given a query node q_j and a data node d_i, we can view $prefix(q_j)$ and $prefix(d_i)$ as a query tree and a data tree, respectively, and define the matching of $prefix(q_j)$

in $prefix(d_i)$ similarly.

When the data nodes in an XML document d are partitioned into fragments, finding a matching of a query q becomes more complex and requires seeking matchings of different *subqueries* of q among the fragments. Given a query node q_i in q, we define the *subquery rooted at* q_i, denoted by subquery(q_i), to be the query subtree rooted at q_i.

Example 5.2 Consider the XML document d and query q in Figs. 5.1(a) and (b), respectively, where d is partitioned into seven fragments indicated by the dashed regions of nodes (f_1 to f_7). In f_4, $prefix(c) = /a/m/c$, $minHt(c) = 2$, and $maxHt(c) = 2$. In q, $prefix(i) = /a/b/f//i$, $minHt(a) = 3$, and $maxHt(a) = 4$. Note that *subquery*$(/f)$ matches f_1, *subquery*$(/k)$ matches f_3, and *subquery*$(/s)$ matches f_4. Together with the matchings of query nodes $/b$ in f_2, $/c$ in f_4, and $/a$ and $/m$ in f_7, we have a matching of q in d. □

5.3 Overview of Disseminating Fragmented XML Data

Our approach of processing boolean XPath queries on fragmented XML data consists of three main steps. The first step identifies *what* relevant subqueries to evaluate on each fragment; the second step decides on the *order* in which the fragments are evaluated; and the third step deals with *how* each fragment is evaluated. Figure 5.2 gives an overview of our approach. Each of these steps will be elaborated later in this chapter.

1. **Identify relevant fragments**. Since the data nodes are partitioned among several fragments, finding a matching of an input query q in a fragmented

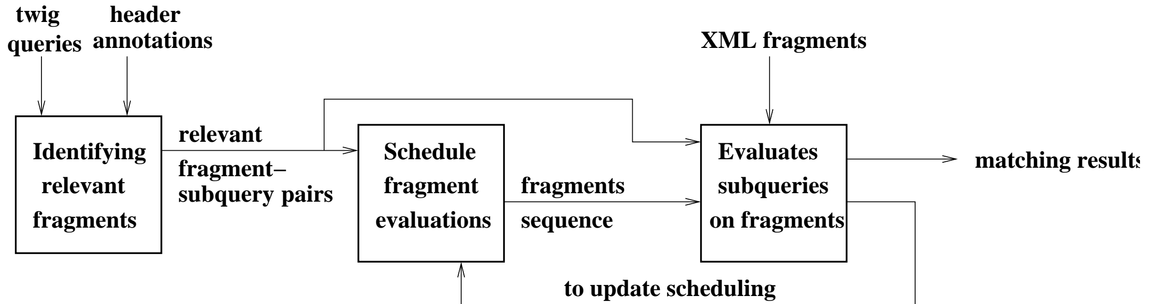


Figure 5.2: Overview of processing XML fragments

XML document generally requires finding the matchings of different query nodes of q in the various fragments. Thus, the first step is to make use of the collection of fragment header information to identify a set of "relevant" matchings to determine for each fragment. The goal is to minimize both the number of relevant matchings as well as the number of fragments to be evaluated.

2. **Schedule fragment evaluations**. The second step is to determine an order in which to process the fragments based on the relevant fragment-subquery pairs obtained in the first step. The goal is to "short-circuit" the query evaluation as early or as much as possible: in the case that there is a matching of the input query, the schedule should identify the matching data nodes early; otherwise, the schedule should avoid processing

3. **Evaluate subqueries on fragments**. This step deals with how to efficiently optimize and process the set of relevant subqueries associated with each fragment. The evaluation results on one fragment may affect the scheduling for the remaining fragments, therefore we may update the scheduling during the evaluations. A key challenge is how to efficiently maintain the intermediate set of matching nodes to facilitate the detection of a matching if it exists.

5.4 Algorithm for Processing XML Fragments

This section presents the detailed approach of processing boolean XPath queries on fragmented XML data. Firstly, the XML fragmentation model used in this work is introduced. Then the structures used to store the fragment header information is discussed. Then the three steps of this approach are elaborated. And finally, two dynamic optimizations are presented.

5.4.1 XML Fragmentation Model

This work assumes a very general data fragmentation model, where an XML document is partitioned into a collection of fragments that satisfy the following three properties:

P1. The fragments are *disjoint*; i.e., each document node belongs to exactly one fragment.

P2. The fragments are *acyclic* in the sense that whenever a fragment f_i contains some data node that is an ancestor of some data node in another fragment f_j, then f_j can not also contain a data node that is an ancestor of some data node in f_i.

P3. The fragments are *complete*; i.e., the original non-fragmented document can be reconstructed from the collection of fragments.

Property P1 is motivated by space-efficiency to avoid node duplication. Property P2 specifies a desirable property to ensure that document nodes are contiguous in the sense that if node x is an ancestor of node y and they both are stored in the same fragment, then all the nodes along the path from x to y should also belong to that fragment. Property P3 is a necessary condition for correctness. These

three properties are rather simple and reasonable requirements in our fragmentation model; and they are indeed satisfied by the various strategies that have been proposed for fragmenting XML data (e.g., [56, 28]). The number of fragments generated by a fragmentation algorithm is typically controlled by a size parameter that determines the maximum number of data nodes per fragment.

Based on the above three properties, a fragment in general can consist of multiple partial subtrees as illustrated by the following example.

Example 5.3 As an example, consider the fragmented XML document in Figure 5.1(a) which is partitioned into seven fragments identified by f_1 to f_7. Observe that the fragments satisfy the three properties (i.e., disjoint, acyclic, and complete). In general, a fragment can consist of a forest of subtrees of nodes (e.g., f_1 contains two subtrees rooted at nodes f and e). Furthermore, a subtree in a fragment does not necessarily correspond to a complete subtree in the XML document (e.g., the subtree rooted at node b in f_2 is partitioned between fragments f_1 and f_2). □

In order to guarantee Property P3 (i.e., fragment completeness), it is necessary to maintain some additional header information for each fragment to enable the fragments to be "stitched" together to reconstruct the original XML document. Specifically, we need to maintain information about the "inter-fragment" edges (e.g., the edge between nodes a and b in Figure 5.1(a)). In addition to ensuring completeness, note that the header information associated with the fragments can also be exploited for query processing as it actually provides some partial structural information about the fragments and their relationships.

5.4.2 Fragment Header Information

In this section, we discuss three annotation schemes for representing fragment header information, namely, `Edge`, `Prefix`, and `Prefix+Level`. These schemes

Fragment ID	Edge
1	(2,1)
1	(2,1)
2	(7,1)
3	(4,1)
4	(7,2)
5	(6,1)
6	(7,2)
7	-

(a)

Fragment ID	Prefix
1	/a[1]/b[1]/f
1	/a[1]/b[1]/e
2	/a[1]/b[1]
3	/a[1]/m[2]/c[1]/k
4	/a[1]/m[2]/c[1]
5	/a[1]/m[2]/c[2]/k
6	/a[1]/m[2]/c[2]
7	/a[1]

(b)

maxHt
2
1
2
1
2
1
2
2

(c)

Figure 5.3: Fragment Header Information (a) `Edge` (b) `Prefix` (c) Additional column for `Prefix+Level`

have different space-performance tradeoffs.

`Edge` **Annotation.** The most straightforward approach to represent a fragment's header information is to record the identity of the parent node of each incoming inter-fragment edge into that fragment. Each node n can be uniquely assigned by an identifier (f_i, m_j), where f_i is the identifier of the fragment containing n, and m_j is the pre-order traversal order of node n within fragment f_i. For example, in Figure 5.1(a), there is one inter-fragment edge into fragment f_2 and so the header information for f_2 would record $(7, 1)$ as the parent node of node b. The complete fragment header information is shown in Figure 5.3(a). While this simple approach is clearly space-efficient and sufficient for completeness, the minimal information being maintained does not provide much useful knowledge of the structural relationships among the fragments that can be exploited for query evaluation (as verified by our experimental results).

`Prefix` **Annotation.** The second method is referred to as `Prefix`, stores more information about the path leading to the root node of each subtree in a fragment. Specifically, each fragment is associated with the following header information: (1) a unique identifier for the fragment; and (2) for each subtree (rooted at a data node

Fragment ID	Prefix	Prefix + Level
1	/f, //m	/f
2	/b, //m	/b
3	/k, //m	/k
4	/c, //m	/c
5	/k, //m	/k
6	/c, //m	/c
7	/a	–

Figure 5.4: Relevant Fragment-Query Node Information

d_i) in the fragment, its prefix given by $prefix(d_i)$.

Example 5.4 Figure 5.3(b) shows the collection of header information for the fragmented XML document in Figure 5.1(a). Each fragment f_i is uniquely identified by a positive integer value i; and each subtree (rooted at a node d_j) contained in f_i, it is represented by its prefix $prefix(d_j)$. Note that for convenience, we have used positional predicates in $prefix(d_j)$ to distinguish among distinct data paths that share the same sequence of element tag names; other means of achieving this purpose (e.g., assigning each node with a unique nodeID attribute value) can be used as well. □

`Prefix+Level` **Annotation.** The third method, `Prefix+Level`, is a simple extension of `Prefix`, that also additionally records $maxHt(r)$ for each subtree rooted at node r in a fragment. Figure 5.3(c) shows the header information representation based on `Prefix+Level` for the fragmented document in Figure 5.1(a). As we shall explain in Section 5.4.3, the additional precomputed information turns out to be very effective in improving query evaluation as it can avoid unnecessary computations.

5.4.3 Identifying Relevant Fragments

In order to determine whether or not there is a match of an input query q in a fragmented document d, it is necessary to determine if there exists a collection of matching data nodes in the fragments that could collectively form a match of q in d. Instead of detecting matches of all query nodes in every fragment, which is clearly inefficient, our goal is exploit the fragment header information to speed up the detection of matching nodes in the fragments by minimizing unnecessary matchings. As a simple illustration, consider again the fragmented document d and twig query q in Figure 5.1. We can conclude that there is no matching of the query subtree rooted at node b in fragment f_4 based on the fact that the subtrees in this fragment do not have the prefix $/a/b$.

Thus, to improve matching efficiency, our goal is to determine for each query node q_j, the set of "relevant" fragments that can potentially contain matchings of q_j. Informally, a fragment f_i is said to be *relevant* for a query node q_j (or equivalently, f_i is a relevant fragment for q_j) if based on the fragment header information, f_i contains some subtree that could contain a matching of $subquery(q_j)$.

For notational convenience, we use $\mathcal{R}$ to denote the set of all relevant fragment-query node pairs between a given fragmented document d and a twig query q; i.e., $(f_i, q_j) \in \mathcal{R}$ iff fragment f_i is relevant for query node q_j. In the following, we elaborate on how relevant fragment-query node pairs are identified for the three types of fragment header information.

`Edge` **Annotation.** The `Edge` annotation provides the least structural information among the three schemes and therefore results in the largest number of relevant fragment-query pairs in $\mathcal{R}$. Specifically, If q_j is both the root query node and a child-step, then $(f_i, q_j) \in \mathcal{R}$ if f_i contains the root data node. Otherwise, if q_j is a non-root query node or a descendant-step, then $(f_i, q_j) \in \mathcal{R}$ for every fragment f_i.

Example 5.5 Consider the example document d and query q in Figure 5.1. With the `Edge` annotation scheme, we have $(f_7, /a) \in \mathcal{R}$ and $(f_i, q_j) \in \mathcal{R} \ \ \forall\ i \in [1, 7]$, $\forall\ q_j$ in q except for root query node $/a$. □

`Prefix` **Annotation.** With the `Prefix` header annotation scheme, $(f_i, q_j) \in \mathcal{R}$ if there exists a subtree rooted at r in f_i such that $prefix(q_j)$ matches $prefix(r)$. For example, in Figure 5.1, since $prefix(/b)$ matches $prefix(b)$ in f_2, we have $(f_2, /b) \in \mathcal{R}$.

However, when q_j is a descendant-step, the relevance checking needs to be more elaborate. For example, although $prefix(//m)$ does not match $prefix(b)$ in f_2 (in Figure 5.1), it is incorrect to conclude that there can not be a matching of subquery(//m) in f_2. Indeed, $prefix(//m)$, which is given by $/a//m$, is equivalent to (/a/m $\cup$ /a//*//m); and it is clear that $/a//*$ matches $prefix(b)$ in f_2. To correctly capture both the cases of relevance matching, we define the *extended prefix* of a query node q_j, denoted by $eprefix(q_j)$, as follows:

$$eprefix(q_j) = \begin{cases} prefix(q_j) & \text{if } q_j \text{ is a child-step,} \\ prefix(q_k)//* & \text{if } q_j \text{ is a descendant-step \& } q_k \text{ is the parent node of } q_j, \\ //* & \text{otherwise.} \end{cases}$$

Therefore, $(f_i, q_j) \in \mathcal{R}$ iff there exists a subtree rooted at r in f_i such that $prefix(q_j)$ or $eprefix(q_j)$ matches $prefix(r)$.

`Prefix+Level` **Annotation.** With the additional maximum height information, the `Prefix+Level` annotation scheme provides a more precise definition of relevance. Specifically, $(f_i, q_j) \in \mathcal{R}$ iff there exists some subtree rooted at r in f_i such that (1) $prefix(q_j)$ or $eprefix(q_j)$ matches $prefix(r)$ and (2) $minHt(q_j) \leq maxHt(r)$.

Example 5.6 Consider again the fragment f_2 and query q in Figure 5.1. With

the `Prefix` annotation, f_2 is relevant for both query nodes $/b$ and $//m$. However, with the `Prefix+Level` annotation, f_2 is relevant only for query node $/b$. The reason that f_2 is not relevant for $//m$ is because $maxHt(b) = 2$ which is less than $minHt(q_m) = 3$. Figure 5.4 shows all the relevant query node matchings for query q in Figure 5.1 under both `Prefix` and `Prefix+Level` annotations. □

5.4.4 Scheduling Fragment Query Evaluations

To optimize the processing of the fragments, it is important to schedule the fragment evaluations so as to minimize the processing of unnecessary fragments (i.e., fragments whose evaluations could be skipped without affecting the query's result). In this section, we present five policies for scheduling fragment evaluations: the first two are the simplest and are query-independent, while the remaining three are query-dependent.

Topological Scheduling, T. This policy evaluates a fragment f_i before another fragment f_j if some node in f_i has an edge pointing to some node in f_j.

Reverse-Topological Scheduling, R. This is the reverse of topological scheduling, where fragment f_i is evaluated before fragment f_j if some node in f_j has an edge pointing to some node in f_i.

Most-Specific Scheduling, S. The intuition for this policy is that a fragment f_i is more likely to contain some query node matching than another fragment f_j if f_i's prefix is more "specific" than f_j's prefix in terms of matching some query node's prefix. This is captured by the *specificity* of a fragment f_i, denoted by $s(f_i)$, which is given by

$$s(f_i) = \max_{(f_i, q_j) \in \mathcal{R}} \{|prefix(q_j)|\}$$

where $|prefix(q_j)|$ denote the number of non-wildcard steps in $prefix(q_j)$. A fragment with a larger specificity value is processed earlier.

Maximal-Matching Scheduling, M. The intuition for this policy is that a fragment that contains more relevant subtrees has a higher chance of producing a matching. By giving priority to such fragments, the objective is to obtain a complete matching of all the query nodes early after matching the first few fragments that contain more relevant subtrees.

This notion is captured by the *maximal-matching metric* of a fragment f_i, denoted by $m(f_i)$, which is given by

$$m(f_i) = \sum_{(f_i, q_j) \in \mathcal{R}} |\{s_{i,k} \mid s_{i,k} \text{ is a subtree in } f_i, s_{i,k} \text{ is relevant for } q_j\}|$$

Fragments are processed in non-increasing maximal-matching values.

Most-Critical Scheduling, C. In contrast to the maximal-matching policy which is designed to efficiently process matching queries, the most-critical policy is optimized for non-matching queries by trying to process earlier "critical" query nodes that can be potentially matched only in very few fragments.

fragment f_i is said to be *potentially matching* for a query node q_j if f_i could contain a matching for q_j. More formally, f_i is potentially matching for q_j if f_i is relevant for an ancestor query node of q_j (including q_j itself).

Let $F(q_j)$ denote the set of fragments that can potentially contain a matching for query node q_j; and let $Q(f_i)$ denote the set of query nodes that can potentially be matched in fragment f_i. It follows from the definition of relevant matching that

$$\begin{aligned} F(q_j) &= \{f_i \mid \exists\, q_k,\ q_k \text{ is an ancestor of } q_j \text{ in } q, (f_i, q_k) \in \mathcal{R}\}, \text{ and} \\ Q(f_i) &= \{q_j \mid \exists\, q_k,\ q_k \text{ is an ancestor of } q_j \text{ in } q, (f_i, q_k) \in \mathcal{R}\} \end{aligned}$$

A query node q_j is defined to be *critical* if $|F(q_j)| \leq |F(q_k)|$ for each query node q_k in q. A fragment f_i is defined to be critical if there exists some critical query node in $Q(f_i)$. We define the *criticality of a critical fragment* f_i, denoted by $c(f_i)$, as follows:

$$c(f_i) = \frac{\sum_{q_j \in Q(f_i)} |F(q_j)|}{|Q(f_i)|}.$$

In this policy, critical fragments are processed before non-critical ones; and critical fragments are processed in non-descending order of their criticality values. Once it is detected that all relevant fragments for a query node have been evaluated without generating any matching for that query node, we can immediately conclude that the input query has no matching and can terminate the evaluation.

Example 5.7 Consider the document d and query q in Figure 5.1. In terms of `Prefix+Level` annotation, $\mathcal{R}q = \{(f_1, /f), (f_2, /b), (f_3, /k), (f_4, /c), (f_5, /k), (f_6, /c)\}$. For the most-specific scheduling, f_2, whose *specificity* is the smallest (i.e. 2), will be processed last. Clearly, the processing of f_2 will be skipped since a matching is found after processing the other fragments. For maximal-matching scheduling, since every fragment has the same *maximal-matching* value, the order of fragment evaluation is arbitrary. To illustrate the most-critical scheduling policy, let us replace the query node $//i$ with $//j$ so that the modified query now becomes a non-matching query. The ordering of the *criticality* values of the 6 relevant fragments is as follows: $f_1 = f_2 < f_5 = f_6 < f_3 = f_4$. We can see that after processing f_1 and f_2, $//j$ has no matching yet and none of the remaining fragments are relevant for $//j$; thus the query will not be matched. □

Discussion. The basic approach presented above implicitly assumes that all fragments are available before the scheduling and evaluation. However, due to the

transmission delay in the network, it may not be practical to wait for the arrival of all the fragments of a document before processing, therefore the approach should be generalized to process fragments in batches. Instead of waiting for all fragments to process, the generalized approach schedules and evaluates on every batch of say w fragments. Specifically, the processor starts to process once w fragments have arrived, and will start the next processing procedure once another w fragments arrive. We refer to w as the scheduling window size. A larger w results a better scheduling policy and enables more optimizations, while it may incur larger waiting time; a smaller w can reduce the waiting time, while it may limit the optimization opportunities due to the local scheduling and evaluations on a small number of fragments. The tradeoff of varying this parameter is explored in Section 5.5.

5.4.5 Evaluating Queries in Fragments

The processing of a data fragment entails the simultaneous matching of the set of subqueries relevant for that fragment. This requires the detection and maintenance of various matching data nodes as the data nodes in a fragment are parsed and processed. The matching of subqueries in fragments involves two challenges. Firstly, since the data subtrees in a fragment are not necessarily complete as different parts of a subtree might be distributed over several fragments, the matching algorithm for fragments needs to be generalized to handle partial matching of subqueries. Secondly, since the fragments are not necessarily evaluated in a "contiguous" manner, the presence of partial matchings in various fragments need to be maintained to enable the partial matchings to be "joined" to detect complete matchings. In the following, the proposed solutions for the above two challenges are elaborated in detail.

Matching subqueries in fragments

Given a certain fragment, we know the set of subqueries that are to be evaluated on it. There exists some algorithms such as XTrie[39] that can match a set of XPath queries on the XML document. However, given a query, these algorithm can only determine whether this query is matched; while to process queries on fragments additionally requires the processor to return the *maximum-matching* subqueries for non-matching queries. The *maximum-matching* subquery is defined as follows :

Definition 5.4.1 (Maximum-matching subquery) *Given a query q and a document d, suppose subquery(q_i) from the query q matches the document d, and there does not exist another subquery(q_j) in which subquery(q_i) is a partial tree of subquery(q_j) that also matches document d, then subquery(q_i) is called the maximum-matching subquery of query q.*

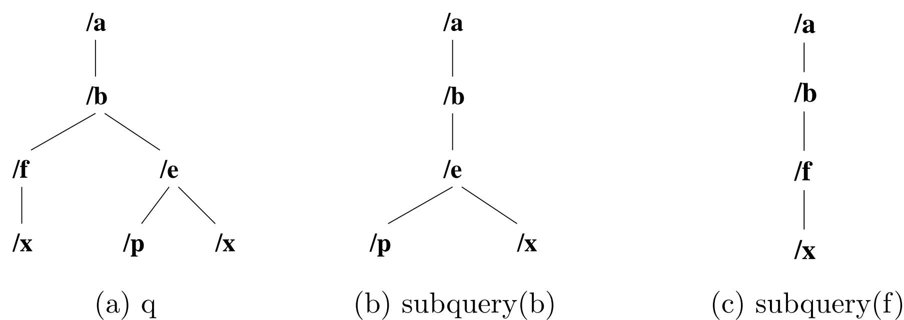


Figure 5.5: Example queries for maximum-matching

The reason for the requirement of *maximum matching* part is as follows. Considering the query q in Figure 5.5 (a) and the fragments in Figure 5.1(a) as an example. Subquery(b) will be evaluated in the fragment f_2, and there does not exist a complete matching of it in f_2. However, f_2 contains a partial matching of subquery(b), i.e. subquery(e) shown in Figure 5.5 (b). The matching of subquery(e) can be combined with the matching of subquery(f), shown in Figure 5.5

(c), in f_1 to form a complete matching of query q. These two matchings cause the matching of the whole pattern. If we do not return the partial matching of subquery(b) in f_2, we will miss to find the matching of whole pattern. We know that f_2 also contains the matching of subquery(x) and subquery(q) which are also the partial matching of subquery(b) in Figure 5.5 (a). However the matching of subquery(x) and subquery(q) will finally form the matching of subquery(b) in Figure 5.5 (b), thus there is no need to detect such matchings, which means that only the *maximum matching* part is necessary.

The subquery matching approach in this thesis is proposed by extending the XTrie algorithm from Chan et al[39]. Given a subquery subquery(q) to be matched on some fragment f, if subquery(q) is completely matched in f, the subquery matching algorithm reports the complete matching of subquery(q) in f; otherwise if subquery(q) is not matched in f, the subquery matching algorithm returns the maximum-matching subqueries for subquery(q) in f.

As mentioned in Section 2.3, in XTrie method, the twig queries are decomposed into substrings, and matchings for substrings are detected during the parsing of XML documents. The matchings of substrings are propagated bottom-up to detect larger partial matchings until the complete matchings are obtained. The partial matchings are maintained and updated during the parsing of documents. Specifically, when an event *start_element(n)* is encountered, the procedure to detect matching substrings is performed, and the matched substrings are recorded. Once a leaf substring is matched, the matching propagation is triggered in the bottom-up way until the largest partial matching (or the complete matching) is detected. When an event *end_element(n)* is encountered, the partial matchings involved with element n that are not be used in the following matching will be cleared.

Given a query node n in twig query q, a record in the form of $(k, b_0b_1 \ldots b_n, l)$ is used to store the matching status for the substring s_k where n is the last element in s_k. In the record, k is the identity of s_k in the ST-table; $b_0b_1 \ldots b_n$ is a bit vector, where b_0 =0 indicates the matching of s_k and $b_i = 0$ indicates the complete matching of subtree rooted at the ith child substring of s_k; and l specifies at which level such kind of matching is detected. It can be derived that if $b_0b_1 \ldots b_n$ are all 0, the complete matching of the subtree rooted at s_k is detected; if $b_0 = 0$ and there exist some $b_i = 0$, a partial matching of the subquery rooted at the ith child of s_k is obtained; otherwise, no partial matching or complete matching at s_k is detected.

Therefore, our algorithm to identify maximum-matching subquery is performed as follows. Once an event *end_element(n)* at level l is encountered, for each record $(k, b_0b_1 \ldots b_n, l)$:

1. If $b_0b_1 \ldots b_n$ are all 0, the complete matching of subtree rooted at s_k is propagated to the ancestor substring of s_k. Then the maximum-matching subquery should be generated in its ancestor substring, thus no maximum-matching subquery is detected.

2. If $b_0 = 0$ and $\exists b_i = 0$, then a maximum-matching subquery subquery(s_i) is obtained. Such maximum-matching subquery is recorded, and will be joined with maximum-matching subqueries from other fragments to detect the complete matching.

3. Otherwise, no maximum-matching subquery is generated.

Detecting complete matchings

In this section, we introduce the approach about how to maintain the matching results of subqueries from each fragment to determine the complete matchings.

The matching of subquery(q_i) means that the path from root to q_i and the subtree rooted at q_i is matched. With respect to q_j, which is q_i's nearest branch node, it means that the branch to q_i is already matched and it needs to be joined with the matching of other branches of q_j to generate the larger matching (if q_i does not have ancestor branch node, the matching of subquery(q_i) equals the matching of the whole query pattern). The identity of the data node that matches q_j in the matching of subquery(q_i) is used to specify the above matching. Specifically, the algorithm to detect the complete matchings using matched subqueries from fragments is performed as follows. Given a twig query q, let n_1, n_2, ..., n_k denote the k branch nodes in q. For each branch b_j of a branch node n_i, an array (denoted as $A_{n_i}^{b_j}$) is created to store these matching identities. Then when the branch b_j of n_i is matched, the identity of the data node that matches n_i is stored in array $A_{n_i}^{b_j}$. If each array $A_{n_i}^{b_j}$ of the branch node n_i contains a common identity, the subquery(n_i) is matched, and such matching is propagated to n_i's ancestor branch node. The complete matching is detected when the propagation reaches the root node of twig query q.

Summary of evaluating queries in fragments

The previous two sections present the solutions for the two main challenges of evaluating queries in fragments. This section gives a whole picture for query evaluations in fragments. The SAX based XML parser is used to parse the XML fragments. As aforementioned, a fragment is generally a forest of subtrees, thus a dummy node is created as the root of all subtrees, which makes each fragment to be a valid XML document.

For different subtrees in one fragment, different sets of subqueries will be evaluated on them. The subqueries are classified in terms of the subtrees to be evaluated

on. Then during parsing the fragment, once a new subtree of the dummy root is encountered, the corresponding set of subqueries should be provided for processing. The algorithm to match subqueries in fragments generates matching results for subqueries, i.e. all matched subqueries and maximum-matching subqueries for non-matching subqueries. The algorithm to detect complete algorithm will record these matching results, and use them to detect the complete matching. The detail algorithm for query evaluation in fragments are shown in Figure 5.6 and Figure 5.7.

Procedure QueryProcessing(q)
  Input: query q; $\mathcal{R}$; fragments $f_1, f_2, \ldots, f_n$
  Output: true or false

```
1:  for i = 1 : n do
2:    if ∃ (f_i, q_j) ∈ R then
3:      level = 0, CurEva = NULL;
4:      while ! END_OF_DOCUMENT do
5:        if START_ELEMENT t then
6:          level += 1;
7:          if level == 2 then
8:            if ∃ trie t associated with this subtree then
9:              CurEva = new Eva(t);
10:         if CurEva != NULL then
11:           CurEva.StartElement (name);
12:       else
13:         if level > 1 && CurEva != NULL then
14:           CurEva.EndElement (name);
15:         level = level - 1;
16:         if level == 1 then
17:           CurEva == NULL;
18:     for each matched subquery and each maximum partial matching do
19:       add the matched data node id to the list of its nearest branch node
          (denoted as q_k);
20:       if propagate(q_k, id) then
21:         return true;
22: return false;
```

Figure 5.6: Algorithm for query evaluation on fragments

The CurEva indicates whether there exists subqueries to be evaluated on the

```
Procedure: Propagate(q_k,id)
  curN = q_k curID = id;
  a_i is the list associated with branching i of curN;
  for i =1: n do
    if not exists k ∈ a_i where k == curID then
      return false;
  while curN is not branching node do
    curN = curN− >parent;
    if curN == Root then
      return true;
  curID = GetID(curN);
  if Propagate(curN, curID) == true then
    return true;
  else
    return false;
```

Figure 5.7: Algorithm for propagation

currently parsing subtree. When a new subtree begins, the algorithm checks whether the subquery set to be evaluated on the subtree is empty. If not empty, the algorithm creates an evaluator to perform the matching operations based on the trie structure during the parsing of the subtree and assign it to CurEva (line 8-9). For each encountered parsed event, i.e. *start_element(n)* or *end_element(n)*, if the CurEva is NULL, the algorithm does nothing; otherwise CurEva will perform the corresponding operations in the matching subquery algorithm (line 10-14). For the event *end_element(n)*, the level should be decreased by 1, and when the level is 1, the parsing of one subtree is finished, the CurEva is re-initialized to be NULL.

After parsing the fragment, for all matching subqueries and all maximum partial matchings, the processor calls the **propagate** procedure to check the larger matching (line 18-21). If the whole query pattern is matched, the processing will terminate and return true, otherwise after all relevant fragments are evaluated, the algorithm returns false.

Example 5.8 Consider the document d and query q in Figure 5.1 and assume that

`Prefix+Level` annotation is used. Suppose f_4 is first processed with the relevant subquery $subquery(/c)$. Since there is no complete matching of $subquery(/c)$ in f_4, the algorithm returns the partial matching $/c/s/y$, which means that $subquery(/s)$ is matched. Now, suppose the next fragment to be processed is f_3, which will result in $subquery(/k)$ being matched. From these two subquery matchings with different fragments, the algorithm will detect a matching of $subquery(/c)$. □

Processing multiple queries

This section discusses the additional extensions to process multiple queries simultaneously on fragmented XML data.

Firstly, we need to modify the algorithm to identify the relevant matchings. The naive approach is to handle the identification query by query. However, it is definitely not scalable to large number of queries, since the time to identify the relevant matchings increases as the number of queries increases. Different XPath queries for the same DTD are likely to share the common prefixes. Then the identification of relevant matchings for the various subqueries of different queries can be processed efficiently by exploiting the common prefixes among different queries. Given a set of XPath queries, we build a prefix sharing tree, denoted as T. Actually, T can be considered as a tree pattern, as defined in Section 5.2, with a special root node "/.". Each node n'_i in T is labeled with an element name or a wildcard that is prefixed by either "/" or "//". Additionally, each node n'_i is associated with an array $A(n'_i)$ to record the identities of the queries whose corresponding nodes are collapsed to n'_i. Similarly, prefix(n'_i) is defined as the path of nodes from the root of T to n'_i (root node "/." is not included).

Let q denote the query to be inserted, for each node n_i in q, if there exists a node n'_i in T such that prefix(n'_i) is the same with prefix(n_i), then q is just added

to the array of n'_i; otherwise, suppose n'_j is the node in T such that prefix(n'_j) is the longest prefix of prefix(n_i), we know that there exist a node n_j in prefix(n_i) that prefix(n_j) is the same with prefix(n'_j), then the subtree rooted at n_j in query q is appended to the node n'_j in T by collapsing n_j with n'_j.

Since T can also be considered as a tree pattern, the same relevant relationship of the nodes in T with each fragment can be defined as in Section 5.4.3. Suppose node n'_i in T is relevant with fragment f_i, then for each node n_i that in $A(n'_i)$, n_i is relevant with f_i. The following example is used to illustrate the approach.

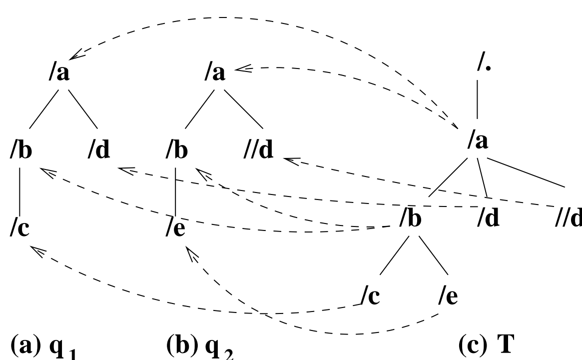


Figure 5.8: Tree patterns and their sharing prefix tree

Example 5.9 Figure 5.8 shows two tree pattern queries q_1, q_2 and prefix sharing tree T built for them. The dotted lines point to the nodes in queries that are collapsed to this node. We should notice that "/d" is not the same with "//d", thus they cannot be collapsed in T. Given a fragment with the header $/a/b$, by comparing T with $/a/b$, we know that query node b in T is relevant. Then we can find the corresponding nodes in the queries, and those nodes are relevant with this fragment. □

Secondly, for the fragment scheduling schemes, the scheduling metric values for scheduling policies S, M and C need to be generalized to consider the subqueries from all queries. The computation of metric values for these policies is addressed

in the following.

- **Most-Specific Scheduling, S**. For processing multiple queries, the fragment that may potentially match subqueries from larger number of queries should be processed early. Then the *specificity* of a fragment f_i is defined as

$$s(f_i) = \frac{\sum_{k=1}^{n} \max_{(f_i, q_j^k) \in \mathcal{R}} \{|prefix(q_j^k)|\}}{n}$$

where q_j^k is the query node from the kth query.

- **Maximal-Matching Scheduling, M**. The *maximal-matching metric* should be modified to consider the relevant subqueries from all queries. Then $m(f_i)$ is modified as follows :

$$m(f_i) = \frac{\sum_{k=1}^{n} \sum_{(f_i, q_j^k) \in \mathcal{R}} |\{s_{i,k} \mid s_{i,k} \text{ is a subtree in } f_i, s_{i,k} \text{ is relevant for } q_j^k\}|}{n}$$

- **Most-Critical Scheduling, C**. To support multiple queries, we first define

$$\begin{aligned} F(q_j^k) &= \{f_i \mid \exists\ q_i,\ q_i \text{ is an ancestor of } q_j \text{ in } q^k, (f_i, q_i^k) \in \mathcal{R}\}, \text{and} \\ Q(f_i) &= \{q_j^k \mid \exists\ q_i^k,\ q_i^k \text{ is an ancestor of } q_j^k \text{ in } q^k, (f_i, q_i^k) \in \mathcal{R}\} \end{aligned}$$

Then the *criticality* of a fragment f_i is modified as :

$$c(f_i) = \frac{\sum_{q_j^k \in Q(f_i)} |F(q_j^k)|}{|Q(f_i)|}$$

Finally, the scheduling metric value should be updated once a query is found to be matched or unmatched. Given a query, there may exist a set of fragments that are relevant. And by processing several of them, the matching result of the query may be obtained. Then the relevance information of this query should be eliminate

from the corresponding fragments, and the scheduling metric values should be updated. Therefore, to process multiple queries, once a matching result for a query is decided, we need to update the scheduling metric values, and such that the scheduling sequence of fragment for the following processing may be changed.

5.4.6 Dynamic Optimizations

This section presents two novel optimizations to further speed up the evaluation of twig queries on fragmented XML data by eliminating certain relevant evaluations. In contrast to the scheduling policies which are based on exploiting static properties of the query and fragment header information to "short-circuit" query processing, our new optimizations utilize dynamic information about the processed fragments to eliminate certain yet-to-be-processed relevant evaluations without affecting correctness. Specifically, the presence or absence of some node matching in a processed fragment can make some evaluation in a yet-to-be-evaluated relevant fragment *redundant* or *unnecessary*.

Eliminating Redundant Evaluations. This optimization is based on using the existence of some matching in a processed fragment to eliminate certain relevant evaluations in yet-to-be-processed fragments. Specifically, suppose there is a matching of $subquery(q_i)$, where q_j is the nearest ancestor branching node of q_i, such that q_j is matched to a data node d_j. Then, the evaluation of $subquery(q_k)$, where q_k is a descendant of q_i (or q_i itself), in a yet-to-be-evaluated fragment is considered *redundant* if this matching requires q_j (which is in the prefix of q_k) to be matched to d_j.

The implementation of dynamically eliminating redundant evaluations is as follows. As mentioned in query processing algorithm, when a matching of some subquery or a matching of some maximum partial matching is found, the procedure

propagate is called to generate larger matching part. Suppose the final obtained matching subquery by calling propagation is subquery(q_n) and q_m is its nearest ancestor branch node. Let id_m denotes the identity of the data node that matched q_m in the matching of subquery(q_n), then for any unprocessed relevant fragment-query pair (f_i, q_j) that can be determined as *redundant subquery* as defined in previous paragraph, we remove it from $\mathcal{R}$. The eliminating redundant evaluation optimization is applied at the end of each call for propagate.

Eliminating Unnecessary Evaluations. This optimization is based on using the absence of some matching in a processed fragment to eliminate certain relevant evaluations in yet-to-be-processed fragments. Specifically, consider a node q_i in a query q which matches a node d_i in a document D. If there exists a descendant query node q_j of q_i in q such that there is no matching of $subquery(q_j)$ in the subtree rooted at d_i, then it follows that there will no matching of $subquery(q_i)$ at d_i. Therefore, for each descendant query node q_k of q_i, the evaluation of subquery $subquery(q_k)$ in a fragment f_k is considered unnecessary if every subtree in f_k is a subtree of d_i.

The work in this thesis detects two kinds of unnecessary subqueries. Suppose $subquery(q_i)$ will be evaluated at a subtree s_{ij} in fragment f_i with root node s_i, and there exists a branch in q_i whose matching can not be found in s_{ij}, then if one of the following two conditions is satisfied, the eliminating unnecessary evaluations can be performed.

- If the unmatched branch will not be evaluated in any other descendant subtree of s_{ij}, then we know that the complete subtree rooted at s_i in the original document does not contain a matching of this branch, hence it cannot contain a matching of subquery(q_i). Therefore, for any subquery(q_j) in which q_j is some descendant node of q_i, if the matching of q_j will finally form a

matching of q_i with the matched root s_i, it is *unnecessary subquery* and can be eliminated before processing.

- Suppose q_k is the nearest branching ancestor node of q_i in the query pattern, let id_{ki} denote the id of the node which will be stored in the list associated with q_k if the matching of Q_i in subtree s_{ij} is found. From the prefixes in the header information, we can obtain the subtree that contains the node with id id_{ik} and the subtrees whose roots are some descendant of node id_{ik}, let S_i denote the set of these subtrees. For each subtree $s_{ir} \in S_i$, if either the unmatched branch will not be evaluated in s_{ir} or it does not matched in s_{ir}, we know that the complete subtree rooted at the node id_{ki} in original document does not contain a matching of this branch, hence it cannot contain a matching of q_k. Therefore, for any subquery(q_j) in which q_j is some descendant node of q_k, if the matching of q_j will finally form a matching of q_k with the matched root id_{ki}, the subquery is also unnecessary.

Example 5.10 Consider the fragmented document d and query q in Figure 5.1. After processing f_1, the evaluation of query node $/f$ at f_2 becomes redundant since there is already a complete matching of $subquery(/f)$ in f_1 under the same data node b in f_2. After processing f_6, the evaluation of query node $/k$ with f_5 becomes unnecessary since (1) there is no matching of $subquery(/s)$ in f_6 and (2) there are no other descendant fragments of f_6 (besides f_5) that could potentially provide a matching of $subquery(/s)$. Thus, even if there is a matching of $subquery(/k)$ in f_5, this will not yield a complete matching of $subquery(/c)$. □

5.5 Experimental Study

This section reports our experimental results. The experiments were designed to evaluate the efficiency of the query processing approach on fragmented data and the effectiveness of scheduling strategies and optimizations. Both synthetic and real-life datasets were used in experimental studies. The performance metric is the query processing time (in ms) which includes both the time to generate the relevant fragment-query pairs based on the query and fragment header information, as well as the time to schedule and evaluate the fragments. All experiments were conducted on a 3 GHz Intel Pentium IV machine with 1 GB of main memory running Windows XP; and all algorithms were implemented using C++.

5.5.1 Experimental Testbed and Methodology

Data Sets. Both synthetic and real-life XML data are used in the experiments. The synthetic data (denoted by D_{XMark}) is generated using the XMark benchmark [16], while the real-life data (denoted by D_{DBLP}) is obtained by extracting 11.5 MB of data from the large DBLP XML document [3].

These datasets are fragmented using Natix's algorithm [56] which controls the maximum number of data nodes in a fragment using a threshold t. Essentially, fragments are formed by traversing the document's data nodes in document order such that whenever the number of nodes visited exceeds t, the current node is used as a separator node to create a new fragment which comprises of all the subtrees below nodes that are left siblings of nodes along the path from the root node to the separator node. By setting t to 5000, we obtain 34 fragments for D_{XMark}; and by setting t to 10000, the document from DBLP is fragmented into 29 fragments (D_{DBLP}). D_{XMark} is used as the default dataset. The parameter values for D_{XMark}

Q_1	//open_auctions/open_auction/annotation//text
Q_2	/site/open_auctions/open_auction/annotation//text
Q_3	/site[//open_auctions/open_auction/annotation//text]/regions//namerica /item/mailbox/mail
Q_4	/site[//regions/europe/item][open_auctions/open_auction//text]/catgraph/edge
Q_5^-	//open_auctions/open_auction/annotation//capital
Q_6^-	/site[people/person/name]/regions[america]//eup//mailbox/mail
Q_7^-	/site[people/profile]/catgraph/from
Q_8^-	/site[open_auctions//annotations/text/captical]/regions//europe//mail
Q_9	//incollection//sup
Q_{10}	/dblp/incollection/title/sup
Q_{11}	/dblp[article//title]//incollection/title
Q_{12}^-	/dblp/article//title/name
Q_{13}^-	/dblp[//article/title]/incollection/name
Q_{14}^-	/dblp[phdthesis/title][incollection/title/sup]/mastersthesis/title/sub

Table 5.1: XPath queries on D_{XMark} and D_{DBLP}

Parameter	Values
Data size (MB) (Scaling factor)	**15 (0.1)**, 35 (0.3), 70 (0.6)
#Fragments	34
#Queries	10, 20, **40**, 80

(a)

Algorithm	Values
Scheduling	`R`, `T`, `S`, `M`, `C`
Header annotation	`E(Edge)`,`P(Prefix)` `PL(Prefix+Level)`
Dynamic optimizations	+(redundant) −(unnecessary)

(b)

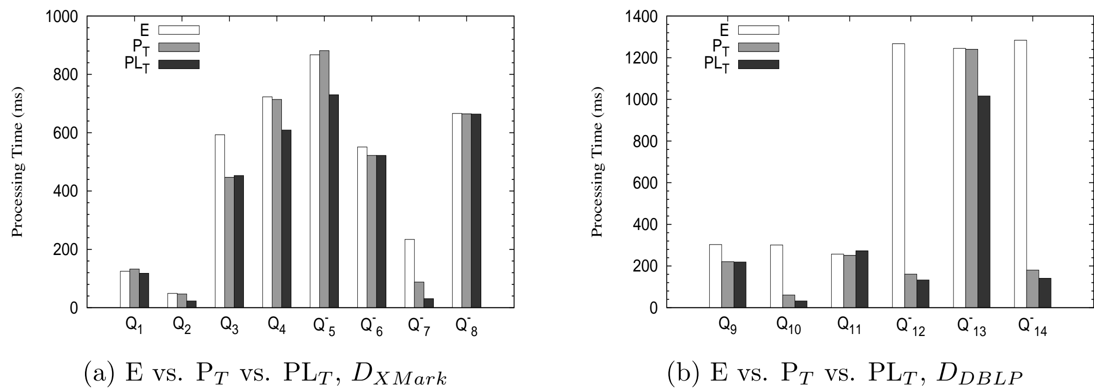
Table 5.2: parameters for D_{XMark} and algorithms

and their default values are shown in Table 5.2(a), where the value in bold is the default value used.

Queries. The XPath queries used in the experiments are listed in Table 5.1, in which the first eight queries are addressed on D_{XMark} and the last six queries are addressed on D_{DBLP}. Q_i is used to denote a matching query and Q_i^- is used to denote a non-matching query. To evaluate the approach for processing multiple queries, we used the XPath generator from the YFilter project [49] to generate a set of random queries.

Algorithms. Various fragmented approaches are compared by exploiting different scheduling strategies, header annotations and optimizations, as shown in Ta-

ble 5.2(b). At first, it is assumed that all fragments are available before the processing, then the scheduling can be performed among all fragments. For notational convenience, A_y^x is used to denote a fragmented approach, where $A \in \{\texttt{E}, \texttt{P}, \texttt{PL}\}$ represents the fragment header annotation scheme used; $y \in \{\texttt{R}, \texttt{T}, \texttt{S}, \texttt{M}, \texttt{C}\}$ (as denoted in Section 5.4.4) represents the fragment scheduling policy used; and x represents the set of dynamic optimizations used. For example, $PL_S^{+,-}$ denote the fragmented approach using the `Prefix+Level` annotation scheme, most-specific scheduling policy, and both dynamic optimizations. To illustrate the effect of scheduling strategies and optimizations, the results for a single query are shown firstly. After that the results for multiple queries are provided. Finally, the size of scheduling window is varied to show the effect on the scheduling strategies and by modelling the delay of each fragment as a percentage of the time to process the fragment, the effect of the delay is also illustrated. The default scheduling window size used is $w = \infty$ (i.e. scheduling is done with all fragments available).



(a) E vs. P_T vs. PL_T, D_{XMark} (b) E vs. P_T vs. PL_T, D_{DBLP}

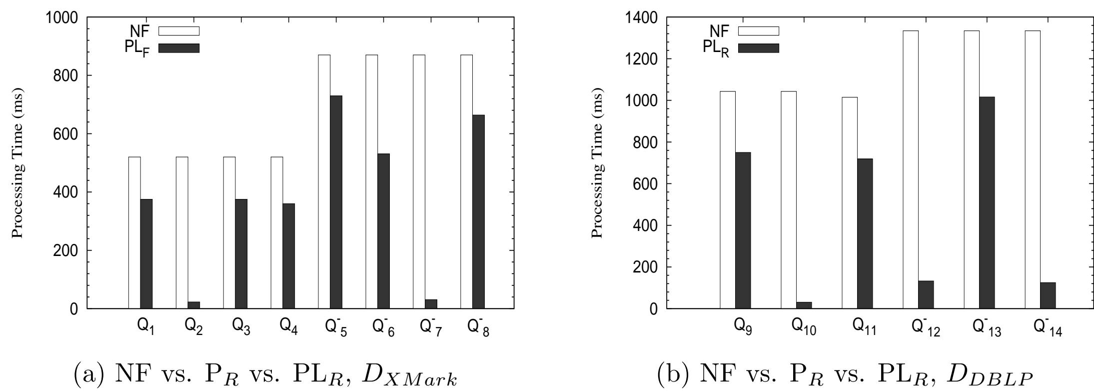
Figure 5.9: Comparison of fragmentation header schemas

5.5.2 Experimental Results

Comparison of header annotation schemes. Figure 5.9(a) and Figure 5.9(b) show the performance for different header annotations schemes for D_{XMark} and

D_{DBLP} respectively. As `Edge` annotation scheme has to process the fragments in topological sequence, thus we use `T` scheduling for `Prefix` and `Prefix+Level`. It shows that `Edge` achieves the worst performance. The reason is that the limited header information in `Edge` is not effective in pruning unnecessary relevant queries on fragments. For the two header annotation schemes that exploit the prefix information, PL_R is consistently more efficient than P_R since `Prefix+Level` annotation is able to exploit the additional $maxHt()$ information to prune off more non-relevant fragments.

By comparing the results in Figure 5.9(a) and Figure 5.9(b), we observe that the performance improvement of PL_R over P_R is more significant for D_{DBLP} than D_{XMark} because the former document is shallower than the latter, which means that there are more opportunities for $maxHt()$-based pruning in D_{DBLP} than in D_{XMark}. Given that `Prefix+Level`-based methods consistently outperforms `Edge` and `Prefix`-based methods, we will not include `Edge` and `Prefix`-based methods in subsequent experimental graphs.



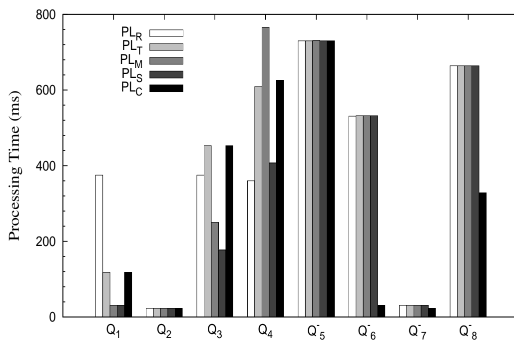
(a) NF vs. P_R vs. PL_R, D_{XMark} (b) NF vs. P_R vs. PL_R, D_{DBLP}

Figure 5.10: Comparison of fragmentation with non-fragmentation

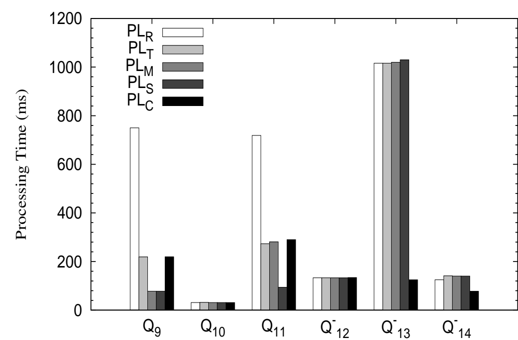
Fragmentation vs. Non-fragmentation. Figure 5.10(a) and Figure 5.10(b) compare the performance of the `NF` against the fragmented approaches using the default scheduling policy (i.e., PL_R) for various queries on D_{XMark} and D_{DBLP}

respectively. The results show that the approach PL_R outperformed NF for all queries; in particular, for Q_2, PL_R reduces the processing time of NF by 95%.

The performance improvement is due to the fact that fragmented approaches are able to process the fragments selectively based on relevant information. If the query is selective such that only a small number of fragments are relevant, PL_R is much better than NF. We record the number of fragmented processed by NF and PL_R, and it shows that for each tested query, PL_R processed a smaller set of relevant fragments. Actually, the time to identify relevant fragments using header annotations is trivial compared with the processing time, thus even for some case that PL_R has to processed each fragment, the performance of PL_R is still competitive with NF.



(a) Comparison of scheduling policies, D_{XMark}



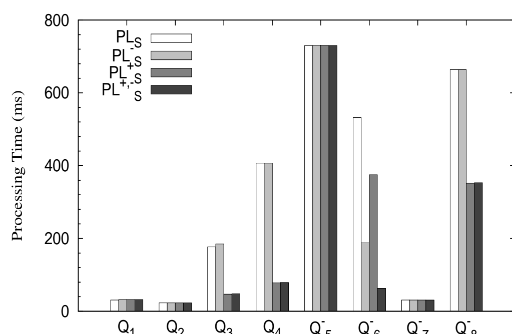
(b) Comparison of scheduling policies, D_{DBLP}

Figure 5.11: Comparison of scheduling policies

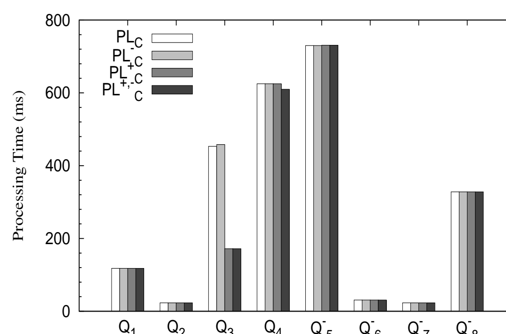
Comparison of scheduling policies. Figure 5.11(a) and Figure 5.11(b) demonstrate the performance of the five different scheduling policies (i.e. R, T, M, S, C) for various queries on D_{XMark} and D_{DBLP} respectively. The results show that in both datasets the S is generally the most competitive for matching queries (except for Q_4 on D_{XMark}); while C is generally the best policy for non-matching queries. As aforementioned, the proposal of policy S is to process the fragment that is the most promising to find a matching subquery, which causes the policy to achieve

good performance for matching queries. The policy `M` also intends to find matching queries earlier. However, the metric of `M` is not as precise as the metric of `S` to find matching queries.

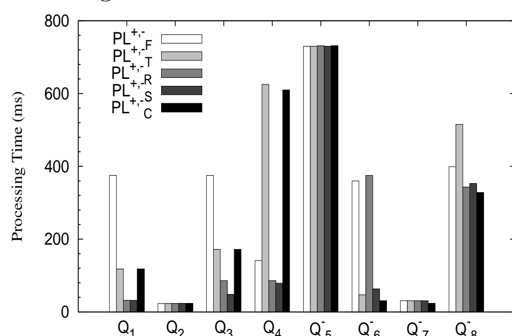
The reason for the relatively weaker performance of the `S` for Q_4 on D_{XMark} is that many redundant evaluations with high specificities for the two most specific branches in Q_4 delay the matching of the remaining branch. However, as shown later, when the redundant elimination optimization is also applied, the most-specific policy (i.e., PL_S^+) outperforms the other policies (including PL_R^+) for Q_4. As the dataset D_{DBLP} always shows similar trends with D_{XMark}, we ignore the results for D_{DBLP} in the following.



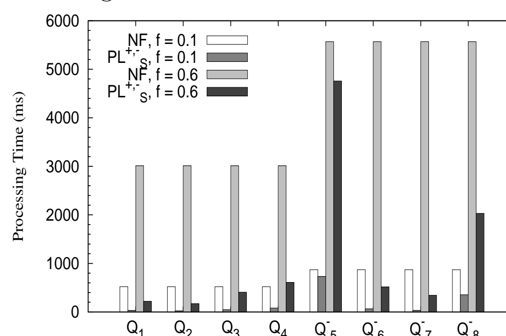
(a) Effect of optimizations on most-specific scheduling



(b) Effect of optimizations on most-critical scheduling



(c) Optimizations on different schedulings



(d) D_{XMark}, numFrag = 34

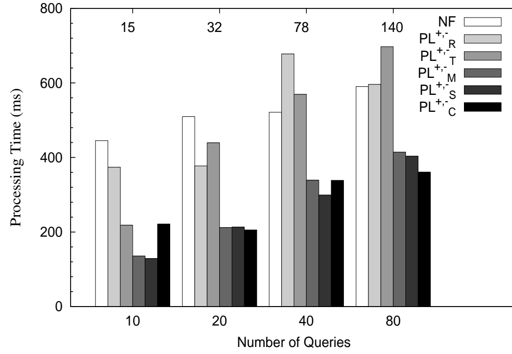
Figure 5.12: Effect of dynamic optimizations, document size, D_{XMark}

Effect of dynamic optimizations. Figure 5.12(a), Figure 5.12(b) and Figure 5.12(c) demonstrate the effect of the two dynamic optimizations for D_{XMark}.

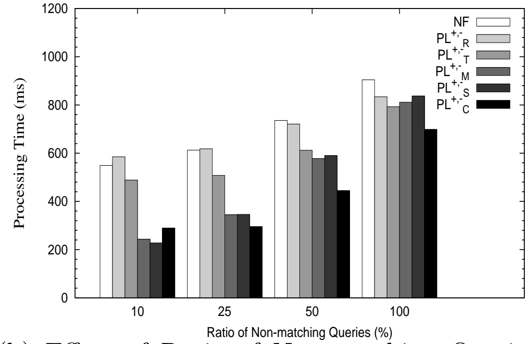
Figure 5.12(a) and Figure 5.12(b) consider the impact of the optimizations on the two best scheduling policies, respectively: S for matching queries and C for non-matching queries. For tree pattern queries that have more than one branch (e.g., Q_3, Q_4, Q_6^-, and Q_8^-), once a matching for a query branch is found, the redundant evaluation optimization can help to further improve performance such as Q_3, Q_4, Q_6^-; and once a matching for a query branch is guaranteed to be nonexist, the unnecessary evaluation optimization can help to further improve performance such as Q_6^-. We found that the redundant evaluation optimization is particularly effective for PL_S^+ since policy `S` is likely to find matchings early.

Our results also reveal that the unnecessary evaluation optimization is less significant than the redundant evaluation optimization due to the fact that the D_{XMark} data provides more opportunities for eliminating redundant evaluations. However, when both the dynamic optimizations are combined, the combination generally achieves the best performance. Figure 5.12(c) compares the effect of the combined optimizations with various scheduling policies on D_{XMark}. The results show that for matching queries, $PL_S^{+,-}$ offers the best performance, while for non-matching queries, $PL_C^{+,-}$ gives the best performance.

Effect of document size. Figure 5.12(d) compares the effect of document size on the performance of the non-fragmented approach (i.e., `NF`) and the fragmented approach $PL_S^{+,-}$ for D_{XMark}. We vary the parameter *scaling factor* in XMark to generate three different XML document with different size shown in Table 5.2(a). The distribution of elements in different XML documents is the same, and the number of fragments in each dataset is kept at 34. To avoid clutter, we only show the results for scaling factors of 0.1 and 0.6. As expected, query processing time increases with larger data size. Furthermore, the performance improvement of $PL_S^{+,-}$ over `NF` becomes more significant as data size increases.



(a) Effect of Number of Queries



(b) Effect of Ratio of Non-matching Queries, #Query = 40

Figure 5.13: Performance for multiple queries, D_{XMark}

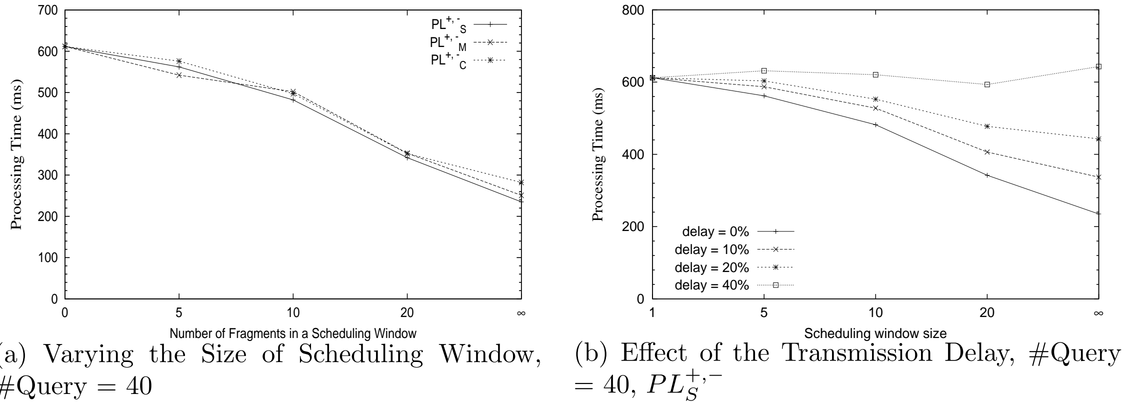
Effect of number of queries. Figure 5.13(a) demonstrates the performance of our approach for multiple queries by varying the number of queries from 10 (Q_{10}), 20 (Q_{20}), 40 (Q_{40}) to 80 (Q_{80}). The y-axis measures the average processing time, including both the time to determine relevant queries and the time to process each relevant fragment to determine the matching results, over all queries in the set. The numbers indicated above the bars represent the time (in ms) to process the fragment header annotations. For a single query, the time to find relevant queries is small enough to be ignored. As the number of queries grows, the time to process the fragment header annotations increases correspondingly. However, for the case Q_{80}, the time for processing header annotations still takes a small part of the whole processing time. For the dataset D_{XMark}, the number of subtrees in each fragment is large, which incurs a large number of prefixes to be compared with queries. Then for other datasets that the number of subtrees in each fragment is small, the time to process header annotations can be reduced.

Another component to affect the average processing time is the size of the fragments parsed in our approach. For Q_{10} and Q_{20}, our approach helps to skip parts of the document for all scheduling strategies, thus fragmentation outperforms the non-fragmentation. Among all $PL_S^{+,-}$ and $PL_M^{+,-}$ achieve the best performance,

since they help to find matching queries earlier; and $PL_C^{+,-}$ also achieves relatively good performance, since it helps to find non-matching queries earlier. As the number of queries increase, more fragments are likely to be relevant with more queries, thus the improvement of the fragmented approach over `NF` may diminish. For Q_{40}, $PL_R^{+,-}$ performs worse than `NF`, since each fragment is relevant and with the time to process the header information, $PL_R^{+,-}$ loses to `NF`. However, we observe that even for Q_{80}, $PL_S^{+,-}$ and $PL_C^{+,-}$ still outperform `NF`; this is due to their effectiveness in short-circuiting subquery evaluations and eliminating redundant/unnecessary evaluations.

Effect of the ratio of non-matching queries. Figure 5.13(b) shows the effect of increasing the ratio of non-matching queries for the default query set Q_{40}. For $PL_R^{+,-}$ and $PL_T^{+,-}$, which are independent of queries, the increasing of average processing time is slower than `NF`. For the case where all the queries are non-matching, each fragmentation approach outperforms `NF`. The reason is that for non-matching queries, `NF` has to scan the whole document, while fragmentation can selectively scan only relevant fragments. For $PL_S^{+,-}$ and $PL_M^{+,-}$, since the propose of these two strategies is to find matching queries earlier, the performance of these strategies decreases as the number of non-matching queries increases. However, they still outperform `NF` even when the ratio of non-matching queries reaches 100%. For $PL_C^{+,-}$, its performance improves as the number of non-matching queries increases since the goal of the most-critical scheduling is to find non-matching query earlier. The queries in the set are generated by choosing the probability of "//" as 0.1. If the queries contain less "//", which means that the queries are more selective, fragmentation can achieve better performance by skipping more irrelevant fragments.

Effect of scheduling window size and transmission delay. Figure 5.14(a)



(a) Varying the Size of Scheduling Window, #Query = 40

(b) Effect of the Transmission Delay, #Query = 40, $PL_S^{+,-}$

Figure 5.14: Effect of Scheduling Window Size and Transmission Delay, D_{XMark}

shows the effect of varying scheduling window size, $w \in \{1, 5, 10, 20, \infty\}$ using policies S C M. We observe that the query processing time generally improves as the scheduling window size w increases since scheduling and processing a larger batch of fragments enables more effective optimizations. Figure 5.14(a) also demonstrates that all three policies, i.e. S C M, show the similar trends.

We also vary the time delay t for w fragments to arrive to begin each batch of scheduling and processing. Here, the t is measured in terms of the percentage of time to parse one fragment, with $t \in \{0, 10, 20, 40\}$. As different scheduling strategies show similar trends when varying the scheduling window size, we only show the results for $PL_S^{+,-}$. Figure 5.14(b) demonstrates that the transmission delay t has larger effect when the scheduling window size is larger, since the arriving of all fragments in a window will take more time. As shown in Figure 5.14(b), as the delay t increases from 0% to 20%, the improvement from a larger w diminishes due to more time spent waiting for the fragments. However, when the delay becomes sufficiently large with $t = 40\%$, the performance for $w = \infty$ is actually worse than that for $w = 1$. This is because the benefit of processing a large batch of fragments is offset by long delay for the fragments to arrive.

5.6 Summary

This chapter introduced an approach for processing XPath boolean queries *directly* on fragmented XML documents without reconstructing the original documents. The proposed query processing strategy consists of three main steps:

(1) Identifying the relevant subqueries to evaluate on each fragment.

(2) Scheduling the order to evaluate the fragments.

(3) Optimizing each fragment evaluation to minimize unnecessary and redundant processing.

As part of the first step, three techniques to represent fragment header information are proposed to enable the effective identification of relevant subqueries for each fragment. For the second step, several fragment scheduling policies are proposed to enable the early determination of either a complete matching or a non-matching outcome. For the third step, two novel optimizations are introduced to maximize the elimination of redundant or unnecessary fragment-query evaluations. The experimental results based on both synthetic and real-life datasets demonstrate the effectiveness of the processing and optimization strategies with a performance improvement of up to a factor of 20 over the conventional approach of processing non-fragmented documents. Among the various fragment header annotation schemes, fragment scheduling policies, and evaluation optimizations, the $PL_S^{+,-}$ combination turns out to be the best approach for evaluating matching queries, while the $PL_C^{+,-}$ combination turns out to be the best approach for evaluating non-matching queries.

Chapter 6

Handling Heterogeneous XML Data

6.1 Introduction

Existing work on XML data dissemination (e.g., [20, 39, 71]) are all implicitly based on a *homogeneous schema* assumption where both the data published by different publishers as well as the users' subscriptions share the same schema. However, since the data publishers in a pub/sub system are autonomous and independent, they generally do not use the same schemas even when their published data are related and belong to the same domain (e.g., product catalogues). Consequently, if a user's subscription is based on the schema of a specific publisher (say P), then while the user can receive relevant documents from P that match his subscription, it is very likely that his subscription will not match relevant data from another publisher P' if the data schemas used by P and P' are different. Thus, the effectiveness of the pub/sub systems in pushing relevant data to consumers becomes diminished in the presence of heterogeneous data schemas.

This chapter addresses the problem of how to improve the effectiveness of XML data dissemination in the presence of heterogenous data schemas. Our problem, referred to as *heterogeneous data dissemination problem*, can be stated as follows. We

consider a pub/sub system where data published by different publishers are based on different schemas. The problem is how to effectively disseminate a document (based on some publisher's schema S) to relevant subscribers whose subscriptions might be based on schemas different from S. Previous work on content-based dissemination of XML data have focused on the special case where all the published data and users' subscriptions conform to a single schema.

For simplicity and without loss of generality, we assume that all the published data are of the same domain such that it is possible to integrate the different publishers' schemas (of the same domain) into a single global schema. Our problem and proposed techniques can be easily extended to the general case where each publisher is associated with multiple schemas corresponding to different domains: the general problem can be reduced to the simpler problem by first partitioning the collection of publishers' schemas into groups of schemas with similar domains, and then generating a global schema for each group of related schemas.

To better motivate our problem, we want to differentiate the *heterogeneous data dissemination problem* with the following two problems.

6.1.1 Data Integration Problem

Data integration problem focuses on how to query multiple data sources with different schemas. In contrast, the problem that we are addressing is on how to compare a published data against a collection of queries (i.e., subscriptions) to identify the matching queries given that the data and queries are based on different schemas. Thus, a fundamental difference between these two problems, which are related by the presence of schema heterogeneity, is that the integration problem belongs to a single-query-multiple-data scenario while the dissemination problem belongs to a single-data-multiple-queries scenario.

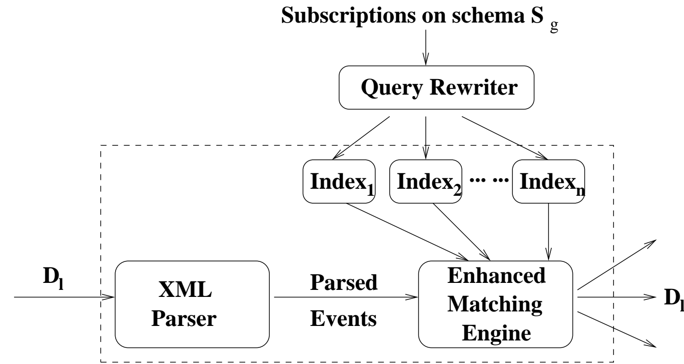


Figure 6.1: Query rewriting approach (QRA)

The central idea to address the data integration problem is *query rewriting*, which is performed as follows. To facilitate the querying over disparate data sources with different schemas, the approach taken is to first integrate the collection of local data schemas into a global schema which then serves as a schema for querying. Each query q against the global schema is processed by rewriting q into a collection of local subqueries against the local schemas and evaluating each local subquery at the appropriate data source.

Now, consider applying the query rewriting idea to solve the heterogeneous data dissemination problem as follows:

(S1) First, integrate the collection $\{S_1, S_2, \cdots, S_n\}$ of different publishers' schemas (that are based on the same domain) into a global schema S_g. The global schema S_g is then made available to users to specify their subscriptions.

(S2) Next, each "global" subscription q_g (which is based on global schema S_g) is then rewritten into a set of local subscriptions $\{q_1,q_2,\cdots,q_n\}$, where each q_i is based on a local schema S_i. To enable efficient matching a published data (conforming to some local schema S_i, $i \in [1, n]$) against local subscriptions based on S_i, an index I_j is constructed for each collection of subscriptions based on local schema S_j, $j \in [1, n]$.

The above approach, which we refer to as the *query rewriting approach (QRA)* is illustrated in Figure 6.1. For each incoming data D (based on some local schema S_j), the matching engine only needs to compare D against the appropriate set of local subscriptions via index I_j.

The query rewriting approach, however, suffers from three drawbacks.

- The scalability of the approach is limited as each input subscription needs to be rewritten into one subscription for each local schema. This increases the space overhead for storing and indexing the expanded set of local subscriptions at each router. Note that although the input global subscriptions are not used directly for document matching, these subscriptions still need to be maintained for generating new rewritings whenever a new local publisher schema is added (or changed).

- The approach also incurs a high update cost. Whenever a new data schema S' is introduced (by an existing or new publisher), it is necessary to generate and install new subscriptions (for schema S') at each router by rewriting the global subscriptions registered at each router to corresponding local subscriptions on the new schema S'.

- The query rewriting approach is an *intrusive* approach, in the sense that it requires making changes to the conventional routing approach; specifically, the matching engine needs to be enhanced to recognize some additional metadata about the document's schema and to support multiple indexes on the schema-specific collections of subscriptions.

6.1.2 Query Relaxation Problem

Another direction taken to address the problem of schema heterogeneity is to apply *query relaxation* techniques (e.g., [105, 22, 81, 21]). This can be viewed as a *schema-independent query rewriting* approach where a query is "relaxed" to multiple queries without relying on knowledge of data schemas but based on making local structural changes to parts of the query. The motivation for this line of work is to enable retrieval of approximate answers to a query and it is often used in combination with some ranking and pruning mechanism during query evaluation at run-time to control the number of relaxed queries generated. However, it is unclear how this technique can be effectively applied to the context of the data dissemination problem since the number of queries registered at each router is large which makes run-time relaxation of a large set of queries a challenging problem. Alternatively, another possibility is to try to precompute the relaxed queries offline; but in the absence of the run-time data, it is unclear how the relaxed queries can be generated efficiently and in a controlled manner without a large set of relaxed queries being produced.

This chapter presents a novel paradigm to solve the heterogeneous data dissemination problem that is based on the principle of *data rewriting*. The new approach is referred as *DRA* for data rewriting approach. The conceptual idea of DRA is as follows. As in step (S1) of QRA, the collection of local schemas from the publishers is first integrated to form a global schema S_g which is then made available to users to specify their subscriptions. Unlike QRA, our DRA does not require query rewriting which means that only the input global subscriptions are indexed at each router. For each incoming data D_ℓ (conforming to some publisher's local schema S_ℓ) to a router, our DRA first rewrites D_ℓ to D_g (which conforms to the global schema S_g) and then match D_g against the registered global subscriptions

in that router. In contrast to QRA, our proposed DRA is more effective for the heterogeneous data dissemination problem because pub/sub systems are typically characterized by the following two properties:

1. the number of subscriptions at each router is large (which limits the scalability of QRA); and
2. the data being disseminated is relatively small (which incurs only a small processing overhead for data rewriting).

Thus, the propose DRA has three key advantages:

1. It is space-efficient as it only stores the registered global subscriptions (unlike QRA which stores an expanded set of rewritten queries for each local schema).
2. It is also update-efficient as additions and changes to local schemas do not require updating of registered queries at the routers.
3. It is also time-efficient as the overhead of data rewriting is low and the matching of the document against a (non-expanded) set of queries is fast.

The rest of this chapter is organized as follows. Section 6.2 presents our novel data rewriting framework. We discuss implementation issues for the various approaches in Section 6.3. The experimental results are presented in Section 6.4. We conclude this chapter in Section 6.5.

6.2 Data Rewriting Framework

This section presents the new framework to solve the heterogeneous data dissemination problem by using *data rewriting* techniques.

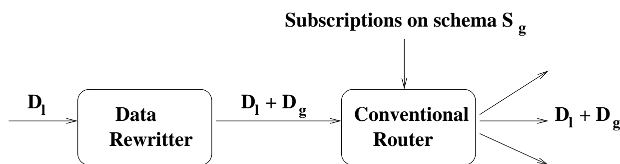
Firstly, the system architecture is presented in Section 6.2.1 and then three main approaches to employ data rewriting are discussed in Section 6.2.2. After that, Section 6.2.3 describes details of schema mappings used in our prototype implementation. It is important to emphasize that our data rewriting framework is orthogonal to the specific techniques for schema integration and mapping in Section 6.2.3 and can be combined with other techniques as well. Finally, Section 6.2.4 introduces the set of data rewriting operators and Section 6.2.5 presents the mechanisms to derive those operators.

6.2.1 System Architecture

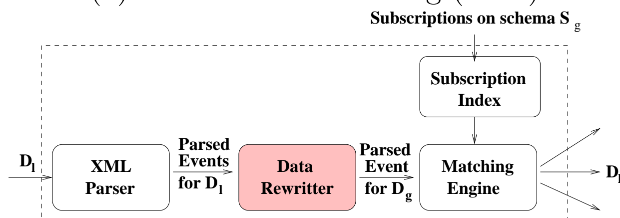
Similar to existing pub/sub systems, the system based on data rewriting has a *mediator agent (MA)* that serves as a coordinator between the data publishers and publishers [50, 23]. Besides collecting schemas from publishers and registering queries for users, the MA is also responsible for integrating the local schemas from the various data publishers to generate a global schema for each data domain.

We use S_ℓ to denote some publisher's local schema, and S_g to denote a global schema integrated from a collection of local schemas of the same domain. We use D_ℓ (resp., D_g) to denote a document conforming to schema S_ℓ (resp. S_g).

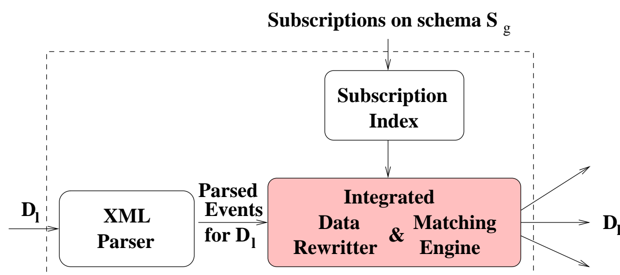
As part of the schema integration process, the MA also creates a *schema mapping*, denoted by $M_{\ell,g}$, for each local schema S_ℓ that is integrated to a global schema S_g. A schema mapping $M_{\ell,g}$ is essentially a data transformation specification that enables an input document D_ℓ to be mapped into an output document D_g that preserves the appropriate information content of D_ℓ. The details of schema mappings used in this work are discussed in Section 6.2.3.



(a) Static Data Rewriting (SDR)



(b) Non-Intrusive Dynamic Data Rewriting (NDDR)



(c) Intrusive Dynamic Data Rewriting (IDDR)

Figure 6.2: Data Rewriting Approaches

6.2.2 Data Rewriting Approaches

This section gives an overview of three main approaches to perform data rewriting. These approaches can be classified based on where the rewriting is done; specifically, referring to the architecture of a typical content-based router in Figure 2.3, the data rewriting step is introduced in three possible locations as illustrated in Figure 6.2: (a) outside of the router, (b) between the parser and matching engine components of the router, and (c) within the matching engine component. These three architectural options have different implementation-performance tradeoffs. In terms of implementation complexity, the last method is an *intrusive* approach in that it requires making substantial changes to some software component: in this

case, the data rewriting step needs to be integrated into the matching engine. In contrast, the first two approaches are *non-intrusive* and have a lower implementation effort.

The different methods can also be classified into *static* or *dynamic* approaches depending on whether the data is rewritten only once before being forwarded to the routers (e.g., the first approach) or rewritten dynamically by each router (e.g., last two approaches).

It is important to note that similar to the conventional approach and QRA, the data rewriting approaches also deliver the original document D_ℓ (and not D_g) from the publishers to the users. The purpose of rewriting D_ℓ to D_g is to enable the document to be matched against the global subscriptions.

Static Data Rewriting (SDR). In the *static data rewriting (SDR)* approach (illustrated in Figure 6.2(a)), each published data D_ℓ (based on some publisher's local schema S_ℓ) is rewritten to D_g (based on the integrated global schema S_g) statically (but only once) by either the publisher itself or the MA. The advantage of employing the MA to rewrite the data is that the publishers are shielded from the details of the schema mappings and rewriting processing; however, this requires each publisher to first forward D_ℓ to the MA for the rewriting before the MA forwards the transformed data to the routers for dissemination.

Once D_ℓ has been rewritten to D_g, both D_ℓ and D_g are forwarded together to the network of routers for dissemination. Here, D_g serves as metadata to enable the forwarding of the payload data D_ℓ. Since the subscriptions stored in each router are based on the global schema S_g, D_g is used for matching against the subscriptions to detect matching subscriptions and decide to which router(s) the data needs to forwarded next. For forwarding to the local subscribers at a router, only the actual data D_ℓ needs to forwarded.

One advantage of SDR is that it is a non-intrusive approach that can be easily implemented. However, the tradeoff is that the amount of data that is being forwarded is roughly doubled compared to the conventional approach.

Dynamic Data Rewriting (DDR). To avoid the transmission overhead of SDR, an alternative strategy is for each router to forward only D_ℓ but the tradeoff is that each router now needs to rewrite the data D_ℓ *dynamically.* This approach is referred to as *dynamic data rewriting (DDR)* approach. Note that DDR does not materialize D_g. Instead, the data rewriting is conducted during the parsing of documents for query evaluation. Specifically, the XML parser still parses D_ℓ, while the evaluation of queries is equivalent to match these queries against D_g.

Two dynamic data rewriting approaches are proposed based on the location to perform data rewriting.

- **NDDR** The first option is to perform the rewrite outside of the matching engine by installing a new software component, called the *data rewriter*, between the document parser and matching engine as shown in Figure 6.2(b). The data rewriter essentially rewrites D_ℓ to D_g by intercepting the sequence of events E_ℓ that is generated by the event-based XML parser (as it parses the input document D_ℓ) and generating a modified sequence of events E_g to the matching engine such that E_g is equivalent to the sequence of events generated by parsing D_g. We refer to this approach as *non-intrusive dynamic data rewriting (NDDR)* approach since it does not require making any changes to the existing XML parser and matching engine components. The challenge of NDDR is how to intercept the sequence of events in *data rewriter*, which is elaborated in Section 6.3.1.

- **IDDR** The second option is to rewrite the data within the matching engine itself as shown in Figure 6.2(c); we refer to this approach as *intrusive dynamic*

data rewriting (IDDR) approach. The challenge of IDDR is how to match queries against the parsed events that are encountered out of document order. This is to be elaborated in Section 6.3.2.

In order for a router R to perform data rewriting, R needs to have access to the schema mappings generated by the MA. There are two possible options for routers to access the schema mappings. The first option is for each router to keep a copy of all the schema mappings generated by the MA. Thus, the MA needs to disseminate its generated schema mappings to all the routers during an initialization process.

The second option is for each published data D_ℓ to be disseminated along with its appropriate schema mapping $M_{\ell,g}$ as part of the data's header information. The additional header information can be inserted into the published data either by the publisher itself or by the MA. In the first case, the MA needs to disseminate relevant schema mappings to each publisher as part of an initialization process so that the publisher can insert the appropriate schema mapping for each published data. In the second case, each publisher simply disseminates its published data D_ℓ via the MA, and the MA becomes responsible for inserting the appropriate schema mapping into the data's header before sending it out for dissemination.

Comparing the two options of accessing schema mappings, the first option is less space-efficient since the schema mapping information is replicated in every router; consequently, the first option is also more costly to maintain when updates arise (e.g., when a new data schema is introduced by a new or existing publisher). In contrast, by not replicating the schema mapping information, the second option is more space- and update-efficient at the cost of a slightly higher transmission overhead.

To distinguish between the options of storing all the schema mappings at the routers or disseminating the data with an appropriate schema mapping, we use

$NDDR_{router}$ (resp. $IDDR_{router}$) to denote $NDDR$ (resp. $IDDR$) combined with the first option, and $NDDR_{data}$ (resp. $IDDR_{data}$) to denote $NDDR$ (resp. $IDDR$) combined with the second option.

Alternative Approaches. This section discusses yet another approach to avoid the transmission overhead of SDR. Unlike the DDR methods which disseminate only D_ℓ, another approach is to disseminate only D_g; we refer to this approach as *dynamic reverse rewriting (DRR)* approach. In DRR, the published data D_ℓ is rewritten to D_g statically only once (similar to SDR) and only D_g is forwarded to the routers for dissemination. While this approach does not incur the dynamic rewriting overhead of DDR during the router-to-router forwarding, it will require a *reverse rewriting* to transform D_g back to D_ℓ when forwarding the data to matching subscribers. There are two options of when the reverse rewriting is performed at some router R. The first option is to do it when the first matching subscriber is detected at R in which case the subscription matching process is suspended and D_g is rewritten to D_ℓ (using some reverse mapping operators). This approach is denoted as EDRR. Note that the reconstructed D_ℓ is only an approximate form of the original published data (e.g., ordering of sibling elements is not preserved). The subscription matching then resumes once the converted D_ℓ is forwarded to the matching subscriber. The second option is to hold back the reverse rewriting until after D_g has been completed parsed and forwarded to all matching neighboring routers. This approach is denoted as LDRR. The tradeoff between these options is that the first option speeds up the forwarding to matching subscribers but delays the forwarding to matching outgoing routers, while the reverse holds for the second option. However, our experimental results show that the additional complexity of reverse rewriting is not a good idea as both the DRR methods are outperformed by the three main approaches.

6.2.3 Schema Mapping

This section describes the schema mapping specification used in our work. A schema mapping, denoted by $M_{\ell,g}$, is a specification that enables an input document D_ℓ (that conforms to a source schema S_ℓ) to be transformed to an output document (that conforms to a target schema S_ℓ) such that the appropriate information content of D_ℓ is preserved in D_g. Each schema mapping can be generated as part of the schema integration process. The work in this thesis adopts a simple schema mapping specification that consists of a tree representation of the source schema (i.e. local schema) annotated with data rewriting operators.

It is important to emphasize that the focus of our work is on using a *data rewriting approach* to solve the heterogeneous data dissemination problem and not on *schema mapping* per se. Thus, we have decided on a schema mapping specification that is reasonably expressive that supports a variety of data transformations and that is also amenable to an efficient implementation. Note that our proposed data rewriting paradigm is orthogonal to the actual choice of schema mapping specification and implementation.

An XML schema is modeled using a tree structure, called a *schema tree*, where tree nodes represent element types and tree edges represent element-subelement relationships. Each node tree is optionally associated with a symbol (?, *, or +) that represents the cardinality of the element that it represents. For simplicity, we do not consider the union and recursion constructs in our schema model. Note that even though a XML schema typically has common substructures and can be more concisely modeled as a graph, it is more convenient to duplicate the common substructures to model the schema as a tree [78] as this makes it easy to specify different transformation operations to different instances of the same substructure. An example schema tree for S_g is shown in Figure 6.3(a).

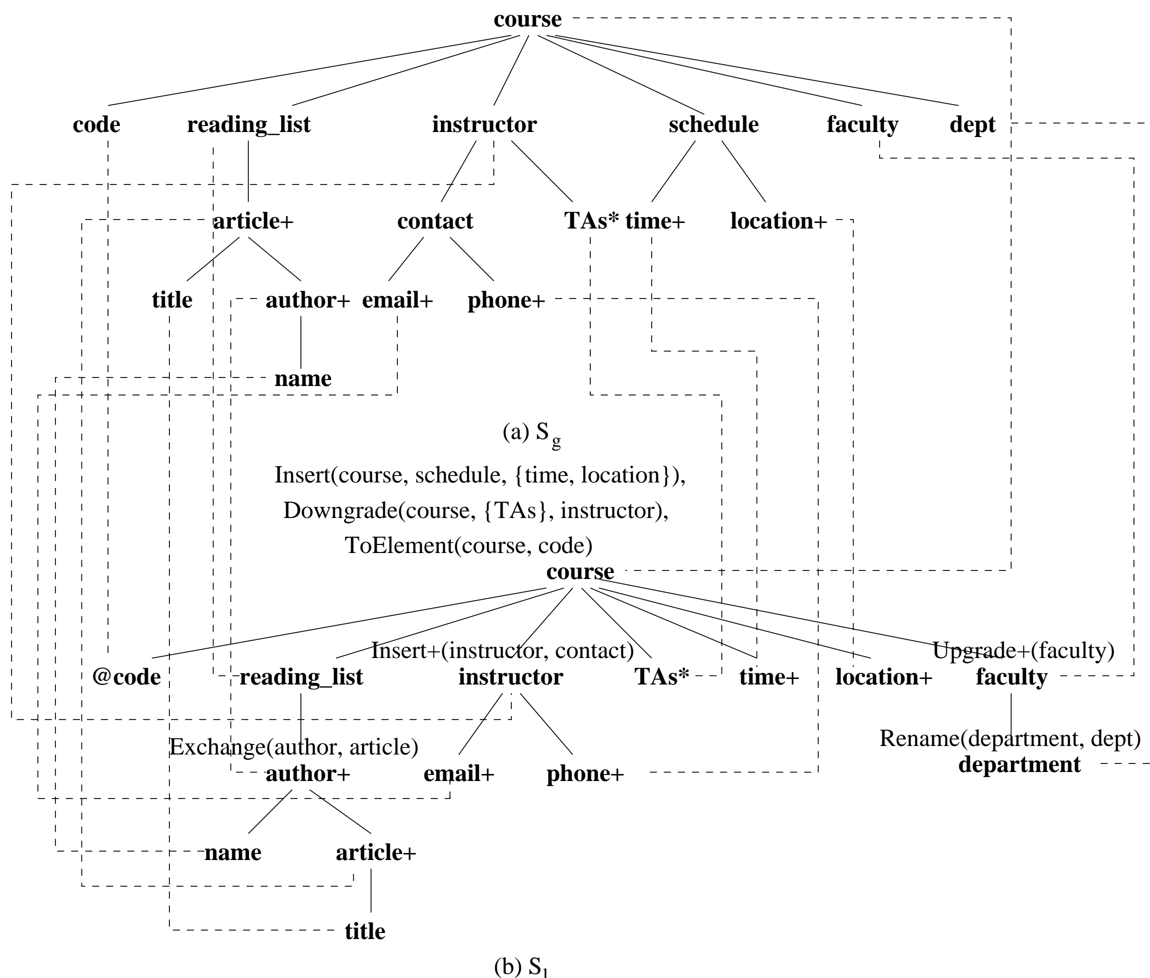


Figure 6.3: Example Schema Mapping $M_{\ell,g}$

We represent a *schema mapping* $M_{\ell,g}$ by an annotated schema tree of S_ℓ. Each node in the schema tree is annotated with a (possibly empty) sequence of *data rewriting operators* (to be discussed shortly). With this schema mapping, we can transform an input data D_ℓ (conforming to S_ℓ) to a data D_g (conforming to S_g that preserves the information contents of D_ℓ) by traversing each element e in D_ℓ (in document order) and applying the sequence of rewriting operations associated with element e in the annotated schema tree.

Given S_ℓ and S_g, $M_{\ell,g}$ can be computed from the following two steps:

1. First, compute a *schema matching* from S_ℓ to S_g using an existing method

(e.g., [82]). The schema matching essentially specifies a 1-to-1 mapping between the elements of S_ℓ and S_g. An example of a schema matching between S_ℓ and S_g is shown in Figure 6.3 (ignore the operator annotations for now) where the 1-to-1 mappings are indicated by the dotted lines.

2. Next, using the computed schema matching and the available set of data rewriting operators, annotate each element type e in S_ℓ with an appropriate sequence of operators to achieve the associated schema matching for e.

Figure 6.3(b) shows an example of a schema mapping $M_{\ell,g}$, where elements `author`, `instructor`, `faculty`, and `department` each has one rewrite operator annotation; while element `course` is annotated with a sequence of three rewrite operators.

6.2.4 Data Rewriting Operators

This section presents six basic data rewriting operators that can express a wide variety of data transformations. The example schema mapping $M_{\ell,g}$ shown in Figure 6.3 will be used as our running example to illustrate the operators. We use E, E', or E_i to denote an element type, and $child(E)$ to denote the set of child subelement types of an element type E.

Rename(E, E'). This operator renames E to E'. In Figure 6.3, the operator `Rename` (department, dept) is applied in S_ℓ to rename the `department` element to `dept` in S_g.

ToElement(E, A). This operator converts the attribute A of E to become a subelement of E such that the value of A becomes the contents of the new element A. This kind of heterogeneity is mentioned in [114]. In Figure 6.3, the `code` attribute of `course` element in S_ℓ is converted to become a new subelement named `code` of element `course` in S_g.

Insert$(E, E_1/E_2/\dots/E_k,\ S)$**,** $S \subseteq \mathit{child}(E)$**.** This operator first moves each child subelement E' of E, where $E' \in S$, to become a child subelement of E_k, where $E_1/E_2/\dots/E_k$ is a new path of elements. The entire subtree rooted at E_1 is then inserted to become a child subtree of E. This is used to resolve the generalization conflicts as mentioned in [104, 114]. `Insert`$^+(E, E_1\ /E_2/\ \dots/E_k)$ is a special case of the `Insert` operator that is equivalent to `Insert`$(E, E_1/E_2/\dots/E_k, \mathit{child}(E))$. In Figure 6.3, the operator `Insert`(course, schedule, {time, schedule}) is applied in S_ℓ to effectively group both the `time` and `location` subelements of `course` to become subelements of a new `schedule` element which is inserted as a new subelement of `course`. As another example in Figure 6.3, the operator `Insert`$^+$(instructor, contact) is applied in S_ℓ to group all the child subelements of the `instructor` element to become subelements of a new `contact` element which is inserted as a new subelement of `instructor`.

Upgrade(E, S)**,** $S \subseteq \mathit{child}(E)$**.** This operator "upgrades" each child subelement E_i of E (together with the subtree rooted at E_i), where $E_i \in S$, to become a sibling of E. `Upgrade`$^+(E)$ is a special case of the `Upgrade` operator that is equivalent to `Upgrade`$(E, \mathit{child}(E))$. In Figure 6.3, the operator `Upgrade`$^+$(faculty) is applied in S_ℓ to move each child subelement of `faculty` (only `department` element in this example) to become a sibling element of `faculty`.

Downgrade(E, S, E')**,** $S \subseteq \mathit{child}(E)$**,** $E' \in \mathit{child}(E) - S$**.** This operator "downgrades" each child subelement E_i of E (together with the subtree rooted at E_i), where $E_i \in S$, to become a child subelement of E'. In Figure 6.3, the operator `Downgrade`(course,{TAs},instructor) is applied in S_ℓ to move the `TA` subelement of `course` to become a child subelement of `instructor` (which is a child subelement of `course`).

Exchange(E, E')**,** $E' \in \mathit{child}(E)$**.** This operator swaps the roles of E and E' so

that E becomes a child subelement of E'. More specifically, the subtree rooted at E (excluding the subtree rooted at E') becomes a new child subtree of E'; and the parent element of E becomes the parent element of E'. This operator is used to resolve the parent and child conflicts in the schema definition as mentioned in [43, 114]. In Figure 6.3, the operator `Exchange`(author,article) is applied in S_ℓ to swap their parent-child roles so that `author` becomes a subelement of `article`. Note that the `Exchange`(E, E') operator can result in the data subtree rooted at E (excluding the subtree rooted at E') to be duplicated multiple times when rewriting data. This effect is illustrated in Figure 6.4 where the operator Exchange(article,author) is applied to rewrite D_ℓ to D_g. Observe that the title information for the first article (i.e., "XML") appears twice in D_g due to the fact the article has two authors. Note, however, that while the author named "John" has authored two articles in D_ℓ, the rewritten D_g does not merge the information from the first and third `author` subtrees into a single subtree. This is because the `Exchange` operator is not a "group by" operator; the latter is more a complex operator that requires some notion of keys for grouping information which is outside the scope of our work.

(a) $\mathbf{D}_1$

(b) $\mathbf{D}_g$

Figure 6.4: Rewriting D_ℓ to D_g with Exchange(article,author)

6.2.5 Deriving Data Rewriting Operators

Given S_ℓ and S_g, $M_{\ell,g}$ can be computed in two steps. Firstly, a *schema matching* is computed from S_ℓ to S_g using some existing method (e.g., [82]). The schema matching essentially specifies a 1-to-1 mapping between the elements of S_ℓ and S_g. An example of a schema matching between S_ℓ and S_g is shown in Figure 6.3 where the 1-to-1 mappings are indicated by the dotted lines. Next, the rewriting operations annotated with each element e in S_ℓ (denoted by $op(e)$) is computed using the computed schema matching and the following six rules.

Given an element e in S_ℓ, we use $par(e)$ to denote the parent of e in S_ℓ, and $map(e)$ to denote the mapped element of e in S_g.

Rename Rule. If the labels of e and $map(e)$ are different, then add Rename$(e, map(e))$ to $op(e)$.

ToElement Rule. If e has an attribute $attr$ such that $map(attr)$ is a child of $map(e)$ in S_g, then add ToElement$(e, attr)$ to $op(e)$.

Insert Rule. If $map(par(e))$ is an ancestor of $map(e)$ in S_g and for each element $e_i \in p$, where p is the path from $map(par(e))$ to $map(e)$, there is no element in S_ℓ that is mapped to it, then add Insert$(par(e), p, e)$ to $op(par(e))$.

Downgrade Rule. If e has a sibling element e' such that $map(e')$ is a child of $map(e)$ in S_g, then add Downgrade$(par(e), e', e)$ to $op(par(e))$.

Upgrade Rule. If e has a child element e' such that $map(e')$ is a sibling of $map(e)$ in S_g, then add Upgrade(e, e') to $op(e)$.

Exchange Rule. If $map(par(e))$ is the child of $map(e)$ in S_g, then add Exchange$(par(e), e)$ to $op(par(e))$.

The above rules are applied in two phases. In the first phase, S_ℓ is traversed in preorder to update $op(e)$ for each visited element e using only the *Exchange* rule.

Based on those $op(e)$, S_ℓ is transformed to S'_ℓ. In the second phase, S'_ℓ is traversed in preorder to update $op(e)$ for each visited element e using only the remaining five rules. For each e visited, the rules are applied in any order to update $op(e)$ if the rule conditions are satisfied. The application of the *Exchange* rule needs to be performed first before the other rules to avoid the ambiguity on other operators caused by the *Exchange* operator.

6.3 Implementation Issues

This section discusses the implementation issues for the two dynamic rewriting approaches, namely, NDDR and IDDR.

6.3.1 Non-intrusive Dynamic Data Rewriting

In NDDR (Figure 6.2(b)), the key component being introduced is the *data rewriter* which is responsible for generating parsed events for D_g from the parsed events of D_ℓ thereby giving the matching engine the illusion that it is matching its global subscriptions against D_g. In this way, we can avoid changing the complex matching engine component.

Cached-Tree. To dynamically rewrite the data, the *data rewriter* needs to change the sequence of the parsed events. Some events can be forwarded to the processor immediately while some events have to be delayed until other events happen. For those events that are to be delayed, the *data rewriter*s use a tree structure to store them, which is called *cached-tree*. Each element in the document corresponds to one node in the *cached-tree*. The node of the *cached-tree* captures the element's name, attributes and content information. If an element e_i is the subelement of element e_j in the document, the node corresponds to e_i is the child of the node

corresponds to e_j in *cached-tree*.

For each event start_element E received by the data rewriter, the rewriter will initiate the sequence of rewriting operations associated with element E in $M_{\ell,g}$. The complexity and therefore the cost of a rewriting operator depends on whether the operator is *blocking* or *non-blocking*, which is described as follows.

1. **Non-blocking operators.** An operator is classified as *non-blocking* if the effect of its rewriting can be pipelined by the data rewriter (in the form of a parsed event for D_g) to the matching engine immediately. `Rename`(E, E'), `ToElement`(E, A), `Upgrade`$^+(E)$ and `Insert`$^+(E, E_1/E_2/\dots/E_k)$ are non-blocking operators.

2. **Blocking operators.** the *cached-tree* is required to handle these operators. The *data rewriter* informs the matching engine about a batch of events from the *cached-tree* once some further event is parsed. `Exchange`(E, E'), `Insert`$(E, E_1/E_2/\dots/E_k, S)$, `Upgrade`$(E, S)$ and `Downgrade`(E, S, E') are blocking operators.

In the following, we first introduce the mechanisms to perform non-blocking operators, followed by the mechanisms to perform blocking operators. Let *Start(E)* denote the event start_element E; and let *End(E)* denote the event end_element E.

Rename(E, E')**.** For the event *Start(E)* from the parser, the data rewriter substitutes the name E with the name E' and forwards the event *Start(E′)* to the matching engine. Similarly, for *End(E)* from the parser, the data rewriter forwards *End(E′)* to the matching engine.

ToElement(E, A)**.** For event *Start(E)* from the parser, the rewriter extracts the attribute A and the value of the attribute A (denoted as $val(A)$). Then attribute A is erased from the element E. The rewriter first sends the event *Start(E)* to the

matching engine (other attributes of E is sent together with *Start(E)*). Secondly, the rewriter notifies the matching engine about the event *Start(A)*, followed by the event $text(val(A))$. Finally, the rewriter sends the event *End(A)* to the matching engine. For *End(E)* the data rewriter just forwards *End(E)* to the matching engine.

Insert$^+$($E, E_1/E_2/\ldots/E_k$). For event *Start(E)* from the parser, the data rewriter informs the matching engine of *Start(E)* followed by the sequence of events *Start(E_1)*, *Start(E_2)*, ..., *Start(E_k)*. And when the *End(E)* is obtained from the parser, the rewriter first sends the event *End(E_k)* to the matching engine, followed by the events *End(E_{k-1})*, ..., *End(E_1)*, and finally the event *End(E)* is forwarded to the matching engine.

Upgrade$^+$(E). For the *Start(E)* from the parser, the data rewriter would first send the event *Start(E)* to the matching engine. Then the rewriter generates an event *End(E)*, and sends it to the matching engine. This is to tell the matching engine that element E is finished, then the following elements that are the children of element E at the parser's side are treated as the siblings of element E at the matching engine's side. The *End(E)* from the parser will be ignored by the data rewriter, since an *End(E)* event has be informed to the matching engine already.

Exchange(E, E'). Since *Start(E')* will be parsed after *Start(E)*, the data rewriter needs to build a *cached-tree* for the subtree rooted at E (excluding the subtree rooted at E'), and also a *cached-tree* for the subtree rooted at E'.

The detail operations for $Exchange(E, E')$) are as follows : when the *Start(E)* is encountered, the rewriter creates a *cached-tree* T_E with root E. The following encountered elements are inserted into tree T_E. When *Start(E')* is obtained from the parser, the rewriter builds a new *cached-tree* $T_{E'}$ with root E'. Then the following elements should be added to tree $T_{E'}$, and the rewriter should add the tree T_E as the child subtree of node E'; when *End(E')* is encountered, $T_{E'}$ is finished.

Then the following parsed elements are inserted to tree T_E. Finally when event *End(E)* is obtained, the rewriter traverses the tree $T_{E'}$ in pre-order sequence. Given each node E in the tree, *Start(E)* is issued to the matching engine when the node is visited for the first time; and *End(E)* is issued when the traverser traces back from the node. Tree T_E is traversed after issuing the *Start(E′)*. There may exist more than one E' as descendants of element E, thus we may have multiple tree $T_{E'}$s, and these trees are traversed one by one. We do not need to store multiple T_E in memory, since the same T_E are shared by all $T_{E'}$s.

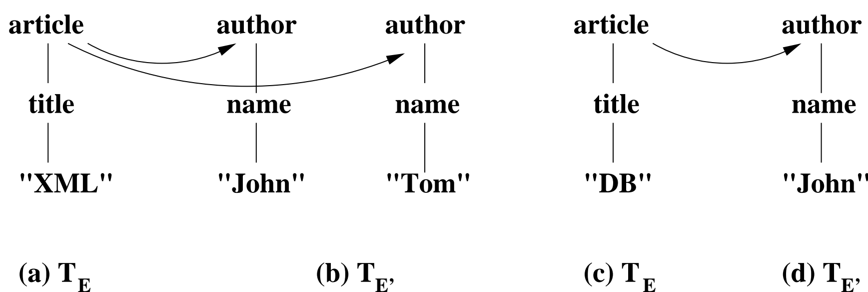


Figure 6.5: The Example for Exchange Operation

Example 6.1 Consider the rewriting of document D_ℓ in Figure 6.4(a), whose mapping is shown in Figure 6.3. Figure 6.5(a)(b) illustrates the *cached-tree* that are created for rewriting the first subtree rooted at "article", and Figure 6.5(c)(d) shows the *cached-tree* for rewriting the second subtree rooted at "article". Take the first subtree rooted at "article" as an example. When the data rewriter gets the *Start(article)*, the $M_{\ell,g}$ indicates that operator $Exchange(article, author)$ is to be performed, then the rewriter creates a tree T_E with root "article". The following elements "title" with its content "XML" is added in T_E. When *Start(author)* is encountered, the rewriting knows that parameter E', i.e. "author", is satisfied, then the tree $T_{E'}$ with root node "author" is created. The following node "name" is inserted to $T_{E'}$. Tree T_E is also added as the child subtree of node "author", as shown by the arrowed line the Figure 6.5. Similarly, when the second "author"

is encountered, the reference to the child subtree T_E is inserted. Finally, when *End(article)* is encountered, the rewriter traverses the tree rooted at "author" to inform the matching engine of the corresponding events. □

Insert$(E, E_1/E_2/\ldots/E_k, S)$**.** For the *Start(E)* from the parser, the data rewriter notifies the matching engine of *Start(E)*. At the same time, the data rewriter creates the *cached-tree* rooted at node E_1 with a child path $E_2/\ldots/E_k$, denoted as T_E. Then for each following element E' which are one level deeper than element E, i.e. E's subelements, the data rewriter checks whether $E' \in S$. If $E' \in S$, the subtree rooted at E' is inserted to the tree T_E as the child subtree of E_k; otherwise, the data rewriter just forwards the event to the processor. When the event *End(E)* is encountered, the data rewriter traverses the tree T_E and issues the corresponding events in the same way as in the *Exchange* operator. Finally, the data rewriter issues the event *End(E)* to the matching engine.

Upgrade(E, S)**.** For the *Start(E)* from the parser, the data rewriter forwards it to the matching engine immediately. Then for each following element E' which is the subelement of E, the data rewriter checks whether $E' \in S$. If $E' \in S$, the rewriter creates a tree $T_{E'}$ with root node E' and the descendant elements of E' are inserted to $T_{E'}$, and the tree $T_{E'}$ is finished when the event *End(E')* is encountered; otherwise if $E' \notin S$, then for element E' and all its descendants, the rewriter forwards the events to the matching engine immediately. When event *End(E)* is encountered, the rewriter first sends *End(E)* to the matching engine, and then the rewriter traverses the set of *cached-tree* $T_{E'}$ and issues the corresponding events to the matching engine.

Downgrade(E, S, E')**.** For the *Start(E')*, the data rewriter creates a tree $T_{E'}$, and the descendant elements of E' are inserted to $T_{E'}$ as the descendant node of E'. For

the *Start(e)*, in which $e \in S$, a tree T_e is created, and the descendants of element e are inserted into T_e. When the *End(E)* (i.e. the parent of the element E' and $e \in S$) is encountered, the data rewriter first traverses the tree $T_{E'}$ in preorder and issues the corresponding events to the matching engine. Before issuing the event *End(E')*, the rewriter finds the tree T_e ($e \in S$), i.e. tree to be moved as the child subtree of element E', and traverses the tree T_e to issue each event to the matching engine. Finally, the event *End(E')* followed by the event *End(E)* are forwarded to the matching engine.

6.3.2 Intrusive Dynamic Data Rewriting

Among the three data rewriting approaches, IDDR (Figure 6.2(c)) is the most complex to implement as it is an intrusive approach that necessitates modifying the matching engine so that it integrates both the dynamic rewriting functionality as well as the subscription matching functionality. To realize this dual functionality efficiently, the matching engine actually maintains partial matchings of subscriptions based on the assembled fragments of D_g that are rewritten from the parsed events of D_ℓ. In this way, we do not need to first materialize the rewritten data D_g before the subscription matching can commence.

To understand why matching in IDDR becomes more complex than the conventional matching in SDR and NDDR, note that the matching engine works by maintaining partial matches of subscriptions as the document is being parsed and the parsed events are being incrementally processed. Once an event start_element E is encountered, the matching engine updates any partial matchings with the new element E at the current context; and once an event end_element E is encountered, the matching engine eliminates the partial matchings that are guaranteed to not lead to any complete matchings. The matching of the elements and the elimination

of partial matchings are based on two properties of conventional event-based XML parsers:

1. Once the start_element event for an element E is received, all the ancestor elements of E must necessarily have been parsed; and
2. Once the end_element event for an element E is received, all the descendant elements of E must necessarily have been processed.

Based on the first property, the matching engine can detect all the partial matchings involving element E for the event start_element E; and based on the second property, when the event end_element E is encountered, the matching engine can safely eliminate all partial matchings that entail the matchings in the subtree of rooted at element E.

However, the above two properties that facilitate the updating of partial matchings are no longer satisfied for IDDR for two reasons. Firstly, some elements in D_g may be parsed earlier than their ancestor elements. For example, the operator `Downgrade`(E, S, E') will move the subtree rooted at E_i, where $E_i \in S$ to become a child subtree of E'. Consequently, element E_i may precede element E' in the document such that the start_element of E_i is output by the parser before the start_element of E'. This means that the matching engine has to process E_i without its ancestor element E'. A similar issue also arises with the operator `Exchange`(E, E').

Secondly, when the event end_element E_i is encountered, it may happen that not all of E_i's descendants in D_g have been parsed. Consider again the operator `Downgrade`(E, S, E'). When element E' precedes element E_i, where $E_i \in S$ in the document, the end-tag for E' is reported by the parser before element E_i which should be the descendants of E'. The operator Exchange(E, E') and

Insert($E, E_1/E_2/\dots/E_k,\ S$) face this issue as well.

The integrated matching engine therefore maintains two types of partial matchings. Given an event start_element E, if all its ancestors have been already parsed, then the partial matchings detected by element E are confirmed. We call such partial matchings as *confirmed partial matchings*; otherwise, if some of its ancestor elements have yet-to-be-parsed, the partial matchings detected by element E cannot be determined. We call such matchings as *potential partial matchings*. The matching engine maintains both *confirmed partial matchings* and *potential partial matchings* that are detected by element E. Once the element that incurs the potential partial matchings have been parsed, the matching engine uses this element to verify the potential partial matchings. The successfully matched potential partial matchings are handled in the same way as the confirmed partial matchings.

To handle the second problem that the descendant elements of an element E could be parsed after the event end_element E, the matching engine continues to keep the partial matchings that can be combined with the matchings from the descendant elements of E to generate larger matchings. These partial matchings are eliminated once the matching engine determines that all the descendants of E have been processed.

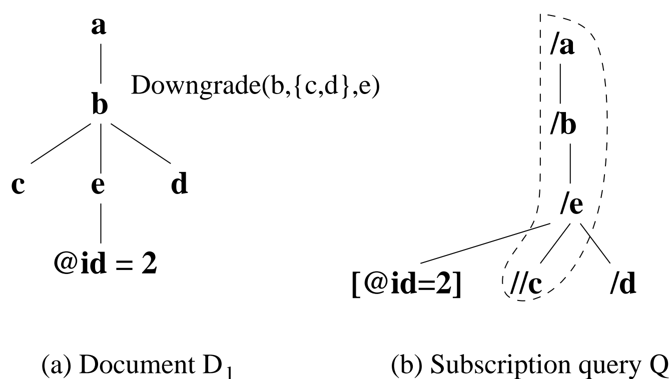


Figure 6.6: IDDR Example

Example 6.2 Consider the document D_ℓ and query Q in Figs. 6.6(a) and (b), respectively. Suppose the operator `Downgrade` (b, $\{c, d\}$, e) is to be performed on D_ℓ. When the event start_element c is received by the matching engine, the engine knows that element c should be matched under element e, which is yet-to-be-parsed. Thus, the matching engine can only detect the partial matching $/a/b/e//c$ as potentially partial matched, which is shown by the path of query nodes enclosed by a dashed region in Figure 6.6(b). This is because the matching engine processor does not yet know whether the element e contains an attribute "id" with a value of 2. Subsequently, when the the event start_element e is parsed, the *potential partial matching* $/a/b/e//c$ is confirmed. When the event end_element e is encountered, since the matching engine knows that there may exist some elements that need to be downgraded as descendants of e, the partial matchings detected by e are still maintained. To handle the start_element of d, the complete matching of query Q is detected. Notice that if query Q was not matched when the end_element b is encountered, then the partial matchings detected by element e can be eliminated since the matching engine guarantees that no elements will be processed as descendant elements of e. □

Discussions. The intrusive dynamic data rewriting problem incurs the evaluations of partial XPath queries on a fragment of XML data, which has some similarity with the problem presented in the previous chapter. However, the solution introduced in the previous chapter is not suitable to resolve the matching problem in the intrusive dynamic data rewriting due to the following reasons.

1. The approach introduced in the previous chapter was proposed to handle the small-scale deployment, such as monitoring applications that run on mobile devices, where the number of queries is not large. However, for the large-

scale deployment described in this chapter, this approach would incur a large number of subqueries, and the cost to derive these subqueries and to index these subqueries on-the-fly is relatively large.

2. The approach for query processing on fragmented XML data requires the knowledge of header information for the fragments before the query evaluations, and such information will be used during the query evaluations to detect the redundant and unnecessary evaluations. However, in the intrusive dynamic data rewriting problem, the incoming XML data is a complete document, and there is no fragment header information before the query evaluations. Thus there is no information about the set of relevant subqueries of other fragments. It follows that no information can be used to detect the dynamic optimizations, i.e. eliminating redundant evaluations and eliminating unnecessary evaluations.

Considering that in the intrusive dynamic data rewriting, the fragment of subtree would be rewritten to a position not far from its current position. Thus to evaluate the fragment of rewritten subtree in the proper context would only incur small modification of the current evaluation context, and the maintenance for potential matching results would only need a short period compared with the whole evaluation procedure. Thus the approach described in this chapter is more suitable for the query evaluation in the intrusive dynamic data rewriting.

6.4 Experimental Study

This section reports our extensive experimental results, which demonstrate the effectiveness of the proposed data rewriting approach. The results show that the

approaches IDDR and NDDR outperform all other approaches under various conditions.

6.4.1 Experimental Testbed

NS2 [5] network simulator extended by the content-based routing application, is used to study the filtering efficiency. Both the linear and tree structure are used for network topology. The parameters for network setup are listed in the first part of Table 6.1. For tree topology, a complete binary tree with four levels and 15 routers in total is used. The experiments are also conducted on a real network, which is denoted as *Real*. Four computers in a LAN are selected to form a linear topology.

Para	Description	Value
T	types of topology	**Linear**, Tree
N	#routers on the linear path	2, **4**, 8
λ	bandwidth (Mbps)	10, **50**, 100
P	#subscriptions per router	1000,**2000**,4000
D	size of the data sets	10K,**20K**,40K,1M
r	cardinality of a certain subtree	2, 4, 6, 8
L	maximum number of steps	**8**
ρ_*	probability of wildcard *	**0.2**
ρ_λ	probability of nested paths	**0.2**

Table 6.1: Parameters and Values (the default values are indicated in bold)

Data sets. We make use of the THALIA benchmark [65], which contains a collection of 40 similar XML schemas representing the university course catalogs from computer science departments around the world. The schema mapping tool provided by [82] is used to generate the global schema and the mappings between the local schemas to the global schema. The documents are generated based on the provided XML documents in THALIA with respect to each schema. The parameters for data sets are shown in the middle of Table 6.1. The size 10K represents the data set in which the size range of documents are [10KB, 20KB). The param-

eter r is used to illustrate the tradeoff between NDDR and IDDR, which is to be explained later. We do not control r to generate documents in other experiments, thus no default value is set for r.

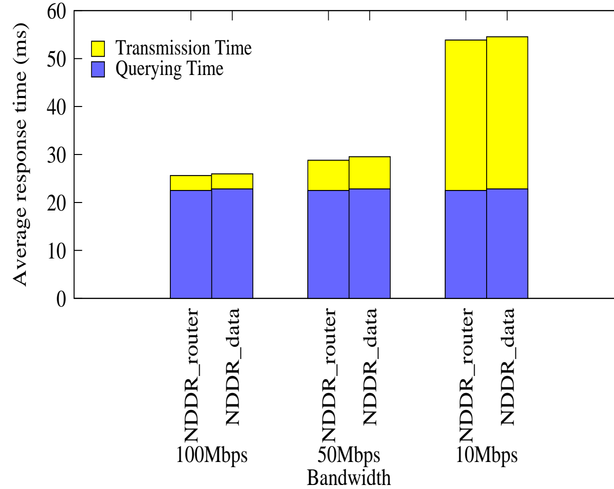
Subscriptions. The XPath queries are generated using XPath generator in [49]. The parameters and values used are shown in the bottom of Table 6.1.

Algorithms and Metric. The performances for SDR, NDDR, IDDR and DRR were studied and compared. The *average response time* is used to measure the performance, which is defined as the average time for all users to receive the document d since it is published. All experiments were conducted on a 3GHz Intel Pentium IV machine with 1GB main memory running Windows XP, and all algorithms were implemented in C++.

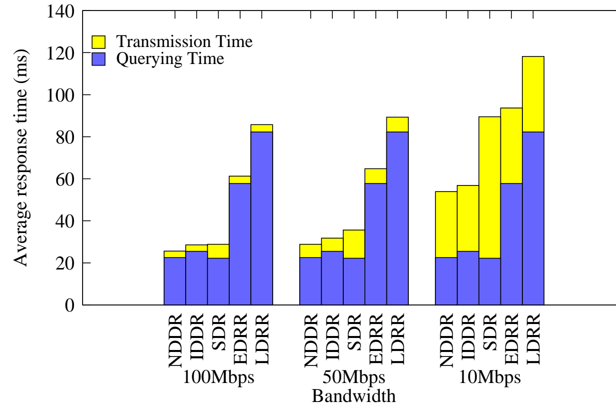
6.4.2 Experimental Results

The *average response time* can be divided into two components : (1) querying time (denoted as T_q), which is the time for matching queries; (2) transmission time (denoted as T_t), which is the time for transmitting data in the network. The stacked barcharts are used to demonstrate these components.

Comparison of different schema mechanisms. As mentioned in Section 6.2.2, there are two options to store the $M_{\ell,g}$ for dynamic data rewriting, which are denoted as $NDDR_{router}$ (resp. $IDDR_{router}$) and $NDDR_{data}$ (resp. $IDDR_{data}$). Figure 6.7(a) shows the performance comparison for $NDDR_{router}$ and $NDDR_{data}$ by varying the bandwidth. The performance of $IDDR_{router}$ and $IDDR_{data}$ shows the similar trends, which is omitted here. We observed that approaches $NDDR_{data}$ and $NDDR_{router}$ always achieve similar performance. It indicates that the header information is small such that the overhead to transmit it is trivial and to create the



(a) Comparison of different schema mechanisms



(b) Comparison of different approaches

Figure 6.7: Comparison of different schema mechanisms & data rewriting approaches

schema mapping on-the-fly based on the header is fast. Therefore, in the following, we use NDDR(resp. IDDR) to represent both $NDDR_{data}$ (resp. $IDDR_{data}$) and $NDDR_{router}$(resp. $IDDR_{router}$).

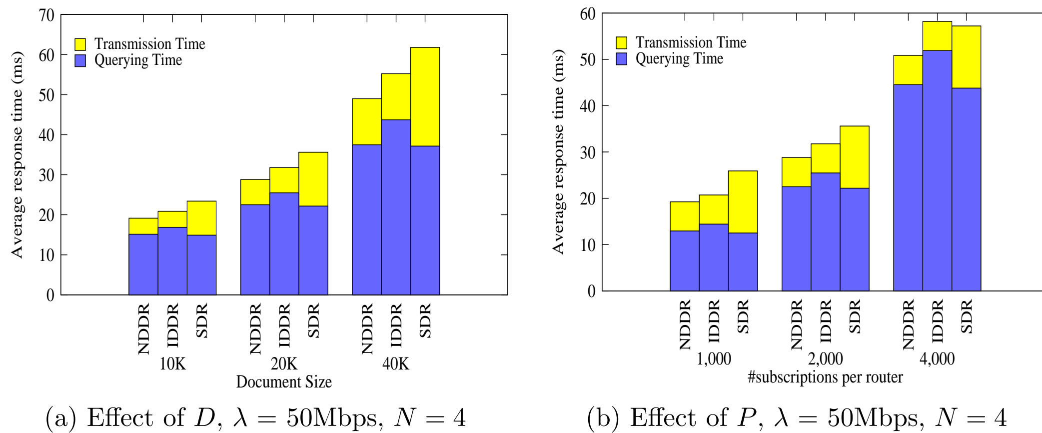
Comparison of different approaches. Figure 6.7(b) compares the performance of different data rewriting approaches. Firstly, it shows that the approaches EDRR and LDRR are obviously outperformed by the proposed dynamic data rewriting approaches, i.e. NDDR and IDDR. This is due to the overhead to rewrite D_g to D_ℓ in these approaches, which takes about 27% of the total time. To rewrite

D_g to D_ℓ immediately once a query from some user is matched in EDRR may delay the matching of the following queries. Since if the query from some user is matched before the query from the downstream router, EDRR will delay the documents to be forwarded to the downstream router, which also increases querying time on the downstream routers. However, to rewrite D_g to D_ℓ after parsing the document completely in LDRR delays the forwarding of the document to end users. This increases the querying time on this router, especially for the queries that are matched at the beginning of the document. And with the overhead to rewrite D_g, LDRR achieves the worst performance. Due to the bad performance of EDRR and LDRR, we ignore the results for them in the following charts.

Secondly, we observe that the dynamic data rewritten approaches (i.e. NDDR and IDDR) outperforms the approach SDR to achieve the best performance. NDDR obtains similar querying time with SDR, which means that the additional cost for dynamic data rewriting in NDDR is trivial. The amount of data transmitted in SDR is about twice of the amount in NDDR and IDDR, thus SDR incurs much larger transmission time. Therefore, the performance of SDR is outperformed by NDDR and IDDR. NDDR and IDDR have the same transmission time. However, due to the complicated matching algorithm in IDDR, it incurs slightly larger querying time. Thus NDDR achieves better performance than IDDR.

Effect of the bandwidth, λ. Figure 6.7(b) demonstrates the effect of network bandwidth in by decreasing λ from 100Mbps to 50Mbps and to 10Mbps. As λ decreases, the components of transmission time T_t for each approach grow. The effect of bandwidth to NDDR and IDDR is the same, since they transmit same amount of data. SDR deteriorates faster as the decreasing of λ, since the amount of data transmitted in SDR is twice of NDDR and IDDR. For λ = 100Mbps, the component of T_t is very small. When λ decreases to 50Mbps, the component of

T_t takes a small part of the response time. However, when λ further decreases to 10Mbps, the component for T_t takes a large part of response time, especially for SDR that T_t is about 78% of the response time. Thus as λ decreases, the improvement of NDDR over SDR increases from 12% at λ = 100Mbps to 41% at λ = 10Mbps. The internet develops fast in recent years, however the bandwidth is still the critical resource, which makes SDR not suitable for small bandwidth environment.



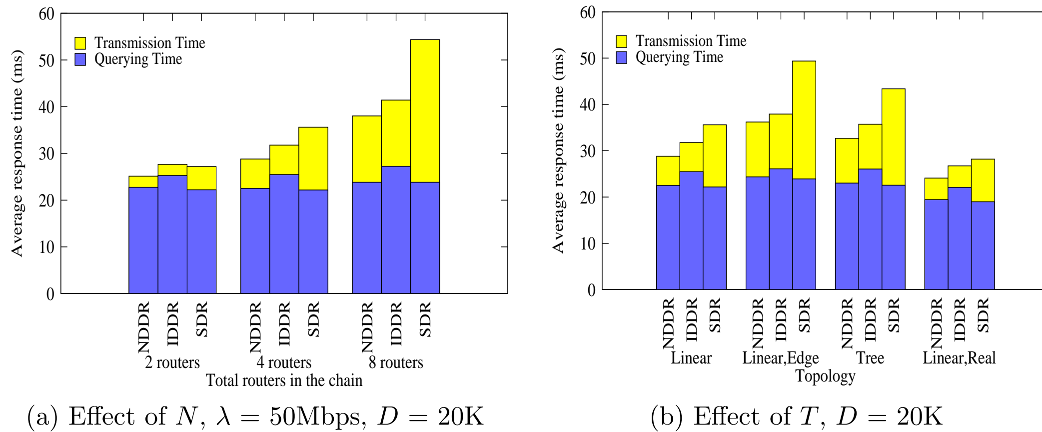
(a) Effect of D, $\lambda = 50$Mbps, $N = 4$ (b) Effect of P, $\lambda = 50$Mbps, $N = 4$

Figure 6.8: Effect of document size and number of subscriptions per router

Effect of the document size, D. Figure 6.8(a) shows the performance by varying D using numbers indicated in Table 6.1, while λ is the default value 50Mbps. As D increases, the *average response time* for all approaches increases due to larger querying time and transmission time. We observe the performance gap between NDDR and IDDR becomes slightly larger, since larger documents have more affected on IDDR due to the more complicated matching algorithm in IDDR. It also shows that the improvement of NDDR over SDR becomes larger as D increases, from 20% at D = 10K to 25% at D = 40K. Similarly, the improvement of IDDR over SDR also increases. The reason is that larger documents incur larger transmission delay in SDR. The results for 1M dataset are omitted here since its average response time is much larger than other datasets, which is not suitable to be shown

in the same chart. We observe that the trends from 10K to 40K also keeps at $D = 1M$, that is the improvement of NDDR over IDDR is 10%, and over SDR is 28%.

Effect of #subscriptions per router. Figure 6.8(b) shows the results by varying the number of subscriptions P from 1000 to 2000 to 4000. As P increases, The querying time T_q for all three approaches increase correspondingly. The increasing of querying time for NDDR and SDR is the same, thus the performance gap between NDDR and SDR keeps the same. However, due to its complicated matching algorithm, the increasing of querying time for IDDR is larger than NDDR and SDR, thus the improvement of NDDR over IDDR becomes larger.



(a) Effect of N, $\lambda = 50$Mbps, $D = 20$K
(b) Effect of T, $D = 20$K

Figure 6.9: Effect of network topology

Effect of the network topology. This section studies effect of the network topology on the performance.

Firstly, we increase N (i.e. #routers on the linear path) from 2 to 4 and to 8. The results are shown in Figure 6.9(a). We observe that as N grows, The transmission time T_t for all approaches increases correspondingly since the document has to travel longer path before arriving at end users. The approach SDR that has to transmit larger amount of data got larger transmission cost as N increases. We

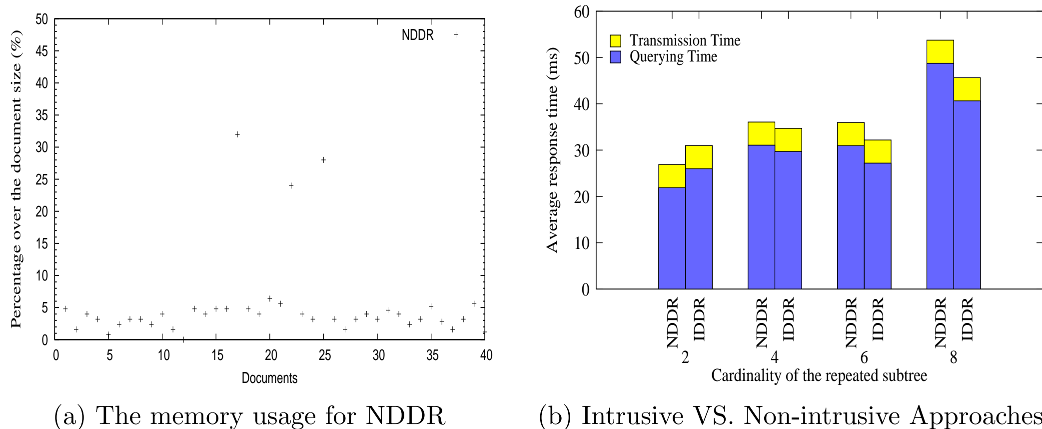
can see that the improvement of NDDR over SDR increases from 10% at $N = 2$ to 31% at $N = 8$.

Secondly, we test the case when only leaf router (the router without the downstream router) has the subscriptions from the users. The results are shown by second cluster of bars in Figure 6.9(b). The transmission to the leaf router incurs larger delay compared with upstream routers. Thus the performance gap between NDDR (also IDDR) and SDR becomes larger.

Thirdly, we show the results on the $Tree$ topology using the third group of bars in Figure 6.9(b). Compared with *Linear*, the $Tree$ topology has more routers as the leaf routers. As aforementioned, queries on leaf routers incur larger T_t, the transmission delay becomes larger in tree topology, thus the performance margin between NDDR (as well as IDDR) and SDR becomes larger.

Results on the real network. We also experimented on a real network *Real* as described in Section 6.4.1. The forth group bars in Figure 6.9(b) show the results on *Real*. As we known, the bandwidth in the LAN is usually large. The bandwidth in the LAN we used is more than 50Mbps. We can see that the performance of all approaches on *Real* have similar trends with the performance of them on NS2 with λ = 50Mbps. NDDR achieves the best performance among all approaches. It proves that the simulation using NS2 measures the performance well.

Memory usage of NDDR. The approach NDDR may need to cache certain parsed events to dynamically rewrite data, which incurs additional memory usage. Figure 6.10(a) shows the memory usage in NDDR for D = 20K. The y-axis is the percentage of the largest memory used for the caching over the size of the parsed document. In the worst case, we need to cache the whole document. However, Figure 6.10(a) shows that for the set of documents, the largest memory usage takes 32% of the document size, and only three documents require cached space over 20%



(a) The memory usage for NDDR (b) Intrusive VS. Non-intrusive Approaches

Figure 6.10: Experimental Results

of the document size. For most of the documents, the memory usage is around 5% of the document size. And as we know, the documents in the data dissemination are usually small, which means that the NDDR approach will not take up much memory.

NDDR vs. IDDR. The previous results show that IDDR is slightly outperformed by NDDR since IDDR makes the matching algorithm more complicated. However, in some situation, IDDR is more efficient than NDDR by sharing the processing of repeated subtrees. For example, operator $Exchange(N_1, N_2)$ makes the subtree rooted at N_1 be repeated at the subtree rooted at N_2 if the cardinality of element N_2 in document is larger than 1. In this experiment, we select the document that contains the operator $Exchange(N_1, N_2)$ and vary the cardinality of N_2 (denoted as r) from 2 to 8 in the step of 2. We observe that when $n = 2$, NDDR is better than IDDR due to the complicated matching in IDDR. However, when $n = 4$, IDDR starts to outperform NDDR, and as n increases, the improvement of IDDR over NDDR becomes larger. The reason is that in IDDR, the processor is aware of the data rewriting, and knows that same subtree rooted at N_1 is repeated under each N_2, thus IDDR shares the processing of the subtree rooted at N_1. As n increases, the improvement of IDDR by sharing the repeated subtree becomes larger, which

#subscriptions	1000	2000	4000
with data rewriting	9.13	9.20	9.37
without data rewriting	1.47	1.54	1.59

Table 6.2: Comparison of recall for with and without data rewriting

compensates the performance loss due to complicated matching algorithm.

Recall comparison for with and without data rewriting. With the data rewriting mechanism, besides documents that exactly match users' queries, the dissemination system will also forward documents conforming to a heterogeneous schema while indeed satisfying users' interests to users. The mechanism helps to increase the recall of the dissemination system such that users have low probability to miss the information that is interested by them. We define the *recall* as the average number of documents that are received by each user. Table 6.2 compares the recall for the dissemination with and without data rewritings by varying the number of subscriptions. We use 100 documents in total, and we observe that the dissemination with data rewritings improves the recall by around six times compared with the dissemination without data rewritings. The data rewriting mechanism helps to transform the documents conforming to heterogeneous data schemas to documents conforming to the global schema such that the documents with heterogeneous structure could also be delivered to users.

6.5 Summary

This chapter introduced the the novel paradigm based on the principle of *data rewriting* to address the schema heterogeneity problem in content-based dissemination of XML data. Several data rewriting approaches are proposed and explored such as SDR, DDR and DRR. With respect to DDR, two options are compared, i.e. NDDR and IDDR. The NDDR is non-intrusive approach in the sense that

it does not require the modification on the matching engine; while IDDR is an intrusive approach where the matching engine should be enhanced. With respect to DRR, two approaches are proposed in terms of the time to perform the *reverse data rewriting*, i.e. EDRR and LDRR.

Based on the experimental results, we have the following observations on the efficiency of various approaches. First, DRR always achieves the worse performance due to the extra cost for reverse rewriting, and its delay to forward documents. Second, to disseminate schema mapping $M_{\ell,g}$ as the data's header information incurs little overhead, and this approach is space- and update-efficient. To compare among IDDR, NDDR and SDR, SDR does not perform well due to the transmission of additional data, especially when the bandwidth is small or the number of hops to subscribers is large. Moreover, IDDR does not scale well as the number of subscriptions or the size of documents increases since it complicates the matching algorithm. Generally speaking NDDR achieves the best performance, and the memory usage in NDDR is small. However, IDDR performs better than NDDR when there are many duplicated subtrees in the rewriting of D_ℓ to D_g.

Chapter 7

Conclusions

This chapter concludes this dissertation by summarizing our work, discussing the contributions of this thesis and presenting some future work.

7.1 Summary

The information explosion nowadays makes the *push-based* communication model more suitable for large scale distributed information system. The features of XML make it become the *de facto* standard for information exchange on the Internet. The above two reasons result in the importance of content-based dissemination of XML data in the large-scale distributed information systems. In this dissertation, we presented three work that aim to improve the efficiency and effectiveness of content-based dissemination of XML data systems.

Chapter 4 introduced a novel global optimization approach by using piggy-backed annotations to improve the filtering efficiency in content-based dissemination of XML data. We have proposed four types of useful annotations, i.e. PS, PD, NS and ND. We have demonstrated that PS is effective to enable the early forwarding of a document at downstream routers; NS and ND are effective to reduce

the number of processed subscriptions on downstream routers; and PD turns out to be not effective since the benefit of PD is offset by the overhead to process it. We have also proposed a new matching protocol, which is called lazy forwarding. The best performance is achieved by the approach using lazy forwarding with the combination of annotations PS, NS and ND.

Chapter 5 presented an approach to evaluate a set of boolean XPath queries on fragmented XML data without reconstructing the original XML document. We have proposed a three-step strategy for query processing. Various scheduling strategies for processing fragments are exploited and two dynamic optimizations are also proposed. The experimental studies demonstrate the effectiveness of the proposed approach with the intelligent scheduling strategies and dynamic optimizations.

Chapter 6 addressed the schema heterogeneity problem in content-based dissemination of XML data. Considering the characteristic of the content-based dissemination system, i.e. multiple-query-single-data, a novel paradigm based on the principle of *data rewriting* is proposed. We have explored a variety of data rewriting approaches, i.e. SDR, DDR and DRR. DDR is shown to be the best approach. We have considered two options to implement DDR, which is IDDR and NDDR. It is shown that in most cases NDDR outperforms IDDR since it does not make the matching engine complicated, and the memory usage in NDDR is small. However, IDDR performs better than NDDR when there are many duplicated subtrees in the rewriting of D_ℓ (i.e. data conforming to some local schema) to D_g (i.e. data confirming to the global schema).

7.2 Contributions

Two important factors for content-based dissemination of XML data systems are the *efficiency* of the system and the *range of functionalities* provided by the system. We believe that this thesis has contributed to a more full-fledged content-based XML dissemination system. The two main contributions of this thesis are listed as follows.

Improving the efficiency. A novel global optimization based on piggybacking annotations is introduced. The global optimization is orthogonal with the existing local optimization approaches such that it can be combined with any local optimization approach to further improve the filtering efficiency. We believe that the global optimization is effective to improve the filtering efficiency significantly.

Expanding the range of functionalities. Functionality is also an important aspect for the content-based dissemination system. This dissertation extends the functionalities of the existing system by providing the approach to handle the XML documents that are published in fragments and to handle the XML documents with heterogeneous schemas. These extensions make the current dissemination system more effective.

7.3 Future Work

This section suggests several future directions based on our research work. The further work for each chapter is described first, followed by the possible future work for the general content-based dissemination system.

Global Optimization for XML Data Dissemination (Chapter 4). The annotations proposed in Chapter 4 only explore the properties of the structural

aspect of XPath expressions. There may exist other types of annotations that could also optimize the filtering efficiency. For example, if a document has an annotation specifying that the values of each attribute "price" in this document are greater than 4, then a router can immediately know the non-matching of a subscription which has a value predicate like [@$price < 3$]. Therefore, there is still something to explore for other types of annotations.

Recently, a new type of subscriptions called *stateful subscriptions* has been proposed [46]. Stateful subscriptions address multiple events, and they are matched by joining the matching on multiple documents. The proposed annotations in Chapter 4 can be used to handle such subscriptions, since if we can determine the non-matching on one document, we can conclude that the subscription does not match the sequence of documents. However, some specific annotations that make full use of the properties of these new types of subscriptions would achieve better performance. Therefore, some annotations should be further studied for these new types of subscriptions.

Additionally, there is another direction to further improve the dissemination efficiency by considering a better mechanism to allocate users' subscriptions to each router and to aggregate the collection of subscriptions with more similar structures into one general subscription.

Handling Fragmented XML Data (Chapter 5). The work in Chapter 5 focuses on the query evaluation on fragmented XML data. However, an intelligent fragmentation strategy for XML data can also benefit the query evaluation on the fragments. For example, based upon the knowledge of the set of subscriptions to be evaluated, a better way to do the fragmentation is that the part of documents that is relevant with a certain workload should be allocated to the same fragment such that only the relevant part of the documents will be processed. Therefore a new

fragmentation strategy that is customized by various workloads could be studied further. It follows that the mechanism to maintain the fragmentation information could be modified as well.

Another aspect to consider in this work is the parallel processing for the set of XML fragments. Since we have disjointed XML fragments, the parallel processing for these fragments may help to further improve the efficiency. The challenges to consider here are how to distribute the workload on different processors and how to combine the partial matching results on different processors to obtain the matching results for the complete queries.

Handling Heterogeneous XML Data (Chapter 6). The *data rewriting* approaches proposed in this chapter do not leverage the knowledge of subscriptions on the router. For the parts of the document that do not affect the subscription matching results, it is unnecessary to rewrite these parts. The routers have the knowledge of subscriptions on them before filtering, and the routers also have the knowledge of the schema of documents, thus they can determine the parts of the document that need not be rewritten. Therefore, a customized data rewriting approach might be studied, and it could further improve the data rewriting efficiency.

The following sections describe some future directions for content-based dissemination systems.

QoS Constraints in the Content-based Dissemination. Many existing information dissemination systems are based on the best effort principle. Subscribers are only allowed to issue their interests for the kind of XML document, but there are no parameters addressing the quality of the dissemination service provided to them. However, in some cases the Quality-of-Service of the filtering of XML documents should be considered. For example real-time trading systems need to guarantee the deadline for the arrival of current prices, and air-traffic control systems require

up-to-date data on aircraft position and status. Although the quality-of-service considered in dissemination systems may reduce the whole systems' throughput, the requirement of users about the quality-of-service should be guaranteed.

To consider QoS problem in dissemination systems, we need to provide more parameters for users to specify, such as the deadline to receive some information [99] and the priority of some queries. The dissemination system should be modified to adopt some strategy to match the incoming documents against subscriptions such that all users' requirements are best satisfied.

Hybrid Content-based Dissemination. The existing content-based dissemination systems either handle pure XML data or process pure attribute-value pairs. However, both XML data and attribute-value based data are published on the Internet, and there may be even other formats of data. A hybrid content-based dissemination system can provide users with a uniform interface to subscribe their queries, while the routers in the system take charge of matching the queries with various formats of published information. For such kind of dissemination systems, we need to consider what type of query interface is proper for users and how to index the queries such that the processing on different data formats can be optimized.

Bibliography

[1] Apache Xerces. http://xml.apache.org/xerces-c/.

[2] DataPower. http://www.datapower.com/products/xmlrouter.html.

[3] DBLP. http://dblp.uni-trier.de/xml/.

[4] Gnome libxml2. http://xmlsoft.org/.

[5] *NS2. version ns-2.1b8. http://www.isi.edu/nsnam/ns/.*

[6] Protein. http://pir.georgetown.edu.

[7] R. Cover (1999), The SGML/XML web page. http://www.oasis.open.org/cover/sgml-xml.html.

[8] Sarvega. http://www.intel.com/software/xml/.

[9] SAX. http://www.saxproject.org/.

[10] Treebank. http://www.cis.upenn.edu/treebank/.

[11] W3C (1999) XML Path Language (XPath) 1.0. http://www.w3.org/TR/xpath.

[12] W3C (2000) Extensible Markup Language (XML) 1.0 (Fourth Edition). http://www.w3.org/TR/xml.

[13] W3C (2006) XQuery 1.0. http://www.w3.org/TR/xquery.

[14] W3C XML Fragment Interchange. http://www.w3.org/TR/xml-fragment, Febrary 2001.

[15] W3C XML Schema Part 0 (Primer Second Edition). http://www.w3.org/TR/xmlschema-0.

[16] XMark. http://monetdb.cwi.nl/xml/index.html.

[17] XMLBlaster. http://www.xmlblaster.org/.

[18] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for active XML. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[19] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2003.

[20] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, 2000.

[21] S. Amer-Yahia, N. Koudas, A. Marian, and D. Srivastava. Structure and content scoring for XML. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, 2005.

[22] S. Amer-Yahia, L. V. Lakshmanan, and S. Pandit. FlexPath : flexible structure and full-text querying for XML. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[23] M. Antollini, M. Cilia, and A. Buchmann. Implementing a high level pub/sub layer for enterprise information systems. In *Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS)*, 2006.

[24] G. Banavar, T. Chandra, B. Mukherjee, and J. Nagarajarao. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th International Conference of Distributed Computing Systems (ICDCS)*, 1999.

[25] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Science (TOCS)*, 2(1):39–59, 1984.

[26] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[27] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup-queries in P2P networks. In *Proceedings of the 6th International Workshop on Web Information and Data Management (WIDM)*, 2004.

[28] R. Bordawekar and O. Shmueli. Flexible workload-aware clustering of XML documents. In *Proceedings of the 2nd International XML Database Symposium (XSym)*, 2004.

[29] A. Bosworth. Data routing rather than databases: the meaning of the next wave of the web revolution to data management. In *Proceedings of the 28th International Conference on Very Large Data Base*, 2002.

[30] A. Boukottaya and C. Vanoirbeek. Schema matching for transforming structured documents. In *Proceedings of the 5th ACM Symposium on Document Engineering*, 2005.

[31] A. Boukottaya, C. Vanoirbeek, F. Paganelli, and O. A. Khaled. Automating XML documents transformations : a conceptual modelling based approach. In *Proceedings of the 1st Asia-Pacific Conference on Conceptual Modelling (APCCM)*, 2004.

[32] J.-M. Bremer and M. Gertz. On distributing XML repositories. In *Proceedings of the 6th International Workshop on Web and Databases (WebDB)*, 2003.

[33] N. Bruno, L. Gravano, N. Koudas, and D.Srivastava. Navigation- vs. index-based XML multi-query processing. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, 2003.

[34] P. Buneman, B. Choi, W. F. Fan, R. Hutchison, Robert, Mann, and S. D. Viglas. Vectorizing and querying large XML repositories. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2005.

[35] S. D. Camillo, C. A. Heuser, and R. dos Santos Mello. Querying heterogeneous XML sources through a conceptual schema. In *Proceedings of the 22nd International Conference on Conceptual Modelling (ER)*, 2003.

[36] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. AFilter : adaptive XML filtering with prefix-caching and suffix-clustering. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, 2006.

[37] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification services. In *Proceedings*

of the 19th ACM Symposium on Principles of Distributed Computing(PODC), 2000.

[38] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree pattern aggregation for scalable XML data dissemination. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 2002.

[39] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath Expressions. *The International Journal on Very Large Data Bases*, 11(4):354–379, 2002.

[40] C.-Y. Chan and Y. Ni. Content-based dissemination of fragmented XML data. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[41] C.-Y. Chan and Y. Ni. Efficient XML data dissemination with piggybacking. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2007.

[42] R. Chand, P. Felber, and M. Garofalakis. Tree-pattern similarity estimation for scalable content-based routing. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, 2007.

[43] Y. B. Chen, T. W. Ling, and M. L. Lee. Designing valid XML views. In *Proceedings of the 21st International Conference on Conceptual Modeling*, 2002.

[44] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transaction on Software Engineering*, 27(9):827–850, 2001.

[45] G. Cugola, E. D. Nitto, and G. P. Picco. Content-based dispatching in a mobile environment. In *Proceedings of the Workshop su Sistemi Distribuiti : Algoritmi, Architetture e Linguaggi*, 2000.

[46] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. While. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Extending Database Technoloty (EDBT)*, 2006.

[47] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT)*, 2006.

[48] A. Deshpande, S. Nath, P. Gibbons, and S. Seshan. Cache-and-Query for wide area sensor databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2003.

[49] Y. Diao, M. Altinel, M. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database Systems (TODS)*, 28(4):467–516, 2003.

[50] Y. Diao, S. Rizvi, and M. Franklin. Towards an Internet-scale XML dissemination service. In *Proceeding of the 30th International Conference on Very Large Data Base (VLDB)*, 2004.

[51] A. L. Diaz and D. L. (1999). XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[52] L. Ding and E. A. Rundensteiner. Evaluating window joins over punctuated streams. In *Proceedings of the 13st ACM Conference on Information and Knowledge Management (CIKM)*, 2004.

[53] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[54] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2001.

[55] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query processing of streamed XML data. In *Proceedings of the 11st ACM Conference on Information and Knowledge Management (CIKM)*, 2002.

[56] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *The International Journal on Very Large Data Bases*, 11(4), 2002.

[57] M. Friedman, A. Levy, and T. Millstein. Navigational plans for data integration. In *Proceedings of the 14th National Conference on Arrificial Intelligence (AAAI)*, 1999.

[58] M. Gertz and J.-M. Bremer. Distributed XML repositories : top-down design and transparent query processing. Technical report, Department of Computer Science, University of California, Davis, 2003.

[59] X. Gong, W. Qian, Y. Yan, and A. Zhou. Bloom filter-based XML packets filtering for millions of path queries. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2005.

[60] T. J. Green, G. Miklau, M. Onizuka, and D.Suciu. Processing XML streams with deterministic automata. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, 2003.

[61] A. Gupta, A. Halevy, and D. Suicu. View selection for XML stream processing. In *Proceedings of the 5th International Workshop on the Web & Database*, 2002.

[62] A. Gupta, D. Suicu, and A. Halevy. The view selection problem for XML content based routing. In *Proceedings of the 22nd International Conference on Principles of Database System (PODS)*, 2003.

[63] A. K. Gupta and D. Suciu. Streaming processing of XPath queries with predicates. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2003.

[64] D. P. H. Kellerer, U. Pferschy. *Knapsack Problems.* Springer Verlag, 2005.

[65] J. Hammer, M. Stonebraker, and O. Topsakal. THALIA : Test harness for the assessment of legacy information integration approaches. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2005.

[66] B. He, Q. Luo, and B. Choi. Cache-conscious automata for XML filtering. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2005.

[67] M. Hong, A. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. White. Massively multi-query join processing in publish/subscribe systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2007.

[68] I. Horrocks. DAML+OIL : a description logic for the semantic web. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 25(1), 2002.

[69] S. Hou and H.-A. Jacobsen. Predicate-based filtering of XPath expressions. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, 2006.

[70] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. *Wierless Networks. Special Issue : Pervasive computing and communications*, 10(6):643–652, 2004.

[71] J. Kwon, P. Rao, B. Moon, and S. Lee. FiST: scalable XML document filtering by sequencing twig patterns. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, 2005.

[72] L. V. Lakshmanan and P. Sailaja. On efficient matching of streaming XML documents and queries. In *Proceedings of the 8th International Conference on Extending Database Technoloty (EDBT)*, 2002.

[73] M. Lenzerini. Data integration : a theoretical perspective. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 2002.

[74] H. Leung and H. Jaconbsen. Efficient matching for state-persistent publish/subscribe systems. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative research (CASCON)*, 2003.

[75] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 1995.

[76] A. Y. Levy, A. Rajamaran, and J. Ordille. Query heterogeneous information sources using source description. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, 1996.

[77] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, 2005.

[78] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, 2001.

[79] F. Mandreoli, R. Martoglia, and P. Tiberio. Approximate query answering for a heterogeneouse XML document base. In *Proceedings of the 5th International Conference on Web Information Systems Engineering (WISE)*, 2004.

[80] I. Manolescu, D. florescu, and D. Kossmann. Answering XML queries over heterogeneouse data. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, 2001.

[81] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava. Adaptive processing of top-k queries in XML. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2005.

[82] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding : a versatile graph matching algorithm and its application to schema matching. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, 2002.

[83] M. M. Moro, P. Bakalov, and V. J. Tsotras. Early profile pruning on XML-aware publish-subscribe systems. In *Proceedings of the 33th International Conference on Very Large Data Bases (VLDB)*, 2007.

[84] B. Nguyen, S. A. G. Cobena, and M. Preda. Monitoring XML data on the Web. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2001.

[85] Y. Ni and C.-Y. Chan. Resolving schema heterogeneity in XML data dissemination by data rewriting (Poster Paper). *Proceedings of the 17th International World Wide Web Conference (WWW)*, 2008.

[86] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus - an architecture for extensible distributed systems. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, 1993.

[87] O. Papaemmanouil and U. Cetintemel. SemCast : semantic multicast for content-based data dissemination. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2005.

[88] J. Pereira, F. Fabret, F. Llirbat, H. A. Jacobsen, and D. Shasha. WebFilter : a high-throughput XML-based publish and subscribe system. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, 2001.

[89] M. Petrovic, I. Burcea, and H.-A. Jacobsen. S-ToPSS : semantic Toronto publish/subscribe system. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, 2003.

[90] M. Petrovic, H. Liu, and H.-A. Jacobsen. G-ToPss : fast filtering of graph-based metadata. In *Proceedings of the 14th International World Wide Web Conference (WWW)*, 2005.

[91] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating web data. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 2002.

[92] H. Prüfer. Neuer beweis eines satzes über permutationen. *Archiv für Mathematik und Physik*, 27:142–144, 1918.

[93] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The International Journal on Very Large Data Bases*, 10(4), 2001.

[94] A. Raj and P. S. Kumar. Branch sequencing based XML message broker architecture. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, 2007.

[95] S. Bose, L. Fegaras. Data stream management for historical XML data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[96] S. Bose, L. Fegaras. XFrag: a query processing framework for fragmented XML data. In *Proceedings of the 8th International Workshop on the Web & Databases*, 2005.

[97] S. Bose, L. Fegaras, D. Levine, V. Chaluvadi. A query algebra for fragmented XML stream data. In *Proceedings of the 8th International Symposium on Database Programming Language (DBPL)*, 2003.

[98] T. Schlieder. Schema-driven evaluation of approximate tree-pattern queries. In *Proceedings of the 8th International Conference on Extending Database Technoloty (EDBT)*, 2002.

[99] S. Schmidt, R. Gemulla, and W. Lehner. XML stream processing quality. In *Proceedings of the 1nd International XML Database Symposium (XSym)*, 2003.

[100] B. Segall, D. Aronld, J. Boot, M. Henderson, and T. Phelps. Content based routing with Elvin4. In *Proceedings of the Australian UNIX and Open Systems User group Conference(AUUG)*, 2000.

[101] Z. Shen and S. Tirthapura. Fast event forwarding in a content-based publish-subscribe system through lookup reuse. In *Proceedings of the 5th IEEE International Symposium on Network Computing and Applications*, 2006.

[102] D.-H. Shin and K.-H. Lee. Towards the faster transformation of XML documents. *Journal of Information Science*, 32(3), 2006.

[103] D. Skeen. *Publish-subscribe architecture : publish-subscribe overview. http://www.vitria.com, 1998.*

[104] H. Su, H. Kuno, and E. A. Rundensteiner. Automating the transformation of XML document. In *Proceedings of the 3rd International Workshop on Web Information and Data Management (WIDM)*, 2001.

[105] D. Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems (TODS)*, 27(1), 2002.

[106] Sun Microsystems, Inc. *Java Message Service (JMS). http://java.sun.com/products/jms, 2002.*

[107] I. Tatarinov and A. Halevy. Efficient query reformulation in peer data management systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[108] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *ACM SIGOPS Operating Systems Review*, 24(3):68–79, 1990.

[109] TIBCO. *TIB/Rendezvous, http://www.tibco.com, 1999.*

[110] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(3), 2003.

[111] M. Uschold, P. Clark, F. Dickey, C. Fung, S. Smith, S. Uczekay, M. Wilke, S. Bechhofer, and I. Horrocks. A semantic infosphere. In *Proceedings of the 2nd International Semantic Web Conference (ISWC)*, 2003.

[112] Z. Vagena, M. Moro, and V. Tsotras. RoXSum: leveraging data aggregation and batch processing for XML routing. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, 2007.

[113] E. Y. C. Wong, A. T. S. Chan, and H.-V. Leong. Efficient management of XML contents over wireless environment by XStream. In *Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC)*, 2004.

[114] X. Yang, M. L. Lee, and T. W. Ling. Resolving structural conflicts in the integration of XML schemas : a semantic approach. In *Proceedings of the 22nd International Conference on Conceptual Modeling*, 2003.

[115] X. Yang, M. L. Lee, T. W. Ling, and G. Dobbie. A semantic approach to query rewritting for integrated XML data. In *Proceedings of the 24nd International Conference on Conceptual Modelling (ER)*, 2005.

[116] C. Yu and L. Popa. Constraint-based XML query rewritting for data integration. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[117] X. Zhang, L. H. Yang, M. L. Lee, and W. Hsu. Scaling SDI systems via query clustering and aggregation. In *Proceedings of the 9th International Conference on Database Systems for Advanced Applications (DASFAA)*, 2004.