Query Processing for Large-Scale XML Message Brokering

by

Yanlei Diao

B.S. (Fudan University) 1998
M.Phil. (The Hong Kong University of Science and Technology) 2000

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Michael J. Franklin, Chair
Professor Ion Stoica
Professor Ray R. Larson

Fall 2005

The dissertation of Yanlei Diao is approved:

_____

Professor Michael J. Franklin, Chair Date

_____

Professor Ion Stoica Date

_____

Professor Ray R. Larson Date

University of California, Berkeley

Fall 2005

Query Processing for Large-Scale XML Message Brokering

by

Yanlei Diao

ABSTRACT

Query Processing for Large-Scale XML Message Brokering

by

Yanlei Diao

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael J. Franklin, Chair

Emerging distributed information systems such as Web services, personalized content delivery, and event monitoring require increasingly flexible and adaptive infrastructures. Recently, the *publish/subscribe* model has gained acceptance as a solution for the loose coupling of systems in terms of communication. Meanwhile, with respect to content, XML (*Extensible Markup Language*) is becoming a *de facto* standard for online data exchange. I propose an approach that integrates publish/subscribe and XML and, in particular, exploits declarative XML queries to offer flexibility and adaptivity in distributed systems. This approach is based on building XML message brokers, which I define as middleware components that perform three main functions: *filtering*, *transformation*, and *routing* of XML messages based on user-specified queries.

In this dissertation, I present YFilter/ONYX, an XML brokering system that provides the brokering functions for large numbers of queries over high volume message flows. I describe the architectural design of this system and its underlying technologies for providing efficiency and scalability. A key innovation is the exploitation of commonalities among queries; specifically, I present a series of novel sharing techniques that YFilter employs for

filtering and more sophisticated transformation. A second innovation is the leveraging of relational techniques in the new context of XML message brokering; YFilter uses an effective mapping from XML transformation to relational query processing, which allows known relational techniques to be applied to achieve simplicity and performance of XML transformation. A third innovation is the design of a distributed system, called ONYX, that pushes declarative queries into the network for content-based routing and incremental processing of messages. I report on the results of extensive performance studies, demonstrating the efficiency and scalability of YFilter/ONYX under a wide variety of XML document types and query workloads.

In conclusion, YFilter/ONYX provides three key components, namely, filtering, transformation, and routing, for high-volume XML message brokering. As the adoption of XML-based distributed infrastructures gains momentum, the techniques developed in YFilter/ONYX and the results reported herein provide a foundation for building large-scale, high-function distributed information systems.

_____

Professor Michael J. Franklin, Chair Date

ACKNOWLEDGEMENTS

I would like to thank my research advisor Professor Michael Franklin for his continued guidance and support during my graduate study at the University of California, Berkeley. He is the one who brought me to the world of systems research and taught me many things necessary for being a successful researcher. In particular, he showed me how to conduct convincing experimental studies, how to do critical thinking and writing, and how to give clear, punchy presentations; he also helped me develop the ability to think on the spot and brainstorm ideas, which was so much lacking in me at the beginning of this study. I also thank him for sharing with me his astute sense of what is and what will be important. I am grateful for the numerous meetings that he had with me to discuss my research problems and to guide me through them with his profound knowledge in many diverse areas. I feel even more indebted to the countless hours that he spent away from his family improving my writing and representations. Without his invaluable support and constant guidance, I could not have completed this dissertation.

I also want to thank Professor Joseph Hellerstein for his infectious enthusiasm for database research, supportive inspiration, deep and accurate advice, and wonderful critiques. My gratitude also goes to Professor Ion Stoica for his constant interest in my research, his insightful comments on important issues of my work, and his valuable advice on system modeling and performance analysis from a networking expert's perspective. I also want to acknowledge Professor Ray Larson for participating in my doctoral committee. And, I thank Dr. Michael Carey and Professor Donald Kossmann for their helpful mentoring in academic and personal settings.

Special thanks go to Professor Hongjun Lu, who introduced me to the database field and encouraged me to pursue my Ph.D. study at UCB in the first place. Although he can no longer be with us, his dedication to database research has motivated me and will continue to motivate me in the many years to come.

I would like to express my sincere gratitude to my colleagues in the YFilter/ONYX team. Special thanks to Shariq Rizvi for inspiring conversations and hard work that led to the development of the ONYX system, to Anil Edakkunni for his one-year commitment to the YFilter code release and maintenance, and to Eugene Wu for leveraging YFilter to a complex event processor. I also thank the undergraduate helpers for their assistance in developing the parser, user interface, and demonstration of the YFilter system. I am proud to be part of the YFilter team with them.

My appreciation also goes to other members of the database group at UC Berkeley. I acknowledge Amol Deshpande, Sailesh Krishnamurthy, Sirish Chandrasekaran, and Fred Reiss for their valuable insight, patience for numerous questions, and willingness to work through technical details with me. I owe thanks to Samuel Madden for providing advice on issues of academic career, and Mehul Shah for helping me develop strategies for dealing with criticisms and frustrations. I am also grateful to the following friends that helped foster new ideas and stimulating discussions: David Liu, Matt Denny, Ryan Heubsch, Boon Thau Loo, and Shawn Jeffery. Finally, I thank all my groupmates for their cooperation during work, sympathy and companionship before paper deadlines, and entertainment after work. They made my office a wonderful place to be and my years of study at UC Berkeley an enjoyable and memorable journey.

The research work was made possible by the financial support from many agencies and groups including the National Science Foundation, Boeing, IBM, Intel, Microsoft, Siemens, and the UC MICRO program.

Last but not least, I am deeply indebted to my parents and sister for their unconditional love, everlasting faith in me, and encouraging support for my years of pursuit in academic career. They gave me the strength to overcome the innumerable obstacles that I encountered during my PhD study. I dedicate this dissertation to them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

Distributed information systems provide users with an integrated view of geographically distributed information and the ability to access the information through a universal service interface. Conceptually, these systems contain three layers in a vertical software stack: *presentation* to the user at the top, *resource management* at the bottom, and a *middleware layer* in between. It is in the middleware layer where the integration of disparate information systems takes place.

Advances in middleware technology have been driven by the growing needs of distributed applications such as personalized content delivery [UserLand Software, 2005; QuoteMedia, 2005], online procurement [Ariba, 2005; BEA Systems, 2002], human resource management [Oracle-PeopleSoft, 2005; Taleo, 2005], network and application monitoring [NetLogger, 2002; Ganglia, 2005], etc. These applications usually require a myriad of autonomous systems to be integrated over wide-area networks; accordingly, distributed systems supporting these applications need to be built on flexible and adaptive infrastructures. A variety of middleware infrastructures have been developed to meet the challenges that these applications present.

1.1 Middleware Infrastructures

Traditional middleware infrastructures are *tightly coupled*. Distributed systems built in a tightly coupled way are inflexible, brittle, and cannot adapt to changes in the underlying systems. Tight coupling can occur at two abstract levels in the integration of disparate systems:

- At the lower communication level, tightly coupled systems use static point-to-point connections between senders and receivers (e.g., using *Remote Procedure Call* [Birrell and Nelson, 1984]). This means that a sender needs to know all its receivers before sending a piece of data. Such communication does not scale to large, dynamic systems where senders and receivers join and leave frequently.
- Tight coupling can also occur at the higher content level, often in cases of remote database access. To access a database, an application needs to have precise knowledge of the database *schema* (i.e., its structure and internal data types). Furthermore, the application is at risk of breaking when the remote database schema changes.

It has become clear that tightly coupled infrastructures are inappropriate for modern Internet-based applications [Bosworth, 2002; Carges, 2005]. To bridge the gap between traditional middleware technology and the needs of modern applications, the computing industry has made tremendous efforts to devise new middleware infrastructures. A promising approach is *message-oriented middleware* (MOM) [IBM, 2002; TIBCO Software, 2002; Oracle, 2005] where data to be exchanged is encoded in messages; these messages are moved from senders to receivers through asynchronous queues. Beyond basic message queuing, however, MOM-based platforms have been constantly improving to incorporate advanced features. Of particular importance are the following two technology trends:

- **Publish/subscribe** [Oki et al., 1993]: Publish/subscribe is a many-to-many communication model that directs the flow of messages from senders to receivers based on receivers' data interests. In this model, publishers (i.e., senders) generate messages without knowing their receivers; subscribers (who are potential receivers) express their data interests, and are subsequently notified of the arrival of messages from a variety of

publishers that match their interests. Publish/subscribe has gained acceptance as a solution for the loose coupling of systems at the communication level.

- **Extensible Markup Language (XML)** [Bray et al., 2004]: At the content level, XML is becoming a *de facto* standard for data exchange on the Internet. The reasons for this widespread adoption are twofold. First, XML is flexible, extensible, and self-describing, so it is suitable for encoding data in a *format* (including both structure and content types) that senders and receivers can agree upon. Such XML-based generic formats enable heterogeneous systems to exchange data without knowing how the data is actually represented in the individual systems. Second, a large suite of XML technologies and toolkits developed recently allow system designers to add enhanced functionality as part of data exchange, e.g., message validation and transformation. For these reasons, XML has been recognized as a solution for the loose coupling of systems at the content level.

These two trends lead to *XML message brokering*, the approach that I take in this dissertation for building flexible, high-function distributed information systems.

1.2 XML Message Brokering

XML message brokering is an emerging middleware infrastructure that leverages recent industrial trends. Compared to existing middleware infrastructures such as RPC-based and publish/subscribe-based middleware, XML message brokering has two distinct features:

- It integrates publish/subscribe and XML to provide a high degree of flexibility at both the communication and content levels in distributed information systems.
- It further exploits *declarative* XML queries to increase the functionality of these systems. Declarative XML queries are high-level statements of user interests applied to XML data. These queries can be used to encode user/application-specific logic for handling

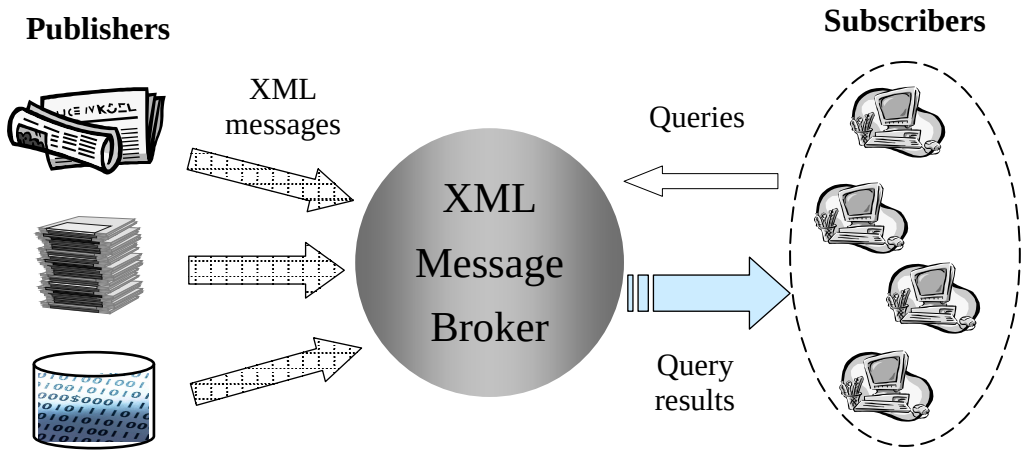


Figure 1.1: Overview of an XML Message Broker

messages in transit (e.g., filtering and transforming them), and can be embedded in the underlying infrastructure of a distributed system to efficiently realize such logic.

1.2.1 XML Message Brokers

In this emerging middleware paradigm, XML message brokers are the key middleware components that serve as central exchange points for messages sent between systems. Figure 1.1 shows the basic context in which a message broker operates. In this figure, the three main components are:

Publishers: Publishers can be many types of information providers such as news agencies, database systems, monitoring applications, etc. They publish information by creating XML messages. An XML message is structured into two main parts: a header and a payload. The header provides application-specific information such as authentication, priority, routing and processing instructions, etc. The payload is the content of the message. Depending on the application, the payload can be a news article, a stock quote, a weather forecast, a technical report, or even a large collection of data items exported from a database.

Subscribers: Subscribers can be end users or applications. They register their data interests with a message broker. For example, a subscriber may be interested in messages

that contain stock quotes of IBM or weather forecasts for San Francisco. These interest specifications can be written in an XML query language such as XPath [Clark and DeRose, 1999] or XQuery [Boag et al., 2003]; specifications written in these languages are essentially declarative XML queries. Subscribers' interest specifications are sometimes also referred to as subscriptions or profiles. In this document, the terms "query", "subscription", and "profile" are used interchangeably.

XML message broker: In its simplest form, a message broker operates as a central server between publishers and subscribers. The two sets of inputs to the broker are queries and continuously arriving XML messages (which are referred to as *streams* of XML messages).

- Inside the broker, arriving queries are stored as *continuous queries* that are applied to all incoming messages. These queries remain effective until they are explicitly deleted.
- Incoming messages are processed on-the-fly against all of the stored queries. For each message, the processing determines the set of queries that are matched by the message; a query result is further created for each matched query. A result can be a copy of the original message or a new, customized message, depending on the requirements of the corresponding query. After a message is processed, all of its query results are delivered to the relevant subscribers in a timely fashion.

Having described the inputs and output of XML message brokers, I now present the core broker functionality. It is important to note that the term "message broker" has been used in many ways in the middleware industry; for example, it is often used to refer to message-oriented middleware that supports the publish/subscribe communication model [BEA, 2002; IBM, 2002]. While such brokers can handle XML-encoded messages and perform simple

routing tasks based on the message header, they do not fully exploit the potential that XML brings to the messaging world.

In this dissertation, I define XML message brokers as middleware components that provide visibility into both the header and the payload of XML messages and that implement application-specific logic for handling such messages. Specifically, given a set of queries and a stream of messages, XML message brokers perform three main functions:

- **Filtering** matches messages against query predicates that represent the data interests of specific subscribers. In a broad sense, query predicates are constraints applied to various fragments of XML messages (a detailed description is provided in Chapter 2.2). For each message, the result of filtering is a set of queries whose predicates are satisfied.
- **Transformation** restructures messages according to query-specific requirements. Besides predicates, a query can also contain a transformation component specifying what fragments to extract from a matching message and how to arrange these fragments in a resulting message. Applying such queries to an incoming message results in a collection of customized messages, one for each query that is matched by the message.
- **Routing** involves transmission of messages over wide-area networks. So far, I have described message brokers in a simple setting; that is, brokers operate as central servers. In scenarios such as Internet-scale data dissemination, however, a network of brokers can be deployed to collaboratively provide the message brokering functionality. In such cases, subscribers register queries and information providers publish messages in a distributed fashion; accordingly, message brokers perform a third function to direct messages from their publishing sites to locations where relevant queries reside and finally to subscribers whose queries have been satisfied.

1.2.2 Example Applications

XML message brokering can be used to support a wide range of emerging applications. In the following, I describe two of them.

Personalized News Delivery: With an increasing supply of information, personalized news delivery has become important to news readers for time savings and better matching of information preferences [Chesnais et al., 1995; Yan and Garcia-Molina, 1999; Yahoo!, 2005]. Today, news providers are adopting XML-based formats to publish news articles online. The *News Industry Text Format* (NITF) [IPTC, 2004] is the most commonly used XML vocabulary among news publishers worldwide, including the New York Times, Agence France Press, and ANSA Italy. Given articles marked up with NITF tags, a personalized news delivery service could allow users to express a wide variety of interests, e.g., "all the sports news", "all the articles written by John Smith", "all the articles referring to the one whose document id is 1234", and "all the events that will take place in San Francisco this weekend". This service could also allow users to specify which portions of the relevant articles, e.g., title and abstract only, should be returned. As soon as a new article is published online, this service delivers the article to all interested users in their required format.

An emerging XML-based technology, *Really Simple Syndication* (RSS) [UserLand Software, 2005], enables similar yet simpler news services: RSS allows news publishers to create updated headlines and delivers these headlines to users according to their URL- and/or keyword-based preferences. In comparison, RSS offers limited personalization functionality by supporting only simple interest specifications and returning results in a fixed headline format.

The desired personalized news delivery services can be built directly using XML message brokering. Users register their interests through a Graphical User Interface that helps users create XML queries and sends these queries to a message brokering system. The system continuously receives XML-encoded news articles from various publishers either by requesting the publishers to push those articles or by using a crawler to fetch the newly published ones. Internally, the system matches the incoming articles against the entire set of queries, transforms the articles into customized results for each matched query, and delivers these personalized results to the relevant users.

Application Integration: A second example application where XML message brokering can play an important role is Application Integration, whose goal is to allow disparate, independently-developed applications to work together. As an example, consider an online quotation system that finds the best price for a product. The system is composed of two types of application: buyers and suppliers. Through a form-based web front end, a buyer creates requests for price quotations for particular products. Each request must be checked against a set of relevant suppliers, each of which offers its price for the product. After the quotations are returned to the buyer, they are compared so that the best price can be reported. There are two difficulties in integrating these two types of application to build the desired quotation system. First, buyers may not have *a priori* knowledge of the set of relevant suppliers. Second, some of the suppliers may be legacy systems that use proprietary formats internally and cannot understand requests encoded in a different format.

These difficulties can be overcome by using a message brokering system and adopting a common XML format for encoding requests for price quotations. Suppliers submit queries to the system describing the categories of products that they provide. If some of the suppliers are legacy systems, their queries also specify how the request messages encoded in the

common format should be transformed to their internal format. On the other side, buyers create request messages and send them to the brokering system. The system matches these messages with suppliers' queries, transforms the messages according to the requirements of the matching suppliers (if necessary), and finally directs the messages in the right format to the relevant suppliers.

Beyond these two application examples, there are many other scenarios where XML message brokering can play a central role. These include online auctions [Ariba, 2005], stock tickers [QuoteMedia, 2005], human resource management [Oracle-PeopleSoft, 2005; Taleo, 2005], network and application monitoring [NetLogger, 2002; Ganglia, 2005], etc. In these scenarios, message brokering functionality would facilitate the development of sophisticated logic for interaction among disparate distributed systems.

1.2.3 Current Industrial Initiatives

Due to its potential for enabling the development of large-scale, high-function distributed applications such as those described above, XML message brokering has drawn increasing interest from the middleware and networking industries. Currently, some of the leading middleware companies are developing brokering functionalities similar to those described above but have not yet deployed them. Also, a number of pioneering networking companies have implemented restricted brokering functionalities in network-oriented settings. In the following, I briefly describe these efforts.

Next-generation message brokers. Middleware providers have accepted that XML is becoming the universal language of Internet data exchange. As a result, leading middleware providers such as Microsoft [Microsoft, 2004] and BEA Systems [BEA Systems, 2005] have initiated efforts to design next-generation message brokers that aim to provide a pipeline of sophisticated operations between inbound and outbound XML messages. Specifically, these

operations validate and reformat inbound messages, e.g., based on the product identifiers contained in the messages, retrieving the corresponding product names from a database and writing the names to the messages; furthermore, these operations filter the messages to determine the set of recipients, transform them to create outbound messages in recipient-specific formats, and finally deliver the resulting messages to the recipients. These message brokers provide a complex flow of operations for message processing and have some aspects of the core brokering functionalities, such as filtering and transformation, as defined in this dissertation.

Application-Aware Network Infrastructures. The XML wave is also heading to the telecommunications world with leading service providers such as Cisco Systems developing Application-Aware Network Infrastructures (AANI). AANI is based on a vision for the transition from packet-oriented networks to intelligent application-oriented networks that consolidate network and application infrastructures to secure, rationalize, integrate, and accelerate applications. Unlike traditional networks that operate on packets or URLs, application-oriented networks operate on entire messages, including all of the content and its semantics. They provide not only connectivity but also message-level access control, filtering, transformation, routing, and many other operations according to business policies.

Compared to the brokering functionalities described in this dissertation, AANI providers have focused on a simpler language for encoding application logic and relied heavily on hardware-based solutions to achieve the performance, simplicity, and security required for wide adoption. Among them, Cisco Systems has recently released its first batch of Application-Oriented Networking Modules [Cisco Systems, 2005] that can be installed in Cisco routers and switches for application-level intelligence, real-time visibility of message content, and cost-effective ownership of consolidated network and application

infrastructures. DataPower Technology [DataPower Technology, 2005] offers newly-developed product lines that enable secure Web services, XML performance improvement, and high speed connectivity among legacy, binary and XML systems. Solace Systems [Solace Systems, 2005] has also debuted with Multiservice Message Routers that provide intelligent routing of application traffic and enhanced operations control that can be beneficial to both applications (e.g., policy enforcement) and network performance (e.g., message prioritization).

These industrial initiatives provide compelling evidence that XML message brokering will become a crucial component of the overall emerging IT (Information Technology) infrastructure. The research presented in this dissertation is of particular relevance to such technology advancement; in fact, important ideas and techniques developed in this research have gained attention and adoption from some of the companies listed above. Compared to the fairly basic functionalities that existing commercial products offer, however, this dissertation aims at higher levels of functionality and thus needs to respond to greater challenges. In the following, I present the technical challenges that this dissertation addresses, describe its unifying themes, and highlight its contributions.

1.3 Challenges

XML message brokering brings many new challenges that have not been addressed in related work, particularly XML query processing, which has been intensively studied in the database literature. The differences between XML message brokering and XML query processing lie in the fact that XML message brokers are deployed on the Internet to integrate widely-dispersed, independently-administrated systems and to support new applications. Such deployment environments raise a number of challenges including:

Scale of processing. A predominant requirement of XML message brokering is scalability. Specifically, message brokering systems must scale along dimensions of message volume, query population, and distribution of the system.

- *Message volume*: The message volume is determined by the number of messages per second arriving at the system and the message size. Depending on the application, the message rate can range up to tens of thousands per second. For example, network and application monitoring systems such as *NetLogger* [NetLogger, 2002] can receive up to a thousand messages per second; *NASDAQ* real-time data feeds [NASDAQ, 2005] include approximately 60,000 messages per second during the market hours. The message size can vary from 1 kilobyte (e.g., XML-encoded stock quote updates) to 20 kilobytes (e.g., XML news articles). To process messages as they arrive, XML message brokers must be able to keep up with such high-volume message flows.
- *Query population*: The query population in a XML message brokering system can also span a wide range, reaching millions of queries for applications such as personalized newspaper generation and mobile operators providing stock quote updates. In the presence of high volumes of messages and large numbers of queries, a key challenge is to efficiently search the huge set of queries to find those that can be matched by a message and construct complete query results for them.
- *Distribution*: Due to the scale of message volume and query population, high-performance XML message brokering may require the use of a network of message brokers to distribute the query population and spread the message processing load. When queries are placed on a large set of brokers, another issue is to quickly identify the set of brokers hosting queries to which a specific message is relevant. Once such brokers are identified, queries on those brokers can be processed as before in parallel.

Robustness in dynamic environments. A second requirement of message brokers is the ability to perform well in highly-dynamic environments. In such environments, subscribers are likely to join and leave and the data interests of existing subscribers may also evolve over time. The frequency of query updates due to these reasons can vary from daily to something much more frequent. As a result, message brokers see a constantly changing collection of queries and need to react quickly to query changes without adversely affecting processing of incoming messages. This issue has been largely ignored by research on XML query processing.

Handling schema-less data. Message brokers receive XML messages from various systems that use internal (not publicly shared) processes to create messages. Even if those systems use the same schema for publication, XML schemas designed for heterogeneous systems are typically general enough that different processes can create messages with significant variability in structure and content. Furthermore, in Internet-based environments, it is not uncommon that message brokers receive XML messages without knowing which schema the messages are based on. These phenomena dictate that XML message brokering cannot be implemented using techniques that rely on the knowledge of schema or the structure of data. As a result, XML message brokering is radically different from traditional database research that depends heavily on such knowledge for query processing and optimization.

1.4 Focus of this Dissertation

After describing XML message brokers and identifying their associated challenges, I now present the focus of this dissertation, including its unifying themes, main technical components, and scope.

1.4.1 Unifying Themes

The previous section identified three challenges in the context of XML message brokering: scale of processing, robustness for query updates, and ability to handle schema-less data. These challenges have shaped the design principles of this dissertation research:

- **Schema-independent query processing**: Message brokers must be implemented using techniques that only require the knowledge of queries for preparing execution plans against which arriving messages are evaluated on-the-fly, but can exploit schemas, if present, to optimize such execution plans for improved performance.
- **Incremental query update**: Message brokers must also be built using techniques that allow execution plans of queries to be updated incrementally, that is, the change of one query does not affect other existing queries.
- **Shared query processing**: XML message brokering systems deployed on the Internet can contain large numbers of queries (as described in Section 1.3); in these systems, significant commonalities among queries are likely to exist. Query processing strategies that ignore such commonalities may perform redundant work that wastes system resources and harms overall performance. Therefore, desirable strategies should be able to exploit such commonalities. To this end, novel techniques need to be devised to identify common portions among queries and effectively share their processing.
- **Reuse of traditional query processing techniques**: XML query processing is complex; efficient processing for a large set of XML queries is even more challenging. On the other hand, there has been thirty years' research on high-performance query processing in traditional (typically, non-XML) database systems. It is crucial to reuse, where possible, traditional query processing techniques in the XML-based message brokering context for simplicity and performance.

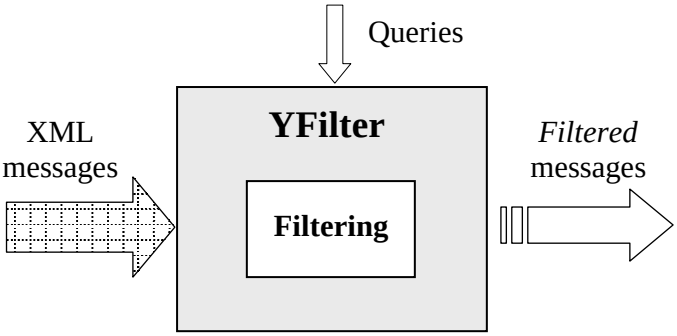


Figure 1.2: Filtering of XML Messages in YFilter

- **In-network query processing**: As the brokering system expands into a network of brokers, routing every incoming message to all the brokers wastes computation power and network bandwidth if the message is irrelevant to the queries on some of the brokers. In such cases, queries can be propagated across the network to bring intelligence to the network routing fabric, to provide flexibility in choosing locations to place broking functionality, and to enhance overall system performance.

1.4.2 Main Components

In this dissertation, I propose, develop, and evaluate YFilter/ONYX, an XML message brokering system that allows users/applications to encode their logic for handling messages using declarative queries and to embed such logic in the underlying infrastructure of an integrated distributed system. The development of this system follows the unifying themes presented in the previous section, and takes place in three major phases:

The initial phase of development focuses on the support for filtering of XML messages. The system developed for this purpose is called *YFilter*. An overview of YFilter is presented in Figure 1.2. Given a set of queries that have been received, for each incoming message, YFilter identifies the subset of queries that are matched by the message. Research issues explored in this phase include shared processing of queries for efficient and scalable filtering,

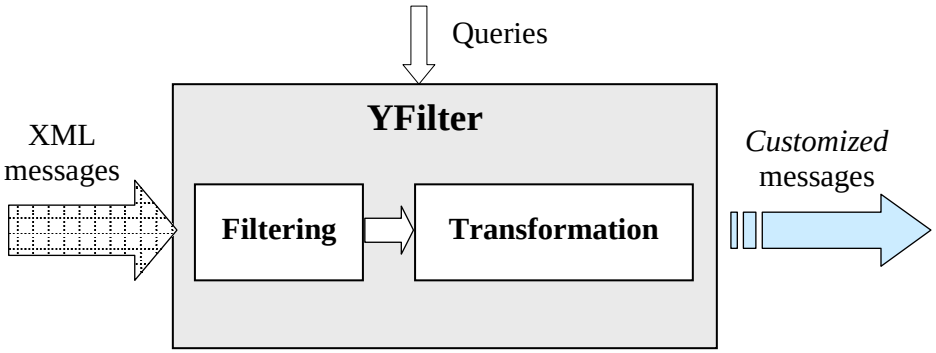


Figure 1.3: Filtering and Transformation of XML messages in YFilter

integrating advanced query features in such shared processing, and incremental maintenance of the system upon query updates. These issues and the complete solutions that YFilter provides for XML filtering are discussed in Chapters 4 and 5.

In the second phase of this research, the YFilter system is extended to also transform messages for customized result delivery. This extension is illustrated in Figure 1.3. As this figure shows, a transformation module is added to YFilter which processes the output of the filtering module and creates customized messages as the final results. The research in this phase is centered on how to support shared processing among transformation queries that incorporate filtering queries as basic components. Details on the transformation extension of YFilter are provided in Chapter 6.

In the third phase of this research, the filtering and transformation functionality of YFilter is extended into a distributed system called *ONYX*. The basic architecture of ONYX is shown in Figure 1.4. ONYX employs a network of message brokers that collaboratively provide high scalability and high functionality. Each message broker in this network runs a YFilter instance to filter and transform messages. In addition, each broker contains a routing component to efficiently forward messages to the downstream brokers that are interested in

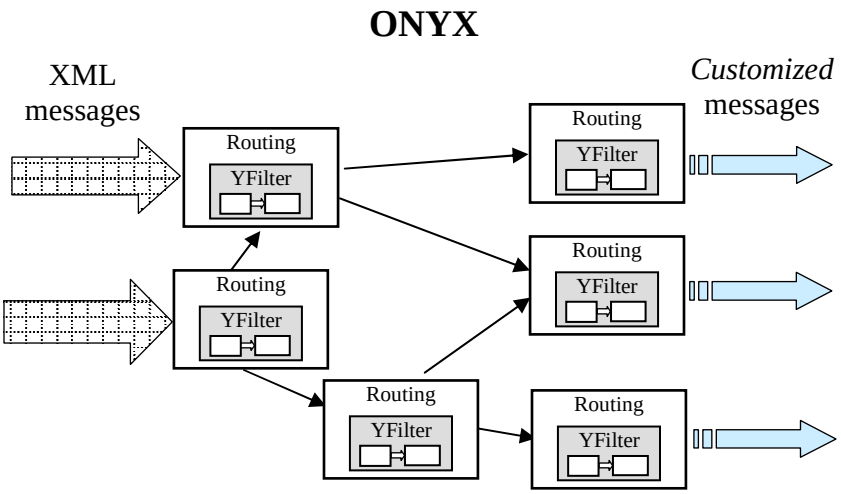


Figure 1.4: Filtering, Transformation and Routing in ONYX

the messages. Interestingly, the routing component is also built on YFilter technology. The ONYX system is presented in detail in Chapter 7.

1.4.3 Scope

XML message brokering is a broad area covering many query processing and architectural issues. This dissertation focuses on the core issues related to providing the brokering functions while meeting the challenges identified in the previous section. To stay focused, several simplifying assumptions are made:

Type of query processing: This research focuses on queries written in standard XML query languages such as XPath [Clark and DeRose, 1999] and XQuery [Boag et al., 2003]. Such queries are applied only to individual messages, for example, to filter and restructure the messages. Message processing with such queries does not involve any interaction across message boundaries, and is referred to as *stateless*. Stateless processing is in contrast to stream query processing [Chandrasekaran et al, 2003; Motwani et al., 2003; Abadi et al., 2003] that maintains *state* over a long stream of messages (as described in Section 3.2.2). Support of such *stateful* processing is a main focus of future work.

Message processing model: Message brokers in this research use a simple execution model: messages are processed one-at-a-time in order of arrival; a message is processed in its entirety before processing is initiated for the next message. Other message processing models can be used. That discussion is beyond the scope of this dissertation.

Range of message routing: Distributed brokering systems route messages from the publishing sites to the locations of relevant subscribers. Message routing in this research covers the range from the publishing sites to the brokers hosting queries relevant to specific messages. The last-hop routing from these query-resident brokers to the subscribers involves a wide variety of communication mechanisms depending on the devices that subscribers use. This issue is a research topic in its own right and is not explicitly addressed in this dissertation.

1.5 Contributions

The main contributions of this dissertation can be summarized as follows.

First, **I identify XML message brokers as key middleware components that perform three main functions on continuously arriving messages on behalf of subscribers: *filtering*, *transformation*, and *routing***. To the best of my knowledge, this work is the first that attempts to address all three issues in the context of XML message brokering.

Second, **for XML filtering, I devise a novel *Nondeterministic Finite Automaton* (NFA)-based approach that aggressively exploits commonalities among queries for shared processing.** By doing so, YFilter provides an order-of-magnitude performance benefit over previous solutions that exploit less or no sharing. It is also highly scalable, supporting up to 100's of thousands of distinct queries with a single processor. Furthermore, it requires only a small maintenance cost for query updates, thus providing a robust solution to XML filtering in dynamic environments.

Third, **for XML transformation, I develop an approach to shared processing of transformation queries that combines YFilter and relational query processing techniques**. The success of this approach hinges on the efficient and scalable foundation that YFilter provides for filtering and an effective use of relational query processing. This is the first algorithm in the literature that supports XML transformation for a large set of queries, i.e., 10's of thousands of them.

Fourth, **I implemented all of the above techniques and evaluated their effectiveness with detailed performance analyses of this implementation**. Results of these analyses demonstrate the efficiency and scalability of YFilter under a wide variety of XML document types and query workloads.

Fifth, **I released YFilter 1.0, a freely available software system containing the filtering engine.** This release has been used in research projects for grid monitoring [GridICE, 2005] and event processing, and has served as an exemplary implementation of such functionality for product-oriented development [Taleo Co., 2005]. Recently, it has also been integrated into Apache Hermes [Apache Hermes, 2004] to provide an implementation of the Web Services Notification specifications [OASIS MSN TC, 2005].

Sixth, **I extend YFilter into a distributed brokering system, *ONYX*, that employs a network of brokers with routing capabilities to provide Internet-scale XML dissemination services**. ONYX pushes queries into the core of the broker network so that intelligent algorithms can be used to route messages and to choose appropriate locations to most efficiently place brokering functionality. This dissertation shows that such in-network query processing enables a distributed brokering system to achieve high scalability as well as high functionality.

Finally, **I propose a number of important open problems in the area of XML message brokering to which my dissertation work has led**. These problems address the needs of emerging applications, such as stateful publish/subscribe and complex event processing.

1.6 Summary

The Information Technology industry is moving towards building large-scale, high-function distributed information systems. Middleware infrastructures play a key role in building such systems. In particular, XML message brokering is emerging as an infrastructure that can meet demanding scalability and functionality requirements. In this chapter, I introduced XML message brokering, identified its main functions and implementation challenges, outlined the key insights for developing solutions that are able to meet such challenges, and summarized the YFilter/ONYX system that provides crucial components for building this new infrastructure.

The remainder of this dissertation presents the YFilter/ONYX system, its core techniques, and results of experimental evaluation of these techniques in greater detail. Chapter 2 provides background on the technical context in which the dissertation research is conducted. Chapter 3 covers related work. Chapters 4 and 5 describe basic filtering and filtering with advanced query support in YFilter, respectively. Chapter 6 discusses YFilter's transformation component. Chapter 7 presents the ONYX extensions to YFilter for distributed brokering, with a focus on routing capabilities. Chapter 8 discusses remaining open issues and sketches directions for future work. Chapter 9 concludes the dissertation.

2 Background

In this chapter, I provide the technical context for the work presented in the subsequent chapters. I begin with a description of the Extensible Markup Language (XML) that is used to encode messages in middleware systems. I then describe the XML query language that is used in YFilter/ONYX for subscribers to write their interest specifications. I also present an overview of XML query processing in both traditional and stream-based settings, which discusses some underlying technologies that YFilter/ONYX uses for developing XML message brokering functionality.

2.1 Extensible Markup Language (XML)

Extensible Markup Language [Bray et al., 2004], abbreviated as *XML*, is a self-describing, flexible, and extensible text format that was originally designed to meet the challenges of large-scale electronic publishing. XML is playing an increasingly important role in the exchange of a wide variety of data on the Internet and in enterprise intranets.

XML describes a class of data objects that are generally called XML documents. XML messages, which were mentioned in the previous chapter, are a special type of XML document. XML provides a mechanism for tagging document content to provide a detailed description of its organization. Specifically, XML allows a document to take a hierarchical structure that consists of a root element and sub-elements; elements can be nested to any depth. Figure 2.1 shows an example XML document that contains a technical report. In this example, the root element is *report*; it contains a sub-element *section* that in turn contains three sub-elements: *title*, *section* (a nested, second-level section), and *figure*.

```
<?xml version="1.0" ?>
<report>
        <section id="intro" difficulty="easy">
                <title>Pub-Sub</title>
                <section difficulty="easy">
                        <figure source="g1.jpg">
                                <title>XML Processing</title>
                        </figure>
                </section>
                <figure source="g2.jpg">
                        <title>Scalability</title>
                </figure>
        </section>
</report>
```

Figure 2.1: An Example XML Document

An XML element starts with a start-tag enclosed by a pair of angle brackets. The start tag consists of a tag name and an optional list of attribute specifications. In the above example, the *report* element does not contain any attributes but its sub-element *section* contains two attributes: *id* with a value "intro" and *difficulty* with a value "easy". An XML element ends with a matching end-tag that is marked by a '/' symbol before the tag name in its enclosing brackets. The content of an element resides between its start- and end- tags, and can contain not only sub-elements but also text data. For example, the first *title* element in Figure 2.1 has the text data "Pub-Sub".

A general set of rules for a document's elements and attributes can be defined in a *Document Type Definition* (DTD) [Bray et al., 2004] or an XML schema [Thompson et al., 2004]. A DTD or XML schema specifies information about a class of documents including all possible structures of documents in the class and the domains of values that attributes in those documents can take. It is important to note, however, that the query processing

techniques described in this dissertation do not require DTDs or XML schemas, but can exploit them, if present, to optimize query processing for improved performance[1].

XML's tagging mechanism and associated technologies for defining rules for such tagging result in three key properties of XML: self-description, flexibility, and extensibility. XML is self-describing because it supports the use of element tags to describe document content. XML is flexible because DTDs and XML schemas allow significant variance in the structure and content of documents; for example, an element, attributes of an element, or text data of an element can be optional, and elements of the same tag name can appear multiple times inside the same enclosing element. Furthermore, XML is extensible because DTDs and XML schemas can be defined and modified by any user. This extensibility is fundamentally different from the *HyperText Markup Language* (*HTML*) [Raggett et al., 1999], which uses a pre-defined, fixed set of tags. It is these three properties that have pushed XML to the forefront of electronic publishing and online information exchange.

2.2 XML Query Language

Having described XML document structure, I now present an XML Query Language that is used in this dissertation to encode subscribers' interest specifications.

XQuery [Boag et al., 2003] is a declarative language for querying XML data. It is designed to be broadly applicable to many types of XML data sources. XQuery is commonly used to locate and extract elements and attributes from XML documents and also to construct

[1] An XML Schema provides richer information than a DTD (e.g., robust and extensible data typing). The information that the work in this thesis exploits for query optimization relates to the structure of documents and is provided by both types of definition. Therefore, XML schema and DTD are treated similarly in this work.

new XML documents using the extracted entities. In this research, I focus on a subset of XQuery for expressing queries over XML messages to filter and transform them.

2.2.1 Path Expressions

A basic, common form of XQuery expressions are *path expressions* that can contain constraints over both structure and content of XML fragments.[2] In this dissertation, path expressions are used to write query specifications for filtering of XML messages.

Path expressions are based on a view in which an XML document is a tree of nodes. Given this view, path expressions are essentially patterns that are matched to nodes in the XML tree. A path expression consists of a sequence of one or more *location steps*. Each location step consists of an *axis*, a *node test* and zero or more *predicates*. An axis specifies the hierarchical relationship between nodes. This dissertation focuses on two common axes: the child axis "/" (i.e., nodes at adjacent levels), and the descendent axis "//" (i.e., nodes separated by any number of levels). In the simplest and most common form, a node test is a name test, which is specified by either an element name or a wildcard operator ("*") that matches any element name.

Each location step can also include one or more predicates to further refine the selected set of element nodes. A predicate, delimited by '[' and ']' symbols, is applied to the element node addressed at a location step. Predicates can specify constraints on the text data or the attributes of the addressed element nodes. In this dissertation, such predicates are referred to as *value-based*. In addition, predicates may also include other path expressions, which are called *nested path expressions*. Nested paths are relative paths with respect to the location

steps where their enclosing predicates reside; accordingly, they are evaluated in the context of each of the element nodes that their enclosing predicates address.

For a concrete example, consider a user who is interested in the title of each figure in a technical report. She can express her interest using Query 1 below (based on a DTD from the XQuery use cases [Chamberlin et al., 2003]). This query specifies that the root element of the document must be *report* and this element must contain a *figure* element somewhere inside (i.e., a "//" location step) which in turn contains a child element *title* (i.e., a "/" location step).

```
Query 1:    $doc/report//figure/title
```

Query 1 is evaluated against each document to which the leading variable $doc is bound. For each document, it returns a query result that contains all the *title* elements matching the entire path expression, listed in their document order. For example, applying this query to the example document in Figure 2.1 creates a result with two matching *title* elements:

```
<title>XML Processing</title>
<title>Scalability</title>
```

For a more complex example, suppose that the user is interested in those sections that are marked as "easy" (thus, suitable for all readers) and contain a title that is "Pub-Sub". She can express these requirements using Query 2 below. This query specifies two constraints on a matching *section* element: (1) its attribute *difficulty* must have a value "easy", which is

---

[2] Path expressions are also defined in the XPath 1.0 specification [Clark and DeRose, 1999] that is largely a subset of XQuery.

specified as a valued-based predicate; and (2) it must have a child element *title* whose text data is “Pub-Sub”, which is expressed using a nested path expression. Note that the ‘@’ symbol in the first predicate indicates that “difficulty” refers to an attribute of the *section* element. In contrast, “title” in the second predicate does not have a preceding ‘@’ symbol, so it refers to a child element of the *section* element.

Query 2: $doc//section[@difficulty = “easy”][title = “Pub-Sub”]

When evaluated against the example document in Figure 2.1, Query 2 returns the top-level *section* element as the result:

```
<section id=“intro” difficulty=“easy”>
    <title>Pub-Sub</title>
    <section difficulty=“easy”>
            <figure source=“g1.jpg”>
                    <title>XML Processing</title>
            </figure>
    </section>
    <figure source=“g2.jpg”>
            <title>Scalability</title>
    </figure>
</section>
```

2.2.2 For-Where-Return Expressions

XQuery also allows customized XML documents to be created using *For-Where-Return* expressions. *For-Where-Return* expressions are a high-level language construct that combines matching and restructuring of XML data. These expressions provide a powerful way to specify requirements for transforming XML messages.

YFilter/ONYX supports a subset of *For-Where-Return* expressions. In this subset, an expression contains:

- A *for* clause containing a *variable name* and a *path expression*; followed by
- An optional *where* clause that contains a set of conjunctive predicates, each of which takes the form of a triplet: a relative *path expression*, an *operator*, and a *constant*; followed by
- A *return* clause that contains interleaved *constant tags* and relative *path expressions*, where all constant tags have a matching close tag.

The semantics of the *For-Where-Return* expression is as follows. The *for* clause creates an *ordered* sequence of variable bindings to element nodes. The *where* clause, if present, restricts the set of bindings passed to the *return* clause. The *return* clause is invoked once for each variable binding. At each invocation of the return clause, tags cause the construction of new element nodes and path expressions select nodes from the current variable binding; if multiple nodes are selected for a path expression, they are grouped and listed in their document order. The final result of the *For-Where-Return* expression is an *ordered* sequence of the results of these invocations.

Continuing with the example from the previous section, the user who issued Query 2 can instead use Query 3 below (whose *for* and *where* clauses together express requirements equivalent to those of Query 2) to transform a matching *section* element to a *new_section* element containing the elements selected from the original *section* using path expressions "/*title*", and "*//figure*".

```
Query 3:  for      $s in  $doc//section[@difficulty = “easy”]
          where    $s/title = “Pub-Sub”
          return   <new_section>
                       {$s/title}
                       {$s//figure}
                   </new_section>
```

Applying Query 3 to the document in Figure 2.1 produces the result below. Compared to the result for Query 2 shown in the previous section, the result here (1) uses a new tag name for the top-level element, (2) includes only a subset of the elements contained in the matching *section* element, and (3) organizes the selected elements in a way such that the *title* elements (note there is only one in this example) are all placed before the *figure* elements, and elements of the same tag name (*title* or *figure*) are listed in their document order.

```
<new_section>
    <title>Pub-Sub</title>
    <figure source=“g1.jpg”>
        <title>XML Processing</title>
    </figure>
    <figure source=“g2.jpg”>
        <title>Scalability</title>
    </figure>
</new_section>
```

So far, I have described XML and a subset of XQuery that are used in this dissertation to encode messages and subscriptions, respectively. Next, I present an overview of processing XQuery queries over XML data, which is referred to as *XML query processing* in the sequel. XML message brokering shares some of the underlying technologies with XML query processing, but also uses more advanced query processing techniques.

2.3 Traditional XML Query Processing

In this section, I describe XML query processing in a traditional setting, that is, where queries are posed against XML data that is persistently stored in a database. There is a large body of work in this area in the database literature. It is not my intention here to provide a thorough survey of this area. Rather, my goal is to provide some basics that are necessary for understanding this dissertation. The interested reader is referred to [Florescu and Kossmann, 2004] for additional information on query processing in XML databases.

As in relational query processing, XML query processing needs a *data model* to describe the data for querying. A widely used XML data model is Infoset [Cowan and Tobin, 2004]. In this model, an XML database is a forest of rooted, labeled trees. Many types of nodes can exist in a tree; the common types include the document node, element node, attribute node, and text node. A document node is the pseudo-root of the tree and points to the top-level element node. An element node corresponds to an element and is labeled with the name of the element. It contains an ordered list of child element nodes and text nodes (in their document order) and an unordered list of attribute nodes. An attribute node is labeled with its attribute name and stores the value of the attribute. A text node simply contains a string of characters that reside between two XML tags. The tree representation for the example document in Figure 2.1 is shown in Figure 2.2, where solid edges are used for ordered nodes and dashed edges are used for unordered nodes. It is important to note that this model is purely conceptual, that is, it is independent of the actual storage structure used in a particular XML database.

I now briefly discuss how an *XML Database Management System* (XDMS) executes an XQuery query. When a query is issued to an XDMS, the compiler of the XDMS parses the

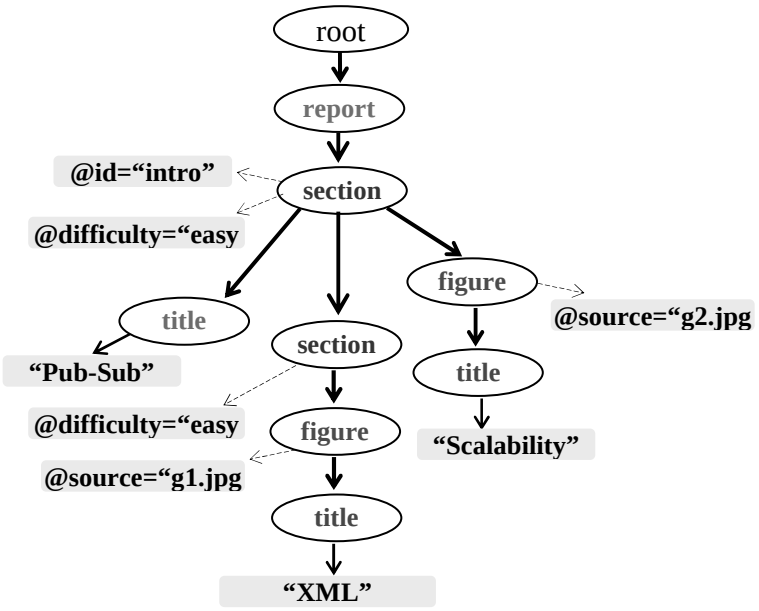


Figure 2.2: A Tree Representation of an XML Document

query and creates an execution plan that determines how to search and process the stored XML data to produce a query result. Then, the runtime system of the XDMS executes the plan to generate the query result. For a concrete example, consider Query 4, a simple path expression with a value-based predicate.

Query 4: $doc/report//section[@difficulty = "hard"]

A simplified execution plan for Query 4 is illustrated in Figure 2.3. As this figure shows, an XML query plan naturally includes navigational access to XML data. The common navigational operations include: (1) *GetChildren* - for each input node, navigate one level down to find the child nodes with a particular name; and (2) *GetDescendant* - for each input node, perform a depth-first search of the subtree rooted at that node to retrieve all the descendant nodes with a particular name. In the plan shown in Figure 2.3, the navigational operations are followed by a selection (σ) that chooses a subset of the input nodes by

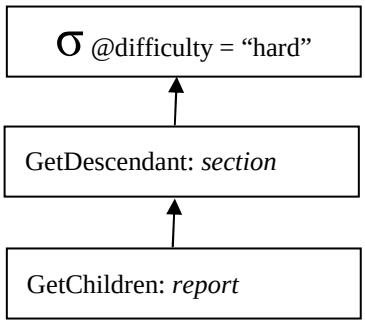


Figure 2.3: A Navigational Query Plan for Query 4

checking the value-based predicate. While navigational access is intuitive and easy to implement, it is often inadequate for efficient query processing. In cases when a tree has a large structure of nodes, traversal all the way from the root is inefficient for finding a few matching nodes that reside deeply in the tree.

An XDMS solves this problem by building indexes over nodes in an XML tree and using index-based access to speed up query processing. Many types of indexes have been proposed [McHugh and Widom 99; Zhang et al., 2001; Jagadish et al., 2002; Halverson et al., 2003]. An example is the element index (E-index). An E-index records the occurrences of an element name inside a collection of documents. Assume that an XML document is parsed to a sequence of items that are either a tag or a text word. An occurrence of an element name is then indexed by (1) its document number, (2) its position in the document, specified by the positions of its start- and end- items, e.g., starting at the 2^{nd} item and ending at the 19^{th} item, and (3) its nesting level in the document, e.g., 2 levels from the root. An E-index is sorted in increasing order of the document number, and then in increasing order of start- and end-items. It is obvious that E-indexes can be used to quickly retrieve all occurrences of a particular element name. In addition, they can be used to evaluate the containment relationships specified by the axes: They allow “A//B” to be evaluated by checking the

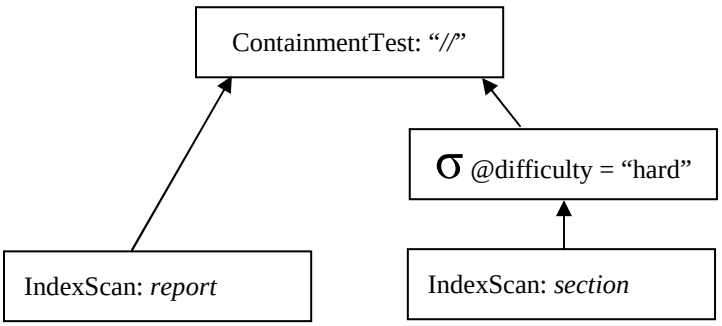


Figure 2.4: An Index-based Query Plan for Query 4

positional containment between these elements, i.e., if the start-item of A occurs before that of B and the end-item of A after that of B. "A/B" can be evaluated by further checking if B occurs one level below A.

Let us revisit Query 4. An E-index, when present, enables a query plan such as that shown in Figure 2.4. At the bottom, this plan uses the E-Index to find all of the occurrences of *report* and *section*. After filtering *sections* using the value-based predicate, it then evaluates the "//" containment between *report*s and *section*s using the positional information that these occurrences carry. Compared to the navigational plan in Figure 2.3, the index-based plan avoids sequential search of the entire document and can provide a significant performance gain when the document is large and the number of *sections* is relatively small.

I end this brief discussion of query processing in XML databases by highlighting two key features of such processing: (1) ***query-initiated execution***, that is, the execution of a query is triggered by the arrival of the query, and (2) ***index-based data access,*** that is, indexes are built over data and used to speed up query processing. In the next section I show how query processing in other environments can differ remarkably from query processing of this type.

2.4 Stream-based XML Query Processing

In an emerging paradigm for XML query processing, XML data continuously arrives from external sources, and queries are evaluated every time when a new data item is received. Such XML query processing is referred to as *stream-based*. A distinctive feature of stream-based processing is the ability to process data as it arrives. This is a natural fit for XML message brokering where messages need to be filtered and transformed on-the-fly. Furthermore, in cases where incoming messages are large, stream-based processing also allows query execution to start long before those messages are completely received, thus reducing the delay in producing results. The work presented in this dissertation is conducted in the stream-based setting.

2.4.1 Event-Based XML Query Processing

Stream-based XML query processing can be performed at the granularity of a document or a smaller constituent piece of a document. Some of the earlier *Continuous Query* systems such as *NiagaraCQ* [Chen et al., 2000] execute queries when new documents arrive. Documents in these systems are simple and small, e.g., stock quote updates and event notifications. As XML gains popularity in a wide range of applications, XML documents have been used for encoding data of diverse types (e.g., astronomy data [NASA, 2003] and biological data [SECSG, 2004]) and immense sizes (e.g., the equivalent of a database's worth of data). To provide efficient processing also for such large documents, more recent systems such as *XFilter* [Altinel and Franklin, 2000], *Tukwila* [Ives et al., 2002], and the *BEA Stream Processor* [Florescu et al., 2004] support fine-grained query processing upon arrival of a start-tag, end-tag, or text data of an element.

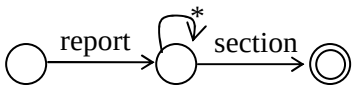
```
< Start Document
< Start Element:        report
< Start Element:        section
< Start Element:        title
  Characters:   Pub/Sub
> End  Element:        title
< Start Element:        section
< Start Element:        figure
< Start Element:        title
  Characters:   XML Processing
> End  Element:        title
> End  Element:        figure
> End  Element:        section
  …
> End  Element:        section
> End  Element:        report
> End Document
```

Figure 2.5: An Example of the SAX API

Fine-grained XML query processing can be implemented via event-based APIs. A well known example is the *SAX* interface [Megginson, 2000], which reports low-level parsing events incrementally to the calling application. Figure 2.5 shows an example of how a SAX interface breaks down the structure of the sample XML document from Figure 2.1 into a linear sequence of events. “Start document” and “end document” events mark the beginning and the end of the parse of a document. A “start element” event carries information such as the name of the element and its attributes. A “characters” event reports a text string residing between two XML tags. An “end element” event corresponds to an earlier “start element” event and marks the close of that element. To use the SAX interface, the application receiving the events must implement handlers to respond to different events. In particular, stream-based XML query processors can use these handlers to implement event-driven query processing.

/ report // section

(a) path expression



(b) a corresponding FSM

Figure 2.6: A Path Expression and its Corresponding FSM

2.4.2 A Finite State Machine-Based Approach

A popular approach to event-driven XQuery processing has been to adopt some form of *Finite State Machine* (FSM) to represent path expressions [Altinel and Franklin 2000; Ives et al., 2002]. This approach is based on the observation that a path expression written using the axes ("/", "//") and node tests (element name or "*") can be transformed into a regular expression. Thus, there exists an FSM that accepts the language described by such a path expression [Hopcroft and Ullman 1979].

Both XFilter [Altinel and Franklin 2000] and Tukwila [Ives et al., 2002] create an FSM for each path expression by mapping the location steps of the path expression to machine states. Figure 2.6 shows an example FSM created for a simple path expression, where the two concentric circles represent the *accepting state* of this FSM. Arriving XML documents are then parsed with an event-based parser (e.g., a SAX parser). The events raised during parsing are used to drive the execution of query FSMs; in particular, "start element" events drive the FSMs through their various transitions, and "end element" events cause the FSMs to backtrack. A path expression is said to match a document if during parsing, the accepting state for that path is reached.

2.4.3 Indexing of Queries

As stream-based systems are deployed on the Internet, it is likely that the set of queries will be large. This is particularly true with *XML filtering* systems [Altinel and Franklin 2000; Diao et al, 2002; Chan et al., 2002; Green et al., 2003] that usually evaluate hundreds of thousands of path expressions against incoming XML documents. In such cases, it is prohibitively expensive to use a brute force approach that iterates over the query set and executes them one at a time.

Researchers from a number of projects have observed that XML filtering is essentially the inverse problem of querying a database. In a traditional database system, a large set of data is stored persistently. Queries, coming one at a time, search the data for results. Indexes enable the data to be searched without having to sequentially scan it. In an XML filtering system, a large set of queries is persistently stored. Documents or their parsing events, coming one at a time, drive the matching of the queries. Accordingly, indexing the queries can enable selective matching of documents to queries.

Based on this insight, an important optimization for XML filtering has been to build an index over the queries, and to use the parsing events of a document to probe the query index. This approach quickly results in a smaller set of queries that can be potentially matched by the document. As such, significant work can be saved by avoiding processing queries for which the document is irrelevant. XFilter was the first filtering system to exploit this idea. It builds a dynamic index over the states of query FSMs; this index identifies the states that the FSM execution is attempting to match at a particular moment. The content of the index constantly changes as parsing events drive the execution of the FSMs (see [Altinel and Franklin, 2000] or Appendix A of this dissertation for details).

XFilter, however, is still subject to scalability problems as it ignores another important opportunity for optimization – sharing in query processing. In large-scale filtering systems, significant commonalities are likely to exist among queries. By creating an FSM per query, XFilter fails to exploit such commonalities, thus performing redundant work. In contrast, this dissertation research proposes advanced query indexing structures in which common query fragments share their representation in the query indexes and thus are processed at most once.

I conclude the discussion on stream-based XML query processing by summarizing its two key features: (1) ***data-driven, incremental processing***, that is, query processing is driven by the arrival of data and, in particular, is performed incrementally upon arrival of fine-grained XML parsing events; and (2) ***indexing over queries,*** which allows selective matching of incoming documents to a large set of queries.

2.5 Summary

In this chapter, I provided technical background for the XML message brokering functionality presented in this dissertation. I described the Extensible Markup Language (XML) for encoding messages and a subset of the XQuery query language for specifying filtering and transformation requirements. I also presented an overview of XML query processing in both traditional and stream-based settings, which covered XML technologies such as the XML data model, event-based parsing, Finite-State Machine-based approaches, and query indexing. These techniques provide a foundation for XML message brokers.

3 Related Work

In this chapter, I provide a survey of academic and industrial work related to XML message brokering. I first describe the design space for XML message brokers and show how the unexplored areas of the space motivate the research described in this dissertation. I then present a broad overview of other work related to the architectural and query-processing issues of XML message brokering. In later chapters, more detailed comparisons are made to particularly relevant research.

3.1 Design Space for XML Message Brokering

In this section, I present a design space for XML message brokering, including relevant industrial products and research projects, and describe the position of this dissertation in relation to those efforts.

Figure 3.1 shows a diagram illustrating the design space. The diagram consists of two dimensions. The Y-axis relates to the *style of processing*. In a coarse-grained fashion, this design space considers centralized and distributed processing. Distributed processing spreads the processing load and has the potential for truly Internet-scale message brokering services; not surprisingly, distributed processing requires more sophisticated processing techniques. The X-axis relates to the *expressiveness* of message brokering services, which is determined by the data model and query language that the message brokers support. From the least to the most expressive, there are four categories.

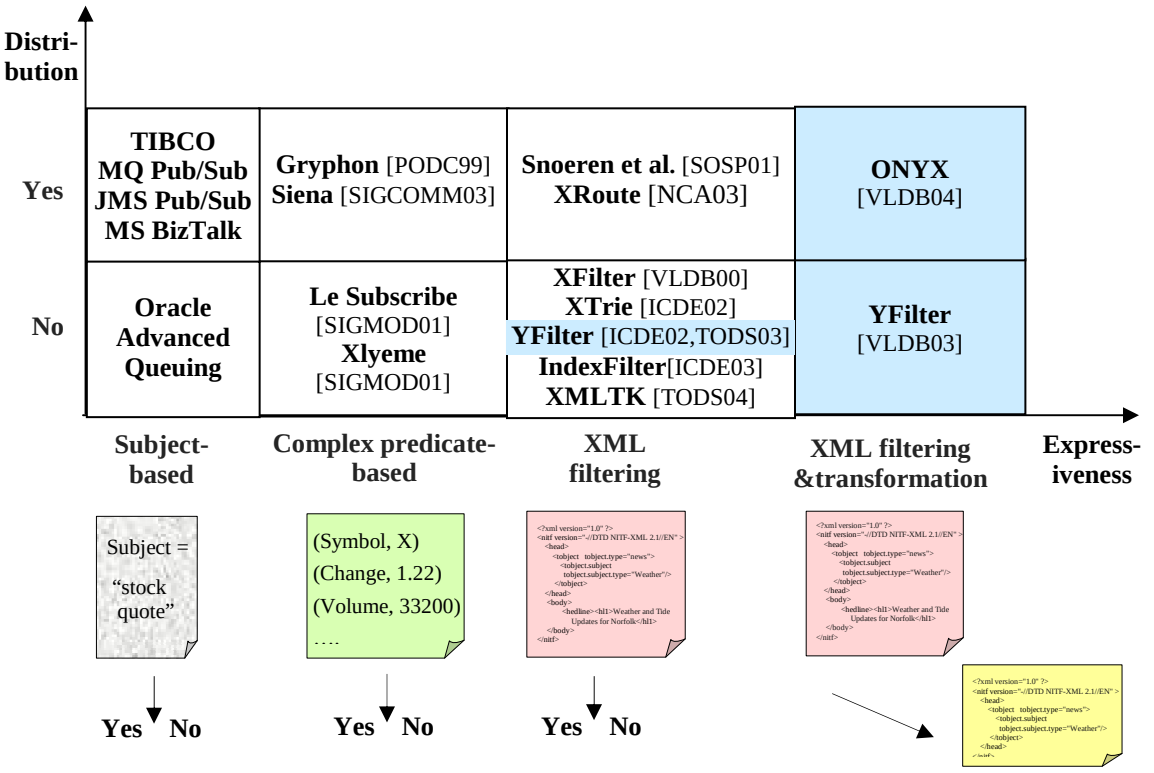


Figure 3.1: Design Space for XML Message Brokering

Subject-based: In subject-based systems, publishers label each message with a subject from a pre-defined set or hierarchy. For example, the subject of a message can be "stock quote" or "sports/golf". Users subscribe to the messages having a specific subject; that is, user queries specify a subject of interest. In these queries, users can also apply a selector to the message header (e.g., a conditional expression over the data fields in the header [Sun Microsystems, 2002]) to refine the set of relevant messages within a particular subject. A query result is a "yes" or "no" answer, indicating if a message has matched the query. In the positive case, the message is delivered to the corresponding user. The expressiveness of subject-based systems is restricted by the opaqueness of the message content; that is, these

systems simply treat message payloads as text strings without exploiting the richness of content that XML provides.

Most commercial publish/subscribe systems are subject-based. Among them, the widely-used ones include TIBCO Rendezvous [TIBCO Software, 2002], MQ Series publish/subscribe [IBM, 2002], JMS publish/subscribe [Sun Microsystems, 2002], Microsoft BizTalk Server [Microsoft, 2004], and Streams Advanced Queuing [Oracle, 2005]. Some of them use centralized computing [Oracle, 2005], as they are built on a database system and provide a unified interface for applications to access a message queue or a database. Others support distributed processing [TIBCO Software Inc., 2002; IBM, 2002; Sun Microsystems, 2002; Microsoft, 2004], as such systems are designed to integrate widely dispersed information providers and consumers that belong to different administrative domains.

Complex predicate-based: In the second category, publish/subscribe systems model message content as a set of attribute-value pairs. A stock quote represented in this model is illustrated in the lower part of Figure 3.1. These systems allow user queries to contain a set of predicates connected using “and” and “or” operators to specify constraints over values of the attributes. More specifically, a predicate is a comparison between an attribute and a constant using relational operators such as equality (‘=’), greater-than (‘>’), or less-than (‘<’). For example, a predicate-based query applied to the stock quote shown in Figure 3.1 can be “Symbol=‘X’ and (Change > 1 or Volume > 50000)”. Query answers are still “yes” or “no”, resulting in relevant messages being delivered to their interested users.

There have been a number of research projects on complex predicate-based message filtering. *Le Subscribe* [Fabret et al., 2001] and *Xlyeme* [Nguyen et al., 2001] are predicate-based systems that use centralized processing. They employ sophisticated filtering algorithms

based on indexing predicates and clustering queries according to the common constituent predicates. *Gryphon* [Aguilera et al., 1999] and *Siena* [Carzaniga and Wolf, 2003] are distributed predicate-based systems that aggregate user queries into compact, precise in-network data structures that can be used to efficiently route messages to relevant systems or other nodes in the network.

XML filtering: The third category of message brokers starts to exploit the richness of XML-encoded messages. Here, message content can take a hierarchical structure, as illustrated in Figure 2.1 from Section 2.1, which encompasses repetition of element names (e.g., a purchase order contains multiple line items), recursion of elements (e.g., a section contains a nested section), and attributes and text data. User queries are written using path expressions, a small yet common subset of XQuery as described in Section 2.2.1. Query answers are "yes" or "no" as before. By combining rich XML structure and path expressions, XML filtering provides greater expressiveness in specifying data interests, resulting in potentially more accurate filtering of messages.

A large number of solutions to XML filtering have been developed in the database community. These solutions emphasize the efficient processing of a set of queries and are typically centralized. *XFilter* [Altinel and Franklin, 2000] and *Index-Filter* [Bruno et al., 2003] construct indexes over the queries. *YFilter* [Diao et al., 2002; Diao et al., 2003], which is presented in Chapters 4 and 5 of this dissertation, *XTrie* [Chan et al., 2002], and *XMLTK* [Gupta and Suciu, 2003; Green et al., 2004] also exploit the commonalities among queries. These solutions are discussed and compared in Chapters 4 and 5.

While the database community has focused on centralized solutions to XML filtering, the networking community has explored solutions in distributed environments. The distributed

solutions focus on networking issues such as in-order-delivery and network resilience [Snoeren et al., 2001] or minimizing space requirements at routers [Chand et al., 2003]. For query processing, however, they use either a general-purpose XML toolkit to process queries one at a time or an XML filtering approach (e.g., XTrie) that is not designed to handle demanding query workloads. As a result, these solutions cannot scale for the large user communities that many Internet-based applications such as stock tickers [NASDAQ, 2005] and personalized news delivery [UserLand Software, 2005] are facing.

XML filtering and transformation: The last category, XML filtering and transformation, extends the previous one by also transforming messages to provide customized results. Such transformation is needed for application integration, personalization, and adaptation to wireless devices (recall the example applications given in Section 1.2.2). To support message transformation, queries are written using a richer subset of XQuery, in particular, the *For-Where-Return* expressions as described in Section 2.2.2.

YFilter provides the first algorithm in the literature that can support transformation for large numbers of queries. This topic is discussed in Chapter 6. ONYX further extends YFilter by incorporating filtering and transformation functionality into distributed computing, thus gaining substantial benefits such as aggregated bandwidth and computing power. ONYX is presented in detail in Chapter 7.

3.2 Other Related Work

The design space presented in the previous section covers the systems and projects most relevant to XML message brokering. Besides those, this dissertation is also related to a large body of research work in the Information Retrieval (IR), database, networking, and

programming language communities. I now briefly survey the relevant research in these areas, focusing on the work that has not been discussed previously.

3.2.1 Information Retrieval

The modeling and matching of user profiles have been extensively investigated in the context of Information Filtering and Selective Dissemination of Information research, e.g., [Foltz and Dumais 1992]. User profiles in these scenarios are intended for unstructured, text-based systems and typically use sets of keywords to represent user interests. In general, IR profile models can be classified as either *Boolean* or *similarity-based*. The former model is based on an exact match semantics with profiles consisting of keywords connected by Boolean operators. The latter model uses a fuzzy match semantics, in which a similarity value is assigned to every (document, profile) pair. A document with similarity to a profile over a certain threshold is said to match the profile [Salton 1989; Belkin and Croft 1992; Cetintemel et al. 2000].

The *Stanford Information Filtering Tool* (SIFT) [Yan and Garcia-Molina 1994; Yan and Garcia-Molina 1999] is an Internet news filtering system that supports both profile models. SIFT builds keyword-based indexes over profiles and uses these indexes to match incoming documents with the profiles. It also provides several simple schemes to spread the filtering work to a cluster of machines so that bottlenecks and critical failure points can be avoided.

YFilter/ONYX differs from keyword-based filtering in that it is targeted at application domains in which data is encoded in XML and user queries take advantage of the rich semantic and structural information embedded in the data for more precise filtering and result customization. In addition, ONYX addresses the data dissemination problem in

distributed environments and offers scalable services by using intelligent algorithms for routing and incremental processing of messages.

3.2.2 Database Technologies

The past decade has witnessed significant changes in research directions in the database community; of particular relevance to YFilter are the shift from traditional database processing to Internet-oriented query processing and the surge of XML-based research. I examine the related efforts in the following.

Continuous Query Systems. *Continuous Queries* (CQ) are standing queries that allow users to get new results whenever an update of interest occurs. Such queries are especially useful in Internet-based environments comprising large amounts of frequently changing information. Much of the CQ research [Terry et al. 1992; Liu et al. 1999; Chen et al., 2000; Chen et al., 2002; Madden et al., 2002] has focused on stateless query processing (as described in Section 1.4.3) in a relational setting; that is, updates are relational tuples and queries use simple relational operators, mostly selections, to filter these updates. Due to the relational model used, these systems do not address the challenges of matching constraints over the structure of data. Some of the CQ techniques, however, can be applied to XML message brokering, if the latter can be mapped to a relational domain. Chapter 6 presents such a mapping from XML transformation to relational processing, which enables YFilter to use CQ techniques for improved performance and scalability.

Triggers. Triggers [Stonebraker, 1990; Widom and Finklestein, 1990; Schreier et al., 1991] in traditional database systems are similar to CQ. However, triggers are a complex mechanism that can involve predicates over many data items and can initiate updates to other data items. Thus, trigger solutions are typically not optimized for fast matching of individual

items to vast numbers of relatively simple queries. Some more recent work, however, has addressed the issue of scalability for simple triggers by grouping predicates into equivalence classes and using predicate indexing techniques [Hanson et al 1999]. This work has not addressed the XML-related issues, such as matching constraints over structure and integrating structure matching with value-based predicate evaluation, that YFilter handles.

Relational Stream Management Systems. There has been a significant amount of activity on the topic of handling continuous, rapid, and time-varying tuple streams, which results in the development of a number of stream management systems including *TCQ* [Chandrasekaran et al, 2003], *STREAM* [Motwani et al., 2003], and *Aurora* [Abadi et al., 2003]. These systems support complex continuous queries that compute aggregate values over a period of time called a *window* (i.e., stateful processing as described in Section 1.4.3). Their associated research explores a set of key issues including adaptivity [Chandrasekaran et al, 2003], approximation [Motwani et al., 2003], shared processing [Chandrasekaran et al, 2003; Motwani et al., 2003], and Quality-of-Service [Abadi et al., 2003]. These systems differ from YFilter in that they perform stateful processing but only for relational tuple streams. In contrast, YFilter supports processing over individual messages (i.e., stateless) and focuses on XML-related challenges such as structure matching and message transformation. How to extend YFilter/ONYX to also support computation across message boundaries is a main direction of future work and is further discussed in Chapter 8.

Stream-Based XQuery processors. XQuery evaluation has been studied in the context of continuously arriving XML parsing events (as described in Section 2.4). XQuery processors developed in this setting use a variety of techniques to achieve efficiency, e.g., FSM-based matching of path expressions [Ives et al., 2002], transducer-based processing of

For-Where-Return expressions [Ludascher et al., 2002], and pipelined execution with lazy evaluation [Florescu et al., 2004]. There systems, however, focus on processing of individual queries, and thus are not suitable for large-scale XML filtering and transformation.

3.2.3 Networking Technologies

The ONYX system presented in this dissertation is a distributed XML brokering system that extends YFilter's filtering and transformation functionality with routing capabilities. ONYX is related to many systems that have been developed in the networking community; the connection hinges on the common interest in large-scale data dissemination from a myriad of information providers to large numbers of receivers. Although networking research generally does not exploit declarative queries, the trend is clear: information processing systems and networking systems are converging to solve the Internet-scale data dissemination problem. ONYX represents one effort towards such convergence.

Multicast. Multicast allows a source to send the same content to multiple receivers. Though bandwidth-efficient, IP multicast [Ballardie et al., 1993; McCanne et al., 1996] is hard to deploy because of being a network layer paradigm. This has lead to application-layer solutions like *Overcast* [Jannotti et al., 2000] and *i3* [Stoica et al, 2002]. Proposals have also been made to augment IP multicast with content-based routing features [Opyrchal et al., 2000; Shah et al., 2002]. However, none of this work gives the user fine-grained ways of specifying their interests, like a powerful query language over XML.

Content Distribution Networks (CDN). CDNs provide an infrastructure that delivers static or dynamic Web objects to clients on request from nearby Web caches or data replicas [Dilley et al., 2002; IBM, 2005], thus offloading the main website. Recent work has focused on allowing the user to specify coherence requirements over data [Shah et al., 2003]. This

differs from ONYX, as it deals with content delivery upon request, rather than continuous query processing over streaming messages, and it does not give the user a powerful query language to specify her interests.

Distributed publish/subscribe systems. Distributed publish/subscribe systems support the complex predicate-based model (as described in Section 3.1) and provide many-to-many communication between publishers and subscribers. These systems construct routing tables from user queries to efficiently forward packets [Carzaniga and Wolf, 2003; Aguilera et al., 1999; Banavar et al., 1999], and also compare such routing to alternative schemes with varying query properties [Opyrchal et al., 2000; Mühl et al., 2002]. Many of the results reported can be applied in the context of ONYX. In comparison, ONYX addresses a more challenging problem, as support for rich XML messages and queries leads to increased complexity of routing table construction, data forwarding, and query processing.

3.2.4 Programming Languages

YFilter/ONYX is also related to a family of programming languages that deal with communication and object sharing among distributed entities. Much of the work centers around the Linda programming language developed at Yale University and ideas that have been spawned from the work on the Linda system.

Linda [Carriero and Gelernter, 1989; Gelernter and Carriero, 1992] is a parallel programming extension to popular programming languages, such as C (C-Linda) and Fortran (Fortran-Linda). It is based on a logically global, associative memory called the *tuple space* in which clients can read and write objects called tuples. A tuple contains a set of attributes that can be used for clients to selectively choose which objects to access. As such, tuple

spaces provide inter-process communication and synchronization that is logically independent of the underlying computer or network architecture.

Tuple spaces have many commercial implementations with various improvements. Sun Microsystems adapted many of the ideas behind Linda to their Java environment. The resulting technology, called *JavaSpaces* [Sun Microsystems, 2000], is a unified platform-independent mechanism for communication, coordination, and sharing of objects between Java technology-based network resources. *TSpaces* from IBM combines sophisticated database functionality with communication middleware [IBM, 2000]. It is essentially a set of network communication buffers (similar to tuple spaces) with database capabilities for reliable storage (but without the use of complex SQL queries).

The idea of loosely coupling applications has been taken a step further by combining the notion of tuple spaces with that of self-describing XML documents. *XML Tuple Spaces* [Rogue Wave Software, 2004] allow developers to take advantage of the loosely coupled nature of XML, the use of Web services for communication between remote locations, and the ability to search for data using a simple subset of an XML query language.

The programming languages and systems described above differ from YFilter/ ONYX in two main aspects. First, they use a traditional request-based communication model; they cannot dynamically push information to clients based on their specified interests, which is a fundamental requirement for XML message brokers. Second, they do not provide a powerful language for filtering and transformation.

3.3 Summary

In this chapter, I discussed academic and industrial work related to the YFilter/ONYX system presented in this dissertation. The discussion shows that none of that work completely

addresses the challenges that arise in the context of large-scale, high-function XML message brokering. In particular, relational query processing techniques do not address XML-related issues such as matching constraints over structure and message transformation; XML query processing systems do not scale along both dimensions of query population and distribution; Networking solutions have generally not exploited the rich content of XML messages or incorporated efficient processing of user queries in the routing paradigm; Programming languages for distributed object sharing use a communication model inadequate for XML message brokering and lack a powerful query language.

4 Basic Filtering with YFilter

In this chapter, I describe the first technical component of my dissertation research, namely, the YFilter approach to XML filtering. This chapter explores sharing in matching the basic structure of queries. The discussion of how to extend such shared processing to handle advanced query features including value-based predicates and nested path expressions is postponed until the next chapter.

4.1 Introduction

As described in Chapter 2, XML filtering systems provide fast, on-the-fly matching of XML documents to large numbers of query specifications. Queries in these systems usually contain path expressions that can be used to specify constraints over both the structure and content of XML documents. Such queries are referred to as *path queries* in this dissertation. A formal description of the filtering problem is the following:

> *Given (1) a set Q = Q, …, Qn of path queries, where each Qi has an associated query identifier, and (2) a stream of XML documents, compute, for each document D, the set of query identifiers corresponding to the path queries that are matched by D (i.e., a non-empty result can be returned for each of these queries).*

It is important to note that XML filtering returns a set of query identifiers for each incoming document; in other words, it provides a Boolean result for processing a document against a query. A Boolean result can be created for a query as long as a single matching element can be identified from the document being processed. Thus, complex issues in XQuery processing related to multiple matches are not relevant here, which enables

simplified, high-performance query processing. These issues, which dramatically increase the complexity of query processing, are considered for XML transformation in Chapter 6.

This chapter focuses on the first main challenge of XML filtering: the efficient and scalable matching of the structure of path expressions, which lays the foundation for high-performance XML filtering. As described in Chapter 3, XFilter [Altinel and Franklin 2000], the first published XML filtering system, has shown that an approach using event-based parsing and Finite State Machines (FSMs) can provide the basis for structure-oriented matching of path expressions. By creating a separate FSM per path query, however, XFilter could perform redundant work. This is especially true in large-scale filtering systems where significant commonalities among queries are likely to exist.

Based on this insight, YFilter uses a novel approach that exploits such commonalities by using a single, combined FSM to represent all path expressions. The combined FSM naturally supports the sharing of processing for all common prefixes among path expressions. Furthermore, the combined FSM is implemented as a *Nondeterministic Finite Automaton* (NFA). The NFA-based implementation has several practical advantages including: 1) a relatively small number of machine states required to represent even large numbers of path expressions, 2) the ability to support complicated document types (e.g., with recursive nesting) and queries (e.g., with multiple wildcards and descendent axes), and 3) incremental maintenance of the machine upon query updates.

To investigate the impact of shared path matching, a detailed performance study has been conducted using XFilter, YFilter, and a hybrid approach that does more sharing than XFilter but less than YFilter. The results of this study show that YFilter's shared path matching approach can provide order-of-magnitude performance improvements over XFilter and the

hybrid approach while preserving the flexibility to support a wide variety of document types and query workloads

The remainder of this chapter is organized as follows. Section 4.2 describes the architecture of the YFilter filtering system. Section 4.3 describes the NFA-based path matching algorithm. Section 4.4 presents a detailed performance analysis. Section 4.5 covers work related to specific techniques presented in this chapter. Section 4.6 concludes this chapter.

4.2 Architecture of the Filtering Engine

The architecture of the YFilter filtering system is shown in Figure 4.1. The primary inputs to the engine are user queries and XML messages. For each incoming message, the output consists of a set of identifiers indicating the users to which the message should be delivered. The basic components of the YFilter filtering system are:

- **XQuery parser**: The XQuery parser parses arriving queries and sends the parsing results to the compiler.
- **Query Compiler**: The compiler constructs an execution plan for each arriving query and merges this plan with a global query plan by sharing the common portions of these plans. For filtering purposes, the global query plan primarily contains an NFA representing all of the path queries.
- **XML parser**: Each incoming XML message is run through an event-based XML parser (e.g., a SAX parser). The parser produces parsing events and passes them to the runtime system to drive query execution.

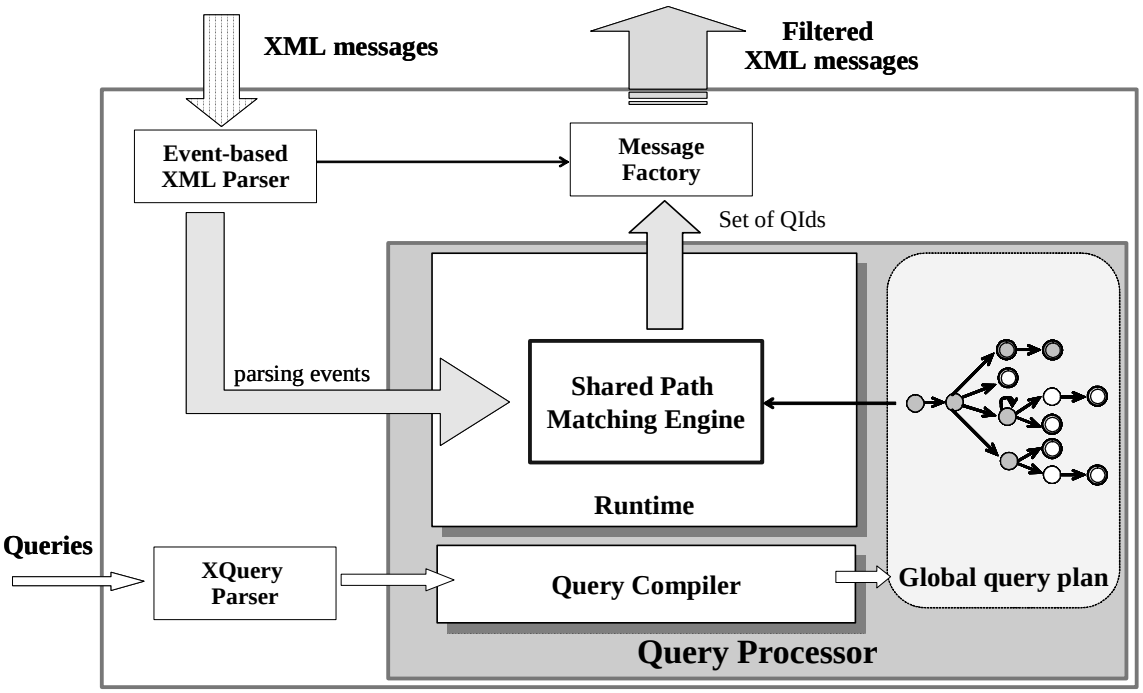


Figure 4.1: Architecture of the YFilter Filtering System

- **Runtime**: For XML filtering, the runtime system uses a component, called a shared path matching engine, to match incoming messages with the NFA-based representation of path queries. When processing a message, this engine uses the XML parsing events to drive the NFA through its various transitions; in this way, it matches all path queries in a shared fashion. As mentioned above, this engine returns a set of identifiers indicating the queries that are matched by the message.
- Message factory: Query processing results are fed to the message factory that prepares messages for final delivery.

The query compiler and the shared path matching engine constitute the core processor of YFilter. In the following sections, I focus on techniques used in these two components.

4.3 Shared Structure Matching

In this section, I describe the YFilter approach to structure-based matching for large numbers of path queries.

4.3.1 Query Representation: A Combined NFA with an Output Function

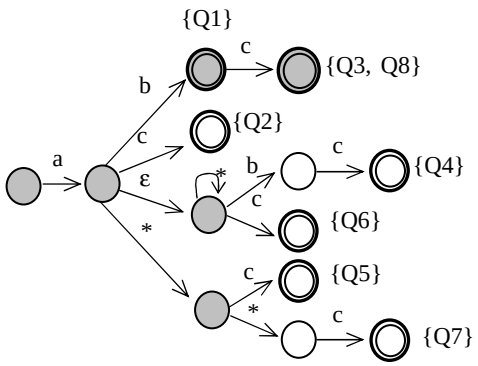
YFilter uses a novel approach that identifies commonalities among path queries and shares the processing among them. In this approach, rather than representing each path query as an FSM individually, YFilter combines all queries into a single FSM in the form of a *Nondeterministic Finite Automaton* (NFA). The NFA has two key features: (1) there is one accepting state for each path query and (2) the common prefixes of the paths are represented only once.

Figure 4.2 shows an example of such an NFA representing eight path queries. A circle denotes a state. Two concentric circles denote an accepting state; such states are also marked with the IDs of the queries they represent. A directed edge represents a transition. The symbol on an edge represents the input that triggers the transition. The special symbol "*" matches any element. The symbol "ε" is used to mark a transition that requires no input. In the figure, shaded circles represent states shared by queries. Note that the common prefixes of all the queries are shared. Also note that the NFA contains multiple accepting states. While each query in the NFA has only a single accepting state, the NFA represents multiple queries. Identical (and structurally equivalent) queries share the same accepting state (recall that at this point in the discussion, predicates are not being considered).

This NFA can be formally defined as a *Moore Machine* [Hopcroft and Ullman 1979]. The output function of the Moore Machine here is a mapping from the set of accepting states

Q1=/a/b
Q2=/a/c
Q3=/a/b/c
Q4=/a//b/c
Q5=/a/*/c
Q6=/a//c
Q7=/a/*/*/c
Q8=/a/b/c

(a) Path queries



(b) A corresponding NFA

Figure 4.2: An NFA-based Representation of Path queries

to a partitioning of identifiers of all queries in the system, where each partition contains the identifiers of all the queries that share the accepting state.

4.3.2 Constructing a Combined NFA

Having presented the basic NFA model used by YFilter, I now describe an incremental process for NFA construction and maintenance. The shared NFA shown in Figure 4.2 was the result of applying this process to the eight queries shown in that figure.

The four basic location steps in the subset of XQuery that this work supports are "/a", "//a", "/*" and "//*", where "a" is an arbitrary symbol from the alphabet consisting of all elements defined in a DTD, and "*" is the wildcard operator. Figure 4.3 shows the directed graphs, called *NFA fragments*, that correspond to these basic location steps.

Note that in the NFA fragments constructed for location steps with "//", there is an ε-transition moving to a state with a self-loop. This ε-transition is needed so that when combining NFA fragments representing "//" and "/" steps, the resulting NFA accurately maintains the different semantics of both steps (which will be explained shortly below). The

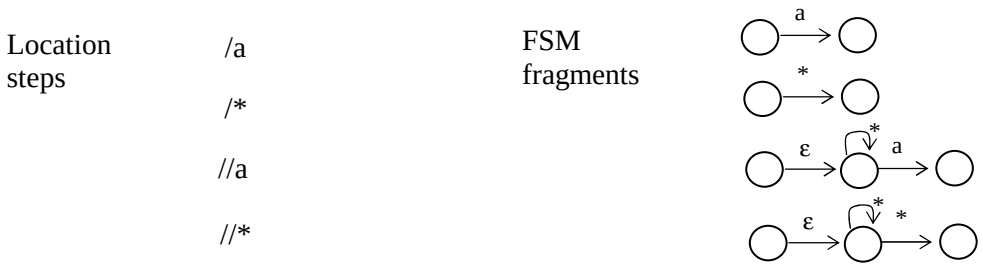


Figure 4.3: NFA Fragments of Basic Location Steps

NFA for a path expression, denoted as NFA_p, can be built by concatenating all the NFA fragments for its location steps. The final state of this NFA_p is the (only) accepting state for the expression.

NFA_ps are combined into a single NFA as follows: There is a single initial state shared by all NFA_ps. To insert a new NFA_p, the combined NFA is traversed until either: 1) the accepting state of the NFA_p is reached, or 2) a state is reached for which there is no transition that matches the corresponding transition of the NFA_p. In the first case, that final state is made an accepting state (if it is not already one) and the query ID is added to the query set associated with the accepting state. In the second case, a new branch is created from the last state reached in the combined NFA. This branch consists of the mismatched transition and the remainder of the NFA_p. Figure 4.4 provides four examples of this process.

Figure 4.4 (a) shows the process of merging a fragment for location step "/a" with a state in the combined NFA that represents a "/b" step. This process does not combine the edge marked by "a" and the edge marked by "b" into one marked by "a,b" as in a standard NFA, because the states after edge 'a' and edge 'b' differ in their outputs, so they cannot be combined. For the same reason, this process treats the "*" symbol in the way that it treats the other symbols in the alphabet, as shown in Figure 4.4 (b).

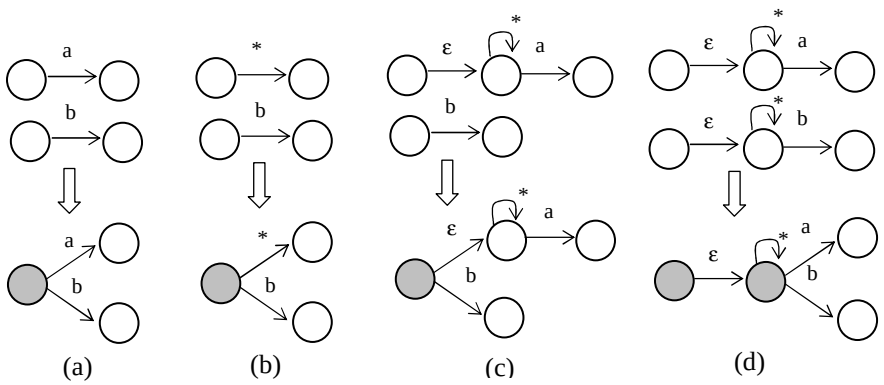


Figure 4.4: Merging NFA Fragments

Figure 4.4 (c) shows the process of merging a "//a" step with a "/b" step, while Figure 4.4 (d) shows the merging of a "//a" step with a "//b" step. Here it can be seen why the ε-transition is needed in the NFA fragment for "//a". Without it, when the fragment is combined with the NFA fragment for "/b", the latter would be semantically changed to "//b". The merging process for "//*" with other fragments (not shown) is analogous to that for "//a".

The "*" and "//" operators introduce *Non-determinism* into the model. "*" requires two edges, one marked by the input symbol and the other by "*", to be followed. The descendent operator "//" means the associated node test can be satisfied at any level at or below the current document level. In the corresponding NFA model, if a matching symbol is read at the state with a self-loop, the processing must both transition to the next state, and remain in the current state awaiting further input.

It is important to note that because NFA construction in YFilter is an *incremental* process, new queries can easily be added to an existing system. This ease of maintenance is a key benefit of the NFA-based approach.

4.3.3 Implementing the NFA Structure

The previous section described the logical construction of the NFA model. For efficient execution, the NFA is implemented using a hash table-based approach, which has been shown to have low time complexity for inserting/deleting states, inserting/deleting transitions, and actually performing the transitions [Watson 1997].

In this approach, a data structure is created for each state, containing: 1) The ID of the state, 2) type information (i.e., if it is an accepting state or a //-child as described below), 3) a small hash table that contains all the legal transitions from that state, and 4) for accepting states, an ID list of the corresponding queries.

The transition hash table for each state contains [*symbol, stateID*] pairs where the *symbol*, which is the key, indicates the label of the outgoing transition (i.e., element name, "*", or "ε") and the *stateID* identifies the child state that the transition leads to. Note that the child states of the "ε" transitions are treated specially. Recall that such states have a self-loop marked with "*" (see Figure 4.3). For such states (called "//-child" states), the self-loop is not indexed in the transition hash table. As described in the next section, this is possible because transitions marked with "ε" are treated specially by the execution mechanism.

4.3.4 Executing the NFA

Having walked through the logical construction and physical implementation, I now describe the execution of the machine. Following the XFilter approach, YFilter executes the NFA in an event-driven fashion; as an arriving document is parsed, the events raised by the parser callback the handlers and drive the transitions in the NFA. In addition, YFilter employs a

stack-based mechanism to deal with two issues that arise in event-based NFA execution over XML data:

- Backtracking in the structure: The nesting of XML elements requires that when an "end-of-element" event is raised, NFA execution must backtrack to the states it was in when the corresponding "start-of-element" was raised. The stack mechanism facilitates such backtracking.
- Non-determinism: Since many states can be active simultaneously in an NFA, the run-time stack mechanism is also used to track multiple active paths.

Details of the execution algorithm are described in the following handlers.

Start Document Handler: When an XML document arrives to be parsed, the execution of the NFA begins at the initial state. That is, the common initial state is pushed to the runtime stack as the active state.

Start Element Handler: When a new element name is read from the document, the NFA execution follows all matching transitions from all currently active states, as follows. For each active state, four checks are performed.

1) First, the incoming element name is looked up in the state's hash table. If it is present, the corresponding *stateID* is added to a set of "target states".
2) Second, the "*" symbol is looked up in the hash table. If it exists, its *stateID* is also added to the set of target states. Since the "*" symbol matches any element name, a transition marked by it is always performed.
3) Then, the type information of the state is checked. If the state itself is a "//-child" state, then its own *stateID* is added to the set, which effectively implements a self-loop marked by the "*" symbol in the NFA structure.

4) Finally, to perform an ε-transition, the hash table is checked for the "ε" symbol, and if one is present, the //-child state indicated by the corresponding *stateID* is processed recursively, according to the three rules above.[3]

After all the currently active states have been checked in this manner, the set of "target states" is pushed onto the top of the run-time stack. They then become the "active" states for the next event. If a state in the target set is an accepting state, the identifiers of all queries associated with the state are collected and added to an output data structure.[4]

End Element Handler: When an end-of-element is encountered, backtracking is performed by simply popping the top set of states off the stack.

Finally, it is important to note that, unlike a traditional NFA, whose goal is to find one accepting state for an input, the NFA execution here must find all matching queries. Thus, even after an accepting state has been reached for a document, the execution must continue until the document has been completely processed.

An example of this execution model is shown in Figure 4.5. On the left of the figure is the index created for the NFA of Figure 4.2. The number on the top-left of each hash table is a state ID and hash tables with a bold border represent accepting states. The right of the figure shows the evolution of the contents of the runtime stack as an example XML fragment is parsed. In the stack, each state is represented by its ID. An underlined ID indicates that the state is a //-child.

---

[3] Note that this process traverses at most one additional level, since //-child nodes cannot themselves contain an "ε" symbol.

[4] If predicate processing is not needed, the accepting state can be marked as "visited" to avoid processing matched queries more than once.

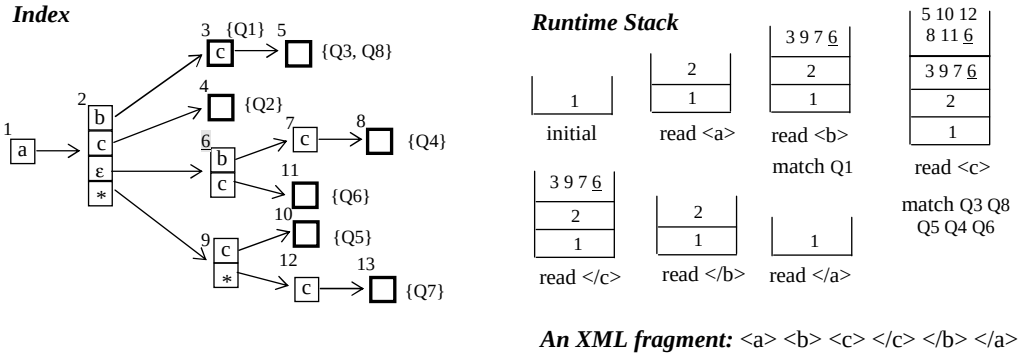


Figure 4.5: An Example of NFA Execution

4.3.5 Discussion

The discussion in the previous sections showed the key benefits of using an NFA-based implementation of the combined FSM: substantial reduction in machine size and incremental construction and maintenance of the machine. Of course, it is reasonable to be concerned that using an NFA could lead to performance problems due to (for example) the need to support multiple transitions from each state. A standard technique for avoiding such overhead is to convert the NFA into an equivalent DFA [Hopcroft and Ullman 1979]. A straightforward conversion could theoretically result in severe scalability problems due to an explosion in the number of states. But, as pointed out in [Green et al. 2003], this explosion can be avoided in many cases by placing restrictions on the types of documents and queries supported, and lazily constructing the DFA.

Results of a detailed performance study (presented in the next section), however, indicate that concerns about NFA performance in this environment are unwarranted. In fact, in the YFilter system, path evaluation (using the NFA) is sufficiently fast, that it is typically not the dominant cost of filtering. Rather, other costs such as document parsing are in many cases

more expensive than the basic path matching, particularly for systems with large numbers of similar queries. Thus, while it may in fact be possible to further improve path matching speed, the substantial benefits of flexibility and incremental maintenance provided by the NFA model outweigh any marginal performance improvements that remain to be gained by even faster path matching.

4.4 Performance of Structure Matching

In this section, I examine YFilter's path matching performance in the absence of predicate evaluation. Recall that the development of YFilter was motivated by the desire to share processing during path evaluation. As such, the focus of this performance study is on the impact of such shared processing.

4.4.1 Algorithms

This study compares the performance of XFilter, YFilter, and a hybrid approach that serves as a middle point between XFilter and YFilter with respect to the amount of sharing exploited. The hybrid approach is used to help quantify the performance impact of improved shared path matching.

XFilter [Altinel and Franklin 2000]. As described in Section 2.4, XFilter creates an FSM for each path query and builds a dynamic index over the states of FSMs to efficiently match the structure of path queries. Among a few indexing methods proposed in XFilter, the experiments presented in the following sections used the List Balance technique, as it was shown to provide better performance overall than the other indexing methods. To help understand the results of comparison of XFilter and YFilter reported below, details on the query indexes and execution algorithm that XFilter uses are provided in Appendix A.

A Hybrid Approach. The hybrid approach is an improved version of XFilter, which exploits some path sharing, but not as much as YFilter. In Hybrid, queries are decomposed into substrings containing only "/" operators (i.e., they are split at "*" and "//" operators). The processing of these substrings is shared, but the processing of the operators between these substrings is done individually for each query. A more detailed description of this approach is provided in Appendix B.

Independently of the research on YFilter, Chan et al. developed several algorithms for XML filtering under the name "XTrie" [Chan et al. 2002]. XTrie uses a "minimal decomposition" of queries that is identical to the decomposition that the Hybrid approach uses. Furthermore, Hybrid's execution model is similar in spirit to the "eager TRIE" version of XTrie in that matching of substrings is shared among queries and transitions between substrings are handled on an individual query basis. It is worth noting that "eager TRIE" is not the best performing approach studied by Chan et al. Other optimizations, orthogonal to the issue of sharing, have been developed in that work.

Despite the similarity between Hybrid and XTrie [Chan et al. 2002], I do not claim to do a direct comparison with that work. However, Chan et al. did compare their approaches to XFilter with List Balance, so as discussed in the following sections, it is possible to gain some insight into the relative performance of the YFilter techniques and the variants of XTrie.

A Simple Optimization. Also, for both XFilter and Hybrid, this study uses a simple optimization that is important in some of the workloads, namely, that *identical* queries are represented in the system only once. This is done by pre-processing the queries and collecting the IDs of identical queries in an auxiliary data structure. This structure is the

same as that used by YFilter to manage query IDs in accepting states. YFilter, of course, does not require such an optimization as it naturally shares processing of identical queries.

4.4.2 Experimental Set-up

The three algorithms (YFilter, XFilter with List Balance, and Hybrid) were implemented in Java. All of the experiments reported here were performed on a Pentium III 850 Mhz processor with 384MB memory running JVM 1.3.0 in server mode on Linux 2.4. The JVM maximum allocation pool was set to 250MB, so that virtual memory and other I/O-activity had no influence on the results. This was also verified using the Linux *vmstat*() command.

Workload Generation. While, as stated previously, the three path matching algorithms do not require DTD information, DTDs were used to generate the workloads for the experiments. This section focuses on workloads generated using the NITF (News Industry Text Format) DTD [IPTC, 2004], which has been used in previous studies [Altinel and Franklin 2000; Chan et al. 2002]. Experiments were also run using two other DTDs: The Xmark-Auction DTD [Busse et al., 2001] from the Xmark benchmark, and the DBLP [Ley 2001] bibliography DTD. Some characteristics of these DTDs are shown in Table 1. Note that all of the DTDs allow an infinite level of nesting due to loops involving multiple elements.

	NITF	Auction	DBLP
number of elements names	123	77	36
number of attributes in total	510	16	14
maximum level of nesting allowed	Infinite	infinite	infinite

Table 1: Characteristics of three DTDs

Given a DTD, the tools used to run an experiment include a *DTD parser*, a *query generator*, an *XML generator*, and an *event-based XML parser* supporting the SAX interface.

The DTD parser which was developed using a WUTKA DTD parser [Wutka 2000] outputs parent-child relationships between elements, and statistical information for each element including the probability of an attribute occurring in an element (randomly chosen between 0 and 1) and the maximum number of values an element or an attribute can take (randomly chosen between 1 and 20). The output of the DTD parser is used by the query generator and the document generator.

A query generator was developed to create a set of path queries based on the workload parameters listed in Table 2. The query generator generates random query strings according to the input DTD and these parameters. In order to remove some semantic redundancy that may be introduced by this random approach, it performs a simple rewriting step in which the following rules are applied in the presented order: 1) For each occurrence of "//*" in a query, turn it into "/*//"; 2) If a query contains multiple consecutive "/*//" substrings, only keep the first one; and 3) If "/*//" occurs at the end of a query, remove "//".

Parameter	Range	Description
Q	1000 to 500000	Number of queries
D	6 to 10	Maximum depth of XML documents and XPath queries.
W	0 to 1	Probability of a wildcard "*" occurring at a location step
DS	0 to 1	Probability of "//" being the operator at a location step
Distinct	True or False	Query strings required to be unique?
P	0 to 20	Number of predicates per query
NP	0 to 3	Number of nested paths per query
RP	2, 3, 5	Max. no. of repeats of an element under a single parent

Table 2: Workload parameters for query and document generation

The query generator can be set to create workloads with or without duplicate queries. This later mode is referred to as the *distinct* mode. If duplicates are allowed, the generator is simply invoked Q times. Otherwise, in the distinct mode the query generator is invoked

repeatedly until Q syntactically unique queries are produced. Of course, in such a *distinct* workload there may be significant overlap in the query strings but no two strings will be identical. Note that in most of the experiments reported in this study, the query generator is used in the distinct mode.

For document generation, IBM's XML Generator [Diaz and Lovell, 1999] was used to create the structure of documents. Two parameters were passed to the generator: maximum depth D, and RP, which specifies the maximum number of times that an element can be repeated under a single parent. As a default, RP is limited to 3. Then attributes of elements were generated according to their probabilities of occurring. The value of an element or an occurring attribute was randomly chosen between 1 and the maximum number of values it can take.

For each DTD, a set of 200 XML documents were created. All reported experimental results were averaged over this set. For each experiment, a set of queries was generated according to the workload setting. For each algorithm run in an experiment, queries were preprocessed, if necessary, and then bulk loaded to build the index and other data structures. Then XML documents were read from disk one after another. The execution for a document returned a bit set, each bit of which indicates whether or not the corresponding query has been satisfied. A new process was used for each experiment run of an algorithm (i.e., 200 documents), to avoid complications from Java's garbage collector.

Metrics. Previous work [Altinel and Franklin 2000; Chan et al. 2002] used "filtering time" as the primary performance metric, which is the total time to process a document including parsing and outputting results. Noting that Java parsers have varying parsing costs, this study instead reports on a slightly different performance metric called "multi-query

processing time (MQPT)". MQPT captures all costs attributable to the filtering algorithms themselves. It is simply the filtering time minus the document parsing time. That is:

Multi-query processing time (***MQPT***) = Filtering time – Document parsing time

Filtering time = Wall clock time from the start of document parsing to the end of output

MQPT for path matching consists of two components: *path navigation* and *result collection*. The former captures the cost of state transitions driven by received events. The latter is the cost to collect the identifiers of queries from the auxiliary data structures and to mark them in the result bit set. Note that when only distinct queries are used in experiments, the cost of result collection is negligible.

Where appropriate, other metrics such as the number of transitions followed, the size of the various machines, and the costs associated with maintenance, are also reported.

4.4.3 Efficiency and Scalability

Having described the experimental environment, I begin my discussion of experimental results by presenting the MQPT results for the three alternatives as the number of queries in the system is increased.

Experiment 1: NITF. In this experiment 200 XML documents were generated using the NITF DTD under the workload (D = 6, RP = 3). The average length of generated documents is 77 in terms of start-end element pairs. The average level of nesting of elements is 5.45.

The MQPT for the three algorithms is first examined as the number of *distinct* queries in the system is increased from 1000 to 150,000 with the probability of "*" and "//" operators each set to 0.2. With this setting, each query contains approximately one "*" operator and one "//" operator. Recall that experiments presented in this section focus on structure

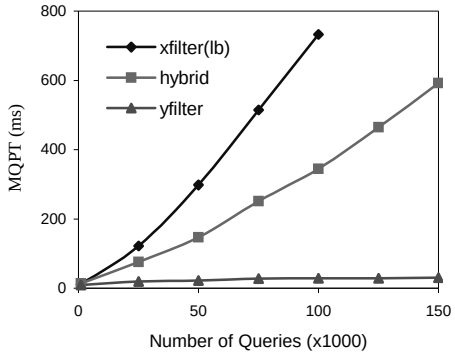


Figure 4.6: Varying number of distinct queries
(NITF, D=6, W=0.2, DS=0.2)

matching only, so there are no predicates on the elements. Predicate processing is studied in Section 5.

The results are shown in Figure 4.6. As can be seen in the figure, YFilter provides the significantly better performance than the other two across the entire range of query populations. XFilter is the slowest here by far, and not surprisingly, Hybrid's performance lies between the two.

As the number of queries increases, YFilter exhibits a slight cost increase and levels off around 30ms when Q is larger than 50,000. In contrast, the processing cost of XFilter increases dramatically, to 732ms at 100,000 and runs out of memory after this point, while Hybrid takes 344ms at this point. Thus YFilter exhibits an order of magnitude improvement for path matching over these other schemes.[5]

The performance benefits of YFilter come from two factors. The first is the benefit of shared work obtained by the NFA approach. YFilter is the most effective of the three at

[5] Note that the performance of XTrie was also compared with that of XFilter [Chan et al., 2002] for a similar workload. The fastest algorithm studied there, called Lazy Trie, was shown to have only about a 4x improvement over XFilter.

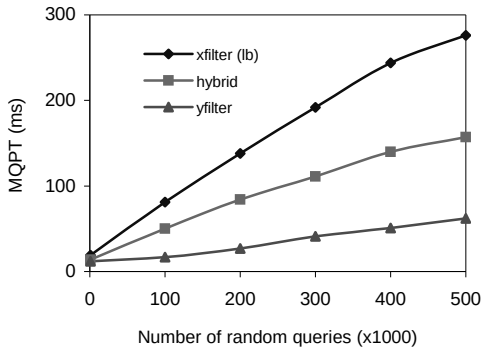


Figure 4.7: Varying number of queries (with duplicates)
(NITF, D=6, W=0.2, DS=0.2)

exploiting commonality among similar, but not exactly identical queries, as it can share all common prefixes of the paths. The second factor is the relatively low cost of state transitions in YFilter compared to the others, which results from the hash-based implementation described in Section 4.3.3. This was verified by comparing the improvement ratio of YFilter over XFilter in terms of path navigation time with that in terms of the number of transitions. For example, when Q is 100,000, XFilter makes 7.4 times more transitions but takes 25.2 times longer to navigate.

The experiment just described, like other XML filtering studies [Altinel and Franklin 2000; Chan et al. 2002; Green et al. 2003] did not address the effect of duplicate path queries on the query processing time. Duplicate paths, however, are likely to exist in a large filtering system. For this reason, the previous experiment was re-run with the query generator set to not remove duplicates. Figure 4.7 shows the MQPT of three algorithms as the number queries in the system is varied from 1,000 to 500,000.

Compared to Figure 4.6, YFilter still achieves a significant performance improvement over Hybrid and XFilter, but the differences among the algorithms are not as great. In particular, XFilter and Hybird seem to scale better, and the cost of YFilter increases.

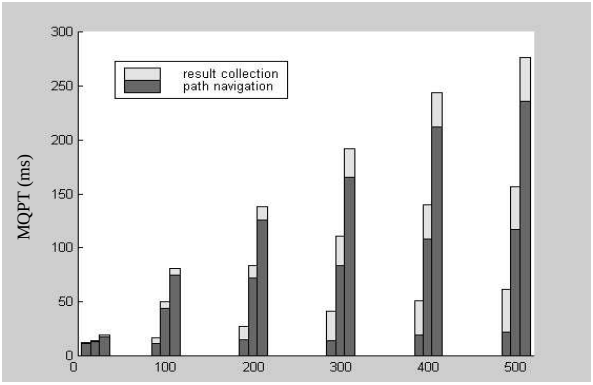


Figure 4.8: Component costs for processing queries containing duplicates
(NITF, D=6, W=0.2, DS=0.2)

Table 3 reports the number of distinct queries among random queries measured in this experiment. It shows the relatively slow increase in the number of distinct queries. Since all three algorithms represent identical queries only once, they all benefit from the slow increase, which explains the improved MQPT of Hybrid and XFilter.

Number of random queries (x1000)	1	100	200	300	400	500
Number of distinct queries (x1000)	0.53	15.7	24.2	30.5	35.6	40.0

Table 3: Number of distinct queries out of randomly generated queries
(NITF, D=6, W=0.2, DS=0.2)

The MQPT is further decomposed into two component costs: path navigation and result collection. Results are shown in Figure 4.8. For each data point, the bars represent from left to right: YFilter, Hybrid, and XFilter. The cost of path navigation at each data point is consistent with that for the same number of distinct queries in Figure 4.6. The cost of result collection, however, becomes significant. Even though result collection was coded carefully, e.g. using unsynchronized data structures and avoiding ID instance copies, its cost is still high in this experiment, because a high percentage of path expressions match each document (34% here for each value of Q compared to less than 10% for most values of Q in the previous experiment using distinct queries). Note that in Figure 4.8, the MQPT of YFilter is

dominated by the cost of result collection starting from the point of Q=300,000. At this point, the number of query IDs collected is 9.3 times the number of state transitions YFilter makes.

The above results for duplicate path queries indicate that experiments using distinct paths may tend to magnify the differences among filtering algorithms in scenarios where duplicate queries are likely. To exhibit a significant performance improvement in practical workloads containing duplicate queries, a filtering algorithm needs to outperform others by a wide margin, as YFilter outperforms Hybrid and XFilter.

Similarly, for both the distinct and random workloads, document parsing is another fixed overhead that contributes to overall filtering time (recall that parsing is not included in MQPT). For example, the Xerces [Apache XML, 1999] parser used in this experiment, set in a non-validating mode, took 168ms on the average to parse a document, completely dominating the NFA-based execution in both cases. Other publicly available java parsers that were also tried include Java XML Pack [Sun Microsystems, 2001] and Saxon XSLT processor [Kay, 2001] supporting SAX 2.0. Saxon gave the best performance at 81 ms, still substantially more than the NFA navigation cost.[6] Thus, while YFilter is not claimed to be the fastest possible path matching approach, it is clear that its performance for both these workloads is sufficiently fast that any further improvements in path navigation time will have at best, a minor impact on overall performance.

[6] I have also experimented with C++ parsers, which are much faster, but even with these parsers parsing time would be expected to be at best similar to the cost of path navigation with YFilter, particularly if YFilter were also implemented in C++!

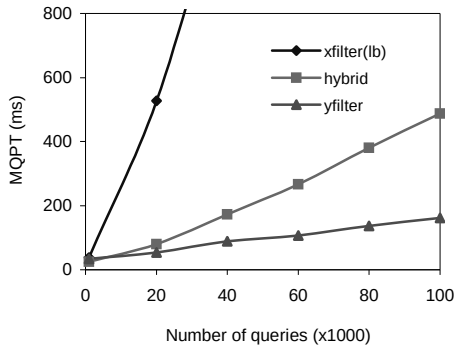


Figure 4.9: Varying number of distinct queries
(Auction, D=8, W=0.2, DS=0.2)

Experiment 2: Other DTDs. Experiments were also run using two other DTDs: DBLP and Xmark-Auction. For these two DTDs, the maximum depth D was set to 8 in order to generate a relatively large set of distinct queries. The setting of W and DS is the same as the previous experiments. Only the results for the *distinct* query workload are reported below. Due to their DTD structures, DBLP tends to generate very short documents, while Xmark-auction tends to produce very long ones. The RP parameter was adjusted to control the document lengths for these experiments. For DBLP, RP was set to 5 and the generated documents contain on average, 16 start and end elements pairs. For Auction, RP was set to 2, obtaining an average document length of 175. The results of these experiments are similar to those obtained using the NITF workload. They are summarized below.

Figure 4.9 shows the MQPT results for the Xmark-Auction workload as Q is varied from 1,000 to 100,000. It can be seen that the trends observed using NITF also hold here: YFilter performs substantially better than XFilter and Hybrid is in between the others. Since documents here are 2.3 times as long as those of NITF, all algorithms take longer to filter the documents. XFilter, however, is particularly sensitive to the length of documents because its

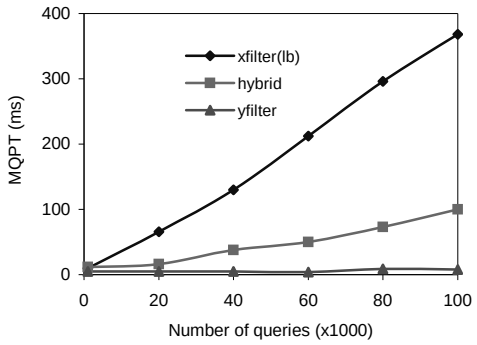


Figure 4.10: Varying number of distinct queries
(DBLP, D=8, W=0.2, DS=0.2)

FSM representation and execution algorithm result in significant memory management overhead, which in turn invokes garbage collection much more frequently.

When the DBLP DTD is used, all algorithms run much faster, as shown in Figure 4.10. However, even though the documents used here are very short, YFilter still achieves substantial performance improvement over XFilter (e.g., 46 times at Q=100,000).

4.4.4 Experiment 3: Varying the maximum depth

This experiment examines the impact of document depth on the performance of the three algorithms. Of particular concern is the performance of YFilter, since deep documents could theoretically cause an exponential blow-up in the number of active states for NFA execution. The NITF DTD was used in all the following experiments. The maximum depth was increased from 6 to 10.[7] For each D value, 50,000 distinct queries were generated.

[7]Note that the value of D was stopped at 10, because in large-scale XML filtering scenarios, documents even that deep are quite rare. In other scenarios such as general XML query processing in large databases, some researchers expect that documents may be more deeply nested. Such scenarios are beyond the scope of this thesis; the interested reader is referred to [Bruno et al., 2003] for a discussion of the performance of NFA-based solutions in such settings.

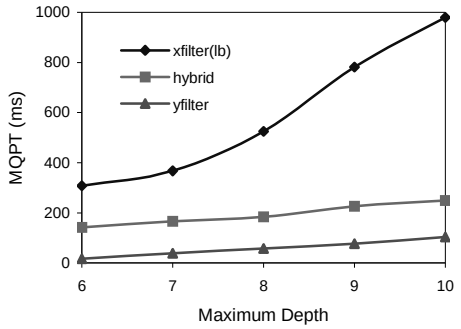


Figure 4.11: Varying maximum depth
(NITF, Q=50,000, W=0.2, DS=0.2)

As can be seen in Figure 4.11, the MQPT for all algorithms increases with the document depth, but YFilter remains the fastest. More importantly, the increase for YFilter is linear. To provide a better understanding these behaviours, the statistics on documents and queries used in this experiment are reported in Table 4. Note that the average document depth (i.e., the average depth of all paths in each document) and query depths do not increase as quickly as D. This is because the DTD dictates that many paths cannot reach a very deep level. As the maximum repeat RP was fixed to 3 in this experiment, a larger value of D also caused longer documents (i.e., more start/end element pairs) to be generated.

Maximum Depth D	6	7	8	9	10
Avg. Document depth	5.45	6.06	6.68	7.28	7.69
Avg. Query depth	5.05	5.70	6.09	6.35	6.53
Avg. Document length	77	107	154	221	271

Table 4: Characteristics of documents and queries as maximum depth varies

Given these statistics, the increase in MQPT of the filtering algorithms can be explained by two factors: the increased document length and the increased document depth. In the case of YFilter, the number of state transitions made increases 5.9 times as D is increased from 6 to 10. Much of the increase comes from the simple fact that there are 3.5 times more start/end element pairs in the documents when D = 10 compared to when D = 6. Although

the increased document depth could theoretically cause exponential increase in the number of transitions, it was not observed in this experiment, because in the NFA execution, most input elements can trigger transitions only from a limited subset of the active states.

Note that the NITF DTD used here is one of the few complicated DTDs published online in terms of the number of elements allowed to be recursive (26 out of 123 elements). For this reason, YFilter's performance shown in this experiment serves as a good indicator of its sensitivity to the maximum level of element nesting in most other practical workloads.

4.4.5 Experiment 4: Varying Non-determinism

In the previous experiments, the W and DS parameters (the probability of "*" and "//" operators, respectively) were fixed at 0.20. Wildcards and "//" operators, however, are the sources of non-determinism in the NFA-based model. Thus, this set of experiments investigates their impact on filtering performance. In order to separate the effects of these two parameters, experiments here fixed one at 0 while varying the other. Note also that a large D value (10) was used in order to allow a reasonable number of distinct queries to be generated for all measured values of W and DS.

Varying W and DS can dramatically impact the properties of the query sets produced by the query generator. Thus, the query generation technique was modified for these experiments. A large set of distinct queries was first generated using the setting (D=10, W=0, DS=0). Then to experiment with different W values, for each query in this set, elements were replaced with wildcards with probability = W; if due to this process, a query became identical to an existing one in the query set, the duplicate query was not added to the set. Query sets for the cases with varying DS were generated similarly.

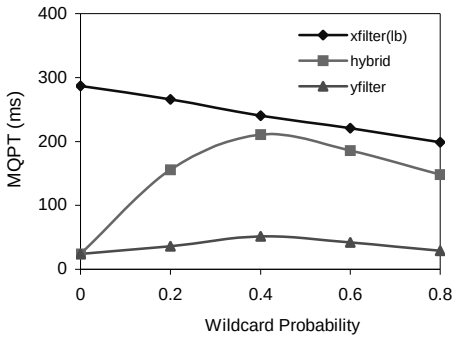


Figure 4.12: Varying wildcard probability (NITF, Q=50,000, D=10, DS=0)

Varying W. Figure 4.12 shows the MQPT results when W is varied from 0 to 0.8 with Q = 50,000.[8] As can be seen in the figure, YFilter again significantly outperforms the others. Note also that it is much less sensitive to this parameter than the other two algorithms. The reason for YFilter's low sensitivity to W is explained as follows. As W increases, the size of the NFA changes slowly, due to the prefix sharing among path expressions. As W is increased from 0, the NFA grows somewhat because the addition of wildcards adds new paths to the NFA. As W is further increased, the NFA size actually begins to decrease, as the queries become more similar to each other. In this experiment, the NFA begins with approximately 82,000 states (when W = 0), and increases to a high of approximately 112,000 when W = 0.4.

In contrast, XFilter's performance improves with increasing W. Since XFilter does not store nodes for wildcards, the number of transitions it makes is reduced as wildcards are added.

In this experiment, the performance of Hybrid demonstrates that it does in fact share common attributes with both XFilter and YFilter. When W and DS are both set to 0, Hybrid

[8] Note that at W=1 very few distinct queries can be generated, so that case is not shown here.

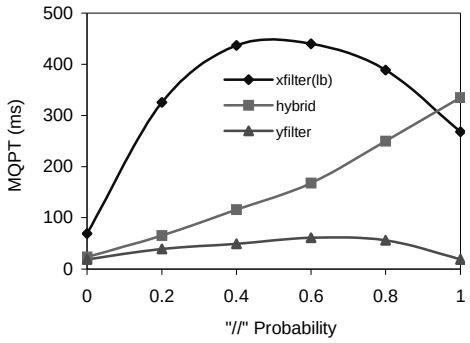


Figure 4.13: Varying "//" probability (NITF, Q=10,000, D=10, W=0)

is similar to YFilter as there is no decomposition of queries. As W (or DS) increases, Hybrid moves more towards XFilter due to increased query decomposition. Beyond a certain point (W = 0.4, here), the benefit of not processing wildcards becomes dominant, and Hybrid's performance improves along with XFilter's.

Varying DS. Figure 4.13 shows the effect of varying DS (the probability of "//" operators) from 0 to 1 with Q = 10,000 (a smaller number of queries was used here because XFilter was unable to complete for the mid-range values of DS with more queries). As in the previous experiment, YFilter has the best performance overall and is less sensitive to the parameter setting than the other two.

The performance of YFilter is again largely explained by the change in the machine size. As DS is increased from 0 to 1, the machine size first increases because of the diversity of axes in location steps in queries, and then decreases, as queries become more similar to each other. The turning point here occurs at DS = 0.6, where the machine size is 2.8 times that at DS = 0, resulting in a 3.2 times increase in MQPT. The performance degradation is kept small due to the shared processing of "//" operators among multiple queries.

In contrast, XFilter does pay a large performance penalty as DS is increased. This penalty is due to the overhead it incurs when processing "//" operators in the presence of recursive elements. Recall that (as described in Section 4.4.1) in XFilter, if a location step "//a" can be matched by recursive "a" elements, the path node of the subsequent location step will be promoted to its candidate list each time "//a" is matched. In XFilter's implementation, if the subsequent location step contains a "//" operator (e.g. "//b"), its path node is simply added to the candidate list multiple times. However, if the next location step contains a '/' operator instead (e.g. "/b"), different instances of this path node are first created and then added to the candidate list to remember all the possible levels where this location step could be matched. Note that the probability of patterns such as "//a/b" first increases with DS and then decreases. The behavior of XFilter in this experiment is determined by multiple promotions of path nodes in general and the overhead of handling these particular patterns.

In this experiment, Hybrid again exhibits characteristics of the other two algorithms. When DS = 0, Hybrid is similar to YFilter, and as DS is increased, it becomes more like XFilter. Hybrid, however, does not exhibit the bell shape, because it uses a single runtime stack to keep track of the active states as in YFilter, rather than promoting path nodes multiple times to remember different document levels as in XFilter. At DS = 1, every query is decomposed into single elements and the performance of Hybrid is very close to XFilter. XFilter actually outperforms Hybrid a little as a benefit of using List Balance.

The experiments on non-determinism have shown that compared to the other two algorithms, YFilter shows relatively little sensitivity to the W and DS parameters. Due to prefix sharing, increasing the probabilities has only a modest effect on the size of the NFA.

As a result, the filtering cost of YFilter is relatively low and robust to changes in these parameters.

4.4.6 Experiment 5: Maintenance cost

The last set of experiments reported in this section deals with the efficiency of maintaining the YFilter structure, which is expected to be one of the primary benefits of the approach. Updates to the NFA in YFilter are handled as follows: To insert a query, the NFA representation of the query is merged with the combined NFA as described in Section 4.3.2, and the identifier of the query is appended to the end of the query ID list at its accepting state. To delete a query, the accepting state of the query is located and the query's identifier is deleted from the list of queries at this state. If the list becomes empty and the state does not have a child state in the NFA, the state is deleted by removing the link from its parent. The deletion of this state can be propagated to its predecessors. An update to a query is treated as a delete of the old query followed by the insertion of the new one.

Deletion is the dual problem of insertion except that modification of the list at the accepting state can be more expensive than appending an identifier to the list. As demonstrated in the previous sections, YFilter's performance is fairly robust with respect to the number of queries in the system. Thus, instead of deleting queries immediately, YFilter adopts a lazy approach where a list of deleted queries is maintained. This list is used to filter out such queries before results are returned. The actual deletions can then be done asynchronously. Thus, this section focuses on the performance of inserting new queries.

The cost of inserting 1000 queries was measured with varying numbers of queries already in the index (which can contain duplicate queries). The insert costs are shown in Table 5. With Q = 2000 (i.e., 2000 queries already in the NFA), it takes 77 ms to insert the

1000 new queries. At this point, the chance of a query being new is high, requiring new states to be created and transition functions to be expanded by adding more hash entries to the states. However, the cost drops dramatically as more queries are present in the system. Beyond Q=50,000, the insertion cost stabilizes around 5 ms. This is because most paths are already present in the index, so a new query can typically be added by simply traversing down a path to an existing accepting state and appending the query ID to the list at that state.

Q (x1000)	2	4	6	8	10	10 ~ 50	60 ~ 500
1000 Insertions (ms)	77	57	30	24	9	6	≈ 5

Table 5: Cost of inserting 1000 queries (ms) (NITF, D=6, W=0.2, DS=0.2)

4.5 Related Work

Much of the work related to structure-based XML filtering was discussed in Chapter 3. This section describes XML filtering systems in more detail and other NFA-based techniques.

As mentioned in Section 3.1, a number of XML filtering systems have been developed to efficiently match a large set of path queries with streaming documents. *XFilter* [Altinel and Franklin, 2000] was described in Section 2.4 and Appendix A. *CQMC* [Ozen et al., 2001] improved upon XFilter by building an FSM for a set of queries identical in structure. *XTrie* [Chan et al. 2002] supports shared processing of query fragments containing only child ('/') axes. *Index-Filter* [Bruno et al., 2003] builds indexes over both queries and streaming data; the index over data speeds up the processing of large documents but its construction overhead penalizes the processing of small ones. *XMLTK* [Green et al., 2004] converts YFilter's NFA to a *Deterministic Finite Automaton* (DFA) to further improve the filtering speed while limiting the complexity of data and frequency of query updates that message brokers support. Compared to the above approaches, YFilter combines fast path matching,

flexibility, and ease of maintenance, thus providing an efficient and robust solution to XML filtering.

DataGuides [Goldman and Widom, 1997; Nestorov et al., 1997] are structural summaries of an XML source database that can be used to browse database structure, formulate queries, and enable query optimization. Creating a DataGuide from a source database has been shown to be equivalent to converting an NFA to a DFA. The NFA-based approach in YFilter differs in that it is intended to represent path expressions rather than data and it must faithfully encode all of the expressions in their entirety, rather than just summarizing them. As a result, the implementations of YFilter's NFA and DataGuides differ significantly.

4.6 Summary

This chapter presented an efficient approach to structure-based filtering of XML documents. This approach merges all path expressions into a single combined NFA to exploit overlap and employs a stack-based mechanism to efficiently execute the NFA over incoming documents. Results of a detailed performance study show that YFilter provides an order-of-magnitude performance benefit over previous solutions. Using YFilter, path matching is no longer the dominant cost for XML filtering. YFilter is also highly scalable, supporting up to 100's of thousands of distinct queries with a single processor. Furthermore, it requires only a small maintenance cost for query updates, thus providing a robust solution to XML filtering in dynamic environments.

The next chapter discusses how to extend YFilter's shared structure matching to support advanced query predicates, and compares alternative techniques for the efficient integration of structure-based and predicate-based filtering.

5 Advanced Query Support for Filtering

The previous chapter demonstrated the substantial performance improvements that can be gained by sharing structure matching through the use of an NFA. Structure matching as such is one part of the XML filtering problem; another part is the evaluation of predicates that are be applied to path expressions for additional filtering. In this chapter, I address the second challenge of XML filtering: predicate processing in shared structure matching.

Shared structure matching complicates the handling of value-based predicates, which address attributes and text data of elements. I describe two alternative techniques for extending the NFA-based structure matching with support for such predicates. The results of a comparative study of these two techniques demonstrate some key differences between shared XML filtering and traditional database query processing. I also describe how YFilter extends the shared path matching approach to handle complex predicates that involve nested path expressions.

5.1 Value-Based Predicate Evaluation

For value-based predicates (e.g., //section[@difficulty = “easy”]/title), one could extend the NFA by including predicates as labels on additional transitions between states. Unfortunately, such an approach would result in a potentially huge increase in the number of states in the NFA, and would also destroy the sharing of path expressions, the primary advantage of using an NFA.

For this reason, YFilter explores two alternative approaches to implement value-based selections. Similar to traditional relational query processing, the placement of predicate

evaluation in relation to the other aspects of query processing can have a major impact on performance. Relational systems use the heuristic of "pushing" cheap selections as far as possible down the query plan so that they are processed early in the execution. Following this intuition, the first approach, called "*Inline*", processes value-based predicates as soon as the relevant state is reached during structure matching. The second, called *Selection Postponed* (*SP*), waits until an accepting state is reached during structure matching, and only at that point, applies all the value-based predicates for those queries whose structure has been matched. Below, I discuss these two alternatives in more detail, and compare their performance experimentally.

Note that in the following description I focus on the processing of predicates on attributes but not text data. Predicates on element data require additional bookkeeping because the data (if present) is delivered by the parser in separate "characters" events that may arrive at any time between the "start element" event and its corresponding "end element" event. Support for such predicates is discussed at the end of this section.

5.1.1 The Inline Approach

In the Inline approach, the information stored in each state of the NFA is extended to include any predicates that are associated with that state. These predicates are stored in a table, as shown in Figure 5.1. Since multiple path expressions may share a state, this table can include predicates from different queries, so (*Query Id*, *Predicate Id*) pairs are used to identify the predicates in the table.

Inline works as follows: When a start-of-element event is received, the NFA transitions as described in Section 4.3.4. For each state reached, the predicates stored there are checked. For each query, bookkeeping information is maintained, indicating which predicates of that

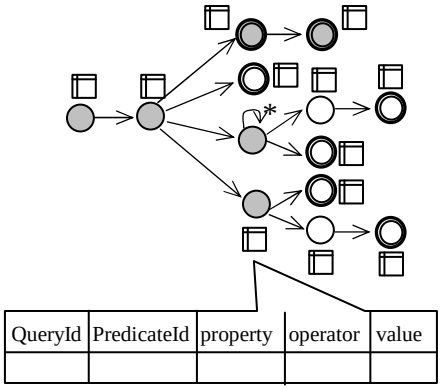


Figure 5.1: Predicate Storage for Inline

query have been satisfied. When an accepting state is reached, the bookkeeping information for the queries of that state is checked, and those queries for which all predicates have been satisfied are returned as matches.

While such an approach sounds conceptually simple, there are several issues to consider. The first is the potential benefit of checking predicates early. The failure of a predicate at a state does not necessarily stop processing along that path because there may be other queries sharing the state that did not fail. Furthermore, if a query contains a "//" prior to a predicate, then even if the predicate fails, the query effectively remains active due to the non-determinism introduced by that axis. For these reasons, the common query optimization heuristic of "pushing selects" to earlier in the evaluation process is not as likely to be effective in this environment.

A second issue is that, due to the nested structure of XML documents, it is possible that backtracking will occur during the NFA processing. Such backtracking further complicates the task of tracking the predicates that have been satisfied. For example, consider Query 5= "//a[@a_1=v_1][@a_2=v_2]" that contains a location step with two predicates (on two different attributes a_1 and a_2 of "a" elements). If care is not taken during backtracking, a fragment

such as "<a a_1=v_1> <a a_2=v_2> " could erroneously be determined to match Query 5 even though the attributes are associated with different "a" elements. This problem can be solved by "undoing" changes made to the predicate bookkeeping information for a state when backtracking from that state.

Unfortunately, the above solution does not solve a similar problem that exists for *recursively nested* elements. Consider Query 5 when applied to a fragment with nested "a" elements: "<a a_1=v_1> <a a_2=v_2> ". In order to distinguish between the two "a"s additional bookkeeping information must be kept. This additional information identifies the particular element that caused each predicate to be set to true. During the final evaluation for a query at its accepting state, the query is considered to be satisfied only if all predicates attached to the same location step are satisfied by the same element. The Inline approach is described in more detail in Appendix C.

5.1.2 Selection Postponed (SP)

Effort spent evaluating predicates with Inline will be wasted if ultimately, the structure-based aspects of a query are not satisfied. The Selection Postponed (SP) approach avoids this problem by delaying predicate processing until after the structure matching has been completed. SP has several other potential advantages. First, since the predicates on different elements in a query are treated as conjunctions, a short-cut evaluation method is possible; when a predicate of a query fails, the evaluation of the remaining predicates of that query can be avoided.[9] Second, there is no need to extend the NFA backtracking logic as for Inline.

---

[9] Note however, that with predicate evaluation it becomes possible to visit a given accepting state multiple times, due to predicate failure. Such short-cut predicate evaluation only saves work for a single visit.

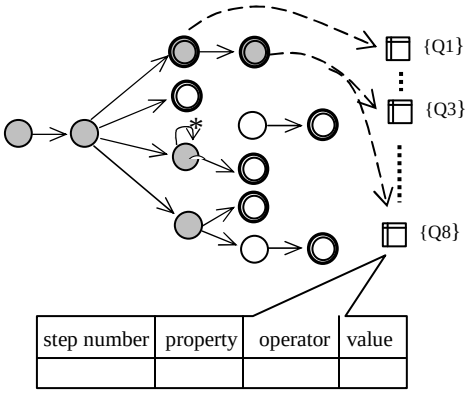


Figure 5.2: Predicate Storage for SP

In SP, the predicates are stored with each query, as shown in Figure 5.2. The predicates of a query are indexed by the “step number” field. When an accepting state is reached in the NFA, selections are performed in bulk for each query associated with the state. If all predicates of a query evaluate to true, then the query is satisfied.

In order to delay selection, however, the NFA must be extended to retain some additional history about the states visited during structure matching. The reason for this is demonstrated by the following example. Consider Query 6 and an XML document fragment as shown in Figure 5.3. When element ‘b’ of the document is parsed, the NFA execution arrives at the accepting state of this query in the NFA (also shown in Figure 5.3). When selection processing is performed for the predicate in Query 6, the processing needs to decide on which of the two ‘a’ elements encountered during parsing to apply the predicate.

A naïve method would be to simply check all of the ‘a’ elements encountered. Unfortunately with more “//” operators in a query or more recursive elements in the document, searching for matching elements for predicate evaluation could become as expensive as running the NFA again for this query. Instead, the SP approach extends the

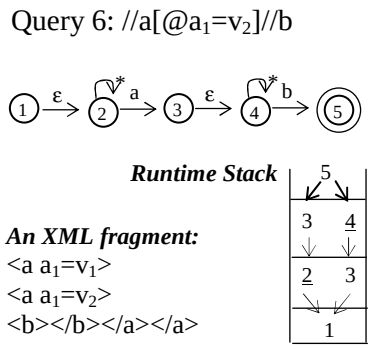


Figure 5.3: A sample query, its NFA, and the NFA execution

NFA execution to output not only *query ID*s, but a list of path matches. Each path match provides a list of document elements that should participate in predicate evaluation.

For example, at the accepting state for Query 6, the NFA execution would report the two path matches “a_1 b” and “a_2 b”, where a_1 represents the first ‘a’ element and a_2 represents the second (nested) ‘a’ element. Since predicates are indexed by “step number”, it is easy for the selection operator to determine which elements need to be tested. For the XML fragment shown in Figure 5.3, the first path match does not satisfy Query 6 because a_1 does not satisfy the predicate, but the second path match does.

The NFA execution is extended to output these path matches by linking the states in the runtime stack backwards towards the root. That is, for each target state reached from an active state, the NFA execution adds a predecessor pointer for the target state and sets the pointer to the active state. Then the target state with the pointer is later pushed onto the runtime stack. An example is shown in Figure 5.3, which includes the content of the stack for the accepting state of the sample query after the XML fragment was read.

For each state that is an accepting state, the NFA execution can traverse backwards to find the sequence of state visits that lead to the accepting state. Note that elements that trigger transitions to “//-child” states (along self-loops) can be ignored in this process, as they

do not participate in predicate evaluation. Returning to the example in Figure 5.3, there are two sequences of state visits, namely "2 3 5" and "3 4 5" that the NFA took when elements a_1, a_2 and b were read. After eliminating the elements that trigger transitions to "//-child" states for each state sequence, the two sequences of matching elements, "a_2 b" and "a_1 b", can be generated for predicate evaluation.

Note that the technique of linking states in the runtime stack using predecessor pointers in SP is similar in spirit to "backward chaining" used in *PathStack* and *TwigStack* [Bruno et. al. 2002]. The idea in both is to use backward pointers to store partial or complete matches of path expressions. The difference is that here SP uses a single runtime stack with backward pointers to store matches for ***all*** path expressions, while *PathStack* requires a stack for each query node.

The evaluation data structures and pseudo-code for predicate evaluation using SP are presented in Appendix D. Note that SP requires no bookkeeping information and that the evaluation code is simple and straightforward.

Finally, as mentioned above, predicates on element data cannot be evaluated with other value-based predicates in a query, because the element data is not returned when the "start element" event is encountered. The fact that selection in SP is decoupled from the event-based processing makes it possible to treat selections involving such predicates simply as blocking operators. To collect information for such selection operators, the elements carried by the path matches are extended to include a data field called "text". When a "characters" event is received, the data returned by this event is appended to the "text" field in the corresponding element. This field is known to be complete when the corresponding "end

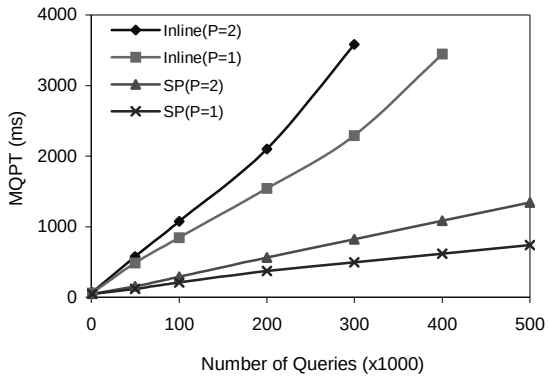


Figure 5.4: Varying number of queries (D=6, W=0.2, DS=0.2)

element" event is encountered. At that moment, selection operators blocked on this field will be signaled to become unblocked.

5.1.3 Performance of Value-based Predicate Evaluation

Having described the Inline and SP approaches to value-based selection, I now present results from an experimental study comparing their performance. The NITF DTD was used for all experiments presented in this section. For query generation, the parameter P (see Table 2) was used to determine the number of predicates that appear in each query. Such predicates are distributed over the location steps uniformly at random. Distinct queries are used in all of the experiments.

The first experiment examines the relative performance of Inline and SP as Q is varied from 1,000 to 500,000. Figure 5.4 shows the MQPT of the two approaches for the cases P=1 and P=2.

As can be seen in the figure, SP outperforms Inline by a wide margin. When P=1, for example, SP takes 375 ms to process 200,000 queries, while Inline takes 1170 ms more. To understand these results, recall the three major differences between Inline and SP.

- **Structure matching and value matching**: Inline performs early predicate evaluation before knowing if the structure is matched, and this early predicate evaluation does not prune future work. In contrast, SP performs structure matching to prune the set of queries for which predicate evaluation needs to be considered.
- **Conjunctive predicates in a query**: In Inline, evaluation of predicates in the same query happens independently at different states, while in SP, the failure of one predicate in a query stops the evaluation of the rest of predicates immediately.
- **Bookkeeping**: Inline requires bookkeeping information for the final evaluation of a query. The maintenance cost includes setting the information and undoing it during backtracking. Note that in addition to reduced MQPT, bookkeeping overhead causes Inline to run out of memory, for Q above 400,000.

When the number of predicates per query is doubled (P=2, also shown in Figure 5.4) both approaches suffer an increase in MQPT. The differences between the approaches, however, are more pronounced. For example, for 200,000 queries containing two predicates each, Inline takes 1534 ms more than SP. Inline also experiences a tremendous increase in the bookkeeping overhead, and runs out of memory with 100,000 queries earlier than P=1.

Figure 5.5 shows the MQPT of the two approaches as the number of predicates per query is varied from 0 to 20 for a relatively small number of queries (Q =50,000). As can be seen in the figure, a large number of predicates compounds the poor performance of Inline. In contrast, SP is much less sensitive to the number of predicates per query. As P increases, the increased cost in SP results from a larger number of invocations of predicate evaluation and longer evaluation periods. Luckily, the negative impact is limited by the short-cut evaluation strategy.

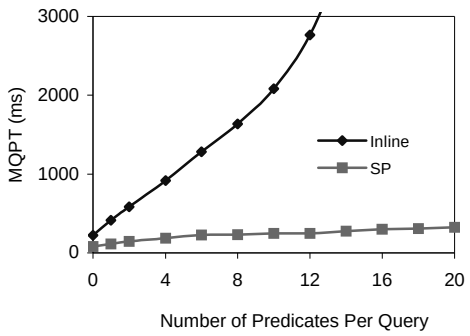


Figure 5.5: Varying number of predicates (D=6, Q=50000, W=0.2, DS=0.2)

The previous experiment demonstrated the benefits of delaying content-based matching in YFilter. One of the major benefits was seen to be the ability to "short-cut" the evaluation process for a query when one predicate fails. This observation raises the potential to further improve the chances of such short-cut evaluation by evaluating highly-selective predicates first, as is done by most relational query optimizers.

If statistics on documents are kept, then the selectivity of predicates on attributes can be estimated from the probability of an attribute occurring in an element and the number of values this attribute can take. Examples of equality predicates on attributes of element "a" are given as follows:

selectivity ([***@attr***]) = probability of the attribute occurring in element '*a*'.

selectivity ([***@attr='v***']) = *Selectivity*[*@attr*] / max. # of values attribute "*attr*" can take.

The selectivity estimates for predicates involving other comparison operators can be derived in a similar way. If predicates are attached to a wildcard in a location step, simple assumptions are made about their selectivity. Other formulas are omitted here.

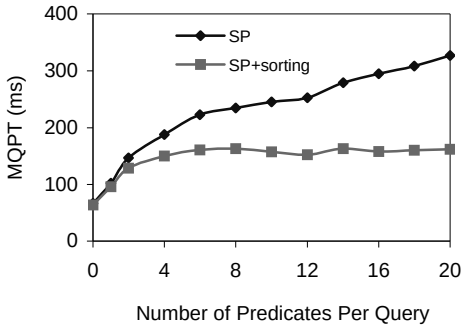


Figure 5.6: Effect of predicate sorting (D=6, Q=50000, W=0.2, DS=0.2)

A simple experiment was performed to examine the potential performance benefits of predicate reordering in the SP approach. Figure 5.6 shows the MQPT for SP with and without sorting, as P is varied from 0 to 20 for Q=50,000. The results indicate that as expected, additional benefits can indeed be gained by predicate sorting, particularly for cases with large numbers of predicates.

5.2 Nested Path Expressions

In the previous section, I described two approaches for value-based predicate evaluation in YFilter. As the experimental results show, SP outperforms Inline by a wide margin. The key feature of SP is that predicate evaluation is postponed until after path matching and is performed by "post-processing" the path matching results. YFilter's technique for handling nested path expressions leverages this post-processing of path matches.

5.2.1 Preliminaries

I begin the discussion on supporting nested paths by more clearly specifying the interface between the NFA-based path matching and the post-processing operators. This interface is based on *path match* structures that identify the document elements that caused the NFA to

reach an accepting state. During parsing document elements are given unique identifiers. Each time an accepting state is reached, the NFA outputs a path match structure for each query associated with that state. At an accepting state that represents a path expression of *n* location steps, each structure generated is simply a list of identifiers of the *n* elements that matched the path expression.

The elements that path matches reference are stored in memory resident data structures created in document parsing. These data structures hold attributes and text data of the corresponding elements which could be used by any operators in post-processing.

5.2.2 Query Decomposition

The original work on XFilter [Altinel and Franklin, 2000] proposed using *query decomposition* to handle nested path expressions. In this approach, the nested paths are extracted from the main path expressions and processed individually. A post-processing phase is used to link matched paths back together to determine if an entire query expression has been matched. The advantage of such an approach is that the path matching component remains untouched. YFilter follows a similar approach, using the NFA/post-processing interface described above. In YFilter, however, this approach has the significant additional benefit that it naturally allows shared path matching to be exploited for nested path expressions.

In the following, I describe the approach by addressing how the nested paths are represented and how they are evaluated. I also present results from a performance study of the implementation of nested paths in YFilter.

5.2.3 Query representation

For ease of exposition, I initially assume only one level of path nesting in queries. In other words, a nested path does not itself contain any nested paths. I then relax this assumption in Section 5.2.5. For such queries, I define three terms: A *main path* is the remaining structure of a query after all the nested paths are removed. An *anchor step* of a nested path is a location step in the main path where that nested path is attached to the main path. An *extended nested path* is a nested path pre-pended with the prefix of the main path up to its anchor step.

In this approach, when a query containing nested paths is parsed, it is decomposed into a list of absolute paths: the main path and any extended nested paths. For example, consider:

Query 7 = “/a[d]//b[e/f]/c”

It contains two nested paths “d” and “e/f”. Query decomposition produces a main path, “/a//b/c”, and two extended nested paths, “/a/d” and “/a//b/e/f”. These paths are assigned identifiers consisting of (*QueryId, PathId*) pairs, where the main path has PathId 0 and the nested paths are numbered sequentially. All of the paths are then inserted individually into the NFA with these identifiers. The interface described above is slightly extended here so that the NFA returns path matches to queries using the Query Ids with the Path Ids attached to the matches for the use inside those queries.

Post-processing is implemented using operators called *Nested Path Filters* (NP-Filter). Each NP-Filter is associated with a single query. Under the assumption of a single level of nesting, only one NP-Filter is required per query. The NP-Filter contains information for

each path of its associated query. For each nested path, it stores the *position* of its anchor step in the main path. This position will identify the last element shared between the extended nested path and the main path. The NP-Filter also contains for each path (main and nested) a *store* to keep the path matches corresponding to that path.[10]

5.2.4 Query evaluation

As previously stated, queries containing nested paths are processed in two phases, path matching and post-processing of the path matching results. The first phase is completely done by the NFA as described in Section 4.3. Thus, the processing of the common prefixes is shared among all the paths, e.g., between main paths and extended nested paths and among the extended nested paths themselves. Upon obtaining a new path match, the NFA delivers it to the queries containing the path, together with the PathId of this path in each of those queries. The recipient queries hold this path match in one of its stores identified using the attached PathId.

Post-processing is performed inside each NP-Filter at the end of document processing. This processing consists of the following steps:

1) *Store check*: If any of the stores of the constituent paths of the query is empty, then return *False*.

[10] In the implementation, a path match store is allocated for each unique path expression and shared among all queries containing this path, so an NP-Filter only contains pointers to these shared stores. Due to this sharing, the stores contain path matches in their entirety, even though any one query may not need all of the elements.

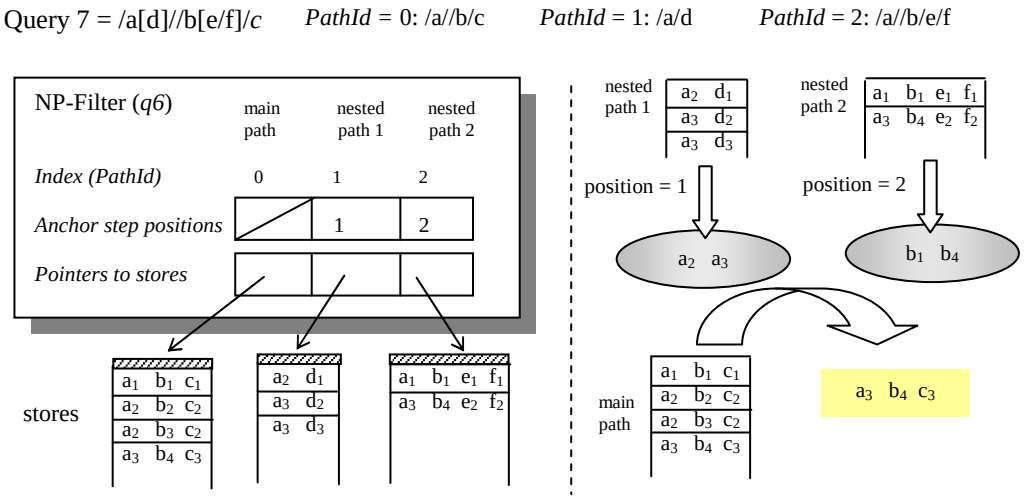


Figure 5.7: An example NP-Filter operator and its match filtering process

2) *Filter construction*: Otherwise, a filter is constructed for each nested path from its corresponding store by extracting the set (no duplicates) of element ids that appear at the anchor step position of the nested path.
3) *Match filtering*: The path match structures of the main path are then pipelined through all the nested path filters. For each main path match, a nested path filter is applied to the element identifier at the corresponding anchor step position. If the filter does not contain this element identifier, the main path match is evicted. If a main path match passes all the filters, the query is evaluated to *True* and the NP-Filter stops.

Figure 5.7 shows the three constituent paths of Query 7 and an NP-Filter operator for it, and illustrates the post-processing performed for this query. On the left of the figure, data structures maintained in the NP-Filter are shown in the upper box. In the list of anchor step positions, the list element at index 1 corresponds to the first nested path (i.e., *PathId* 1), indicating that the anchor step of this nested path is at position 1 in the main path. The list

element at index 2 keeps the position information for the second nested path. In the store list, three pointers link to the stores that contain path matches for the three constituent paths.

The right part of the figure illustrates the execution of the NP-Filter. The arrows drawn top-down depict filter construction for the two nested paths. Anchor step positions are used to extract element ids for each filter. The arrow below the filters illustrates pipelining the main path matches through these two filters. The first main path match is eliminated by the first filter, because the identifier of the ‘a’ element in this match is not contained in the filter. The next two matches are removed by the second filter. Finally the last main path match passes both filters, and the query is evaluated to *True*.

5.2.5 Support of Multiple Levels of Path Nesting

In the above description, I assumed that nested path expressions do not themselves contain nested paths. The approach, however, can be extended to support an arbitrary number of levels of path nesting. First, NP-Filter operators are modified so that they can be configured to output one or all the matches retained from the nested path filters. The rest of the extension is outlined as follows.

For each query involving multiple levels of path nesting, an NP-Filter is assigned to each path expression (absolute or nested) that contains nested paths in its predicates. If additional NP-Filters are assigned to the nested paths of this path expression, the NP-Filter of this path expression treats them as child operators. In this way, a hierarchy of NP-Filters is formed in correspondence to the hierarchy of path nesting.

During post-processing, the hierarchy of NP-Filters is executed bottom up. NP-Filters at the bottom level of the hierarchy access path match stores and perform match filtering as described above. They output all main path matches that are retained from their nested paths

filters. After NP-Filters at the next level receive the path matches from their child operators, they start the execution and output in the same matter. This process continues until the top-level NP-Filter finds any main path match or exhausts all the input matches. The query is evaluated to *True* in the first case and *False* in the second case.

5.2.6 Evaluation of Nested Path Expressions

This section presents a performance analysis of the YFilter approach to processing nested path expressions. Recall that in this approach, path matching is shared among all queries, and post-processing is performed on a per-query basis. This experimental study provides some understanding of the component costs as well as total processing cost in MQPT.

The parameter NP (see Table 2) was used to generate a number of nested path expressions in each query. Such nested paths are distributed over the location steps uniformly at random. The depth of a nested path is determined by the difference between maximum depth D and the actual depth of the location step where this nested path is attached. The setting of parameters W and DS, is also applied to the nested paths. All queries used in the experiments contain only one level of path nesting.

Varying Q and varying NP. In this experiment, the number of distinct queries was varied from 1000 to 200,000 for three values of NP, 1, 2 and 3. Figure 5.8 shows YFilter's performance in terms of MQPT.

An important trend is observed from this figure. As the number of queries grows, there is a fair amount of increase in MQPT to process the first nested paths in queries (see the case of NP=1). Processing additional nested paths in queries (in the cases of NP>1), however, costs only a little more than processing the first nested paths. Consequently, the cases of larger NP values exhibit efficiency and scalability very close to that in the case of NP=1.

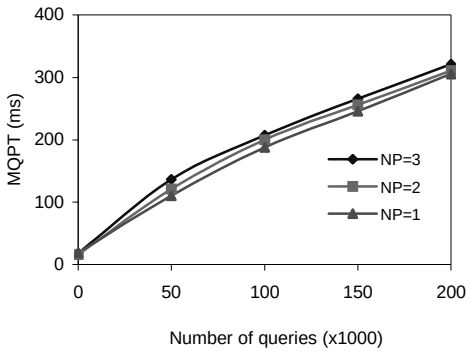


Figure 5.8: Varying number of queries (D=6, W=0.2 DS=0.2, P=0)

For a better understanding of this result, a profiler was implemented to report the costs of NFA-based path matching and NP–Filter operators, and to also provide statistics that help explain the observed execution costs. The above experiment was re-run with the profiler turned on. Due to the overhead of running the profiler, the costs reported in this manner are higher than the costs observed while running the actual experiment. As a sample of the content of the report, Table 6 shows the total cost of path matching, the total cost of all NP-Filters, and some statistics at Q=50,000.

As Table 6 shows, when NP=1, the path matching component costs much more than NP-Filters. The performance study in Section 4.4 has demonstrated that the NFA execution is very efficient. Here, the path matching cost is dominated by generation and delivery of multiple path matches during each of the 5988 visits to accepting states. In contrast, NP-Filters have a relatively low cost, due to the use of the store check as the first processing step. In this experiment, most queries cannot have both constituent paths satisfied by a document, so their NP-Filters only need to perform the inexpensive store check.

From the cases of NP=2 and NP=3 in Table 6, it can be observed that the effects of adding more nested paths are two-fold. First, it increases the cost of path matching, e.g. from

152 ms when NP=1 to 171 ms when NP=3. An analysis of this cost increase is the following. After the additional nested paths are added to the path matching component, they increase the size of the NFA as shown in Table 6 (by 36% from NP=1 to NP=3). This increase, however, is much less than that of the total number of nested paths, due to the path sharing exploited by the path matching. The increased machine size causes some more visits to accepting states during document processing, e.g., 12% more from NP=1 to NP=3, which in turn results in a slightly higher path matching cost. The small increase indicates that after paying the cost for the first nested paths, queries can obtain matches to most of their additional nested paths at no extra cost. In other words, the cost of processing the initial nested paths can be amortized by additional nested paths in queries.

The second effect of adding more nested paths is the slight reduction of the cost of the NP-Filter operators. The additional nested paths increase query selectivity, as evidenced by the reduced number of query matches shown in Table 6. Due to this increased query selectivity, more NP-Filters can terminate due to store checks, thus improving the overall cost of NP-Filters slightly.

The combination of these two effects determines the small increase in MQPT from processing single nested paths in queries to multiple ones in them.

Q=50,000, NP =	1	2	3
Path matching cost (ms)	152	160	171
NP-Filter cost (ms)	33	30	28
…	…	…	…
# of States in the NFA	42198	48523	57468
# of accepting states hit	5988	6193	6701
# of matched queries	3226	1837	770
…	…	…	…

Table 6: Profile on nested path processing (Q=50,000, D=6, W=0.2, DS=0.2)

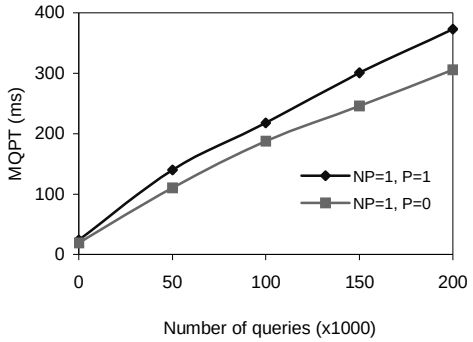


Figure 5.9: Varying number of queries (D=6, W=0.2 DS=0.2, NP=1)

Performance for queries with mixed predicates. This experiment further investigates the performance of YFilter when queries contain both value-based predicates and nested path expressions. To do so, predicate evaluation is integrated into NP-Filters in a way that predicates are applied to path matches immediately after the store check. Thus, predicate evaluation is performed only if all constituent paths in the query are satisfied. Similarly, the later steps of NP-Filter execution, namely, filter construction and match filtering, are executed only when all paths also pass the selection evaluation.

To examine the performance of mixed predicates, query sets were obtained from the case of NP=1 of the previous experiment, and extended by adding a single value-based predicate to the main path of each query. Then the experiment was run by varying the number of queries from 1000 to 200,000 for the two cases, (NP=1, P=0) and (NP=1, P=1). Figure 5.9 shows the MQPT results.

It can be seen that adding a value-based predicate to queries containing nested paths incurs only a very modest increase in MQPT. This phenomenon can be explained by two factors. First, selection operators (using the SP approach) and NP-Filter operators completely share the overhead of path matching, e.g. the NFA-based path navigation and the more expensive operations to generate and deliver path matches. Second, due to the way that

selection evaluation is combined with NP-Filter execution, much of the predicate evaluation is avoided by the store check performed at the beginning of NP-Filter execution.

Experimental results on nested path processing in YFilter can be summarized as follows: 1) There is a fair amount of increase in MQPT to process the first nested paths in queries. The cost is dominated by the overhead of supporting the interface of returning path matches for post-processing. 2) The cost increase can be amortized through path sharing when processing additional nested paths in queries, which results in good efficiency and scalability in the cases of multiple nested paths per query. 3) This cost increase can also be recovered when processing value-based predicates.

5.3 Related Work

In the past few years, there have been a number of efforts to build large-scale, stream-based XML query processing systems. While most of these systems support both structure and value matching to some extent, they have tended to emphasize either the matching of the structure of path expressions [Altinel and Franklin, 2000; Chan et al., 2002; Bruno et al., 2003; Green et al., 2003], or the processing of value-based predicates [Chen et al., 2000; Pereira et al. 2001]. YFilter is, to the best of my knowledge, the first study focused on alternative approaches to combined structure and value-based matching of queries.

MatchMaker [Lakshmanan and Parthasarathy 2002], addresses both issues but focuses on disk-oriented solutions with performance characteristics that differ significantly from other stream-based systems. *XPush* [Gupta and Suciu, 2003] builds a pushdown automaton to support shared matching of both structure and value-based constraints. To achieve efficiency, however, it places constraints on the message content and the use of wildcard and descendant operators in queries and requires periodic reconstruction of the machine.

Tree pattern matching in XML databases [Bruno et al., 2002; Jiang et al., 2003] uses stack-based techniques for encoding partial and complete matches of path expressions in a manner similar to Selection Postponed. It is, however, unclear how these techniques can be extended to support shared matching of multiple tree patterns. In addition, these techniques require the use of indexes over XML data and thus are not directly applicable in streaming environments.

5.4 Summary

This chapter presented a study of integrated approaches to handling both structure-based and content-based filtering of XML documents. Two alternative techniques were investigated for integrating value-based predicate evaluation with the NFA-based structure matching. Experimental results comparing these techniques provide a key insight arising from this study, namely, that structure-based matching and content-based matching cannot be considered in isolation when designing a high-performance XML filtering system. In particular, the experiments demonstrated that contrary to traditional database intuition, pushing even simple selections down through the combined query plan may not be effective, and in fact, can be quite detrimental to performance due to the way that sharing is exploited in the NFA, and due to the existence of descendant operators in queries and recursive elements in XML documents. In addition, this chapter discussed how YFilter supports nested path processing and demonstrated that the solution is efficient even for large numbers of queries containing multiple nested paths each.

6 XML Transformation

XML filtering solutions developed to date have focused on the matching of documents to large numbers of queries, but have not addressed the customization of output needed for emerging distributed information infrastructures. Support for such customization can significantly increase the complexity of the filtering process. In this chapter, I present the second major technical component of this dissertation research, which extends YFilter to transform XML messages on a per-query basis.

6.1 Introduction

As described in Section 1.2, a second requirement of XML message brokers is to transform XML messages according to query-specific requirements, in order to provide customized data delivery and to enable cooperation among disparate, loosely coupled services and applications. High-capacity message brokering systems must be capable of supporting potentially tens of thousands of transformation queries. Thus, approaches that process queries individually are not adequate.

Shared processing of path expressions in YFilter has been shown to be an efficient and scalable foundation for XML filtering in the previous chapters. Thus, a starting point of my research on XML transformation is to leverage the YFilter shared path matching engine, and to develop alternatives for building transformation functionality on top of it. In particular, the research presented in this chapter addresses the following fundamental questions:

- How, and to what extent can the shared path matching engine be exploited for customized result generation?

- What additional *post-processing* of path matching output is needed to support message customization, and how can this post-processing be done most efficiently?

By way of answering these questions, this research has explored three techniques that differ in the extent to which they push work down into the matching engine. As is shown later in this chapter, there is an inherent tension between shared path matching and customized result generation. That is, aggressive path sharing requires more sophisticated post-processing.

Given an efficient shared path matching engine, it is easy for post-processing to become the dominant component of query processing cost. In order to reduce the cost of post-processing, the research on YFilter has developed provably correct optimizations based on query and DTD (if available) inspection that enable the system to eliminate unnecessary operations and choose more efficient operator implementations for post-processing of individual queries.

YFilter has also provided a set of techniques for sharing post-processing work across multiple queries. These techniques are similar in spirit to approaches used in more generic Continuous Query processing systems, but are highly tailored for the specific case of large-scale, high-volume XML message brokering.

All of the above techniques have been implemented on top of YFilter's shared path matching engine.

This chapter proceeds as follows. Section 6.2 presents a problem definition. Section 6.3 describes the system architecture. Sections 6.4 and 6.5 discuss three alternative solutions and a set of optimizations for them. Section 6.6 addresses shared post-processing. Section 6.7

presents results of a performance analysis of the above techniques. Section 6.8 concludes this chapter.

6.2 Problem Statement

As described in Section 2.2.2, query specifications for message transformation are written using a subset of XQuery, namely, *for-where-return* expressions. These expressions contain (1) a *for* clause that binds elements matching a path expression to a variable name, (2) an optional *where* clause that uses a set of conjunctive predicates to filter those element bindings; and (3) a *return* clause that retrieves fragments from each element binding using additional path expressions.

For conciseness, I refer to the path expression in a *for* clause as the "*binding path*", those in a *where* clause as "*predicate paths*", and those inside a *return* clause as "*return paths*". Note that the predicate and return paths of a query are relative to the binding path of that query, as they are prefixed by the variable name that is defined using the binding path. Recall that path expressions can specify structural constraints using child "/" and descendent "//" axes and element name tests. For ease of exposition, such paths are referred to as *navigation paths* (as they are used mainly for structural navigation). Path expressions can also contain value-based predicates that compare the attributes or text data of elements to a constant. In this research, binding paths can contain an arbitrary number of value-based predicates in any location step. A predicate path is a navigation path with a value-based predicate attached to the last location step, and itself is a complex predicate imposed on its binding path. A return path is simply a navigation path.

For an incoming message, the output of query processing contains a result for each matched query represented in an intermediate format for efficient creation of the final

customized message. In a result in this intermediate format, the nodes selected from the message are organized into a sequence of groups, such that each group corresponds to a single invocation of the *return* clause. Inside a group, nodes are contained in a sequence of lists. The sequencing of lists corresponds to the ordering of the return paths in the *return* clause. Each list contains the nodes matching the return path in their document order. For example, the output of Query 3 from Section 2.2.2 would have the following format.

…

$section_i$: [$title_{i1}$], [$figure_{i1}$, …]

$section_{i+1}$: [$title_{(i+1)1}$], [$figure_{(i+1)1}$, …]

…

where $section_i$ represents a group, and the numbering …, i, i+1, … represents the ordering of those groups. The sequence inside a group consists of a list of identifiers of *title* nodes (in this example there is only a single *title* per *section*) followed by a list of identifiers of *figure* nodes. In the remainder of this section, this intermediate representation is referred to as the *groupSequence-listSequence* format.

Having described the model of queries and output, I now formulate the XML transformation problem that this chapter addresses as follows:

Given a large set of queries written in the specified query language, for each message arriving at the message broker, efficiently extract message components in the groupSequence-listSequence format for all queries.

Unlike XML filtering, which returns a Boolean result for processing a message against a query, XML transformation requires query processing to identify all the elements matching a query and to retrieve from them specific message components for constructing a customized

message. Query processing as such has to deal with many complex issues such as ordering and duplicates among multiple matches; shared processing of queries in the presence of these issues is even more challenging. As is shown in the subsequent sections, efficient transformation for large numbers of queries requires significantly sophisticated query processing and optimization techniques that previous filtering systems do not provide.

6.3 YFilter Transformation Architecture

In this section, I present an architectural overview of the YFilter transformation system, and provide details on a particular output format that the shared path matching engine provides for use of the transformation extension.

6.3.1 Architectural Overview

The architecture of the YFilter transformation system is shown in Figure 6.1. Similar to the architecture presented in Section 4.2, the primary inputs are queries and XML messages. The output, however, is different in that for each incoming message, multiple customized messages are delivered to the set of relevant users.

As described in Section 4.2, an arriving query is parsed immediately for use by the **Query Compiler**, where the execution plan of the new query is merged into a pool of shared query plans representing all of the queries in the system. For XML transformation, this shared data structure contains an NFA that represents a set of navigation paths extracted from those queries, and in addition, an operator network that handles the remainders of those queries (after certain navigation paths are taken out). Note that query compilation is incremental as before; that is, the execution plan of a new query is merged with the existing queries without recompiling any of them.

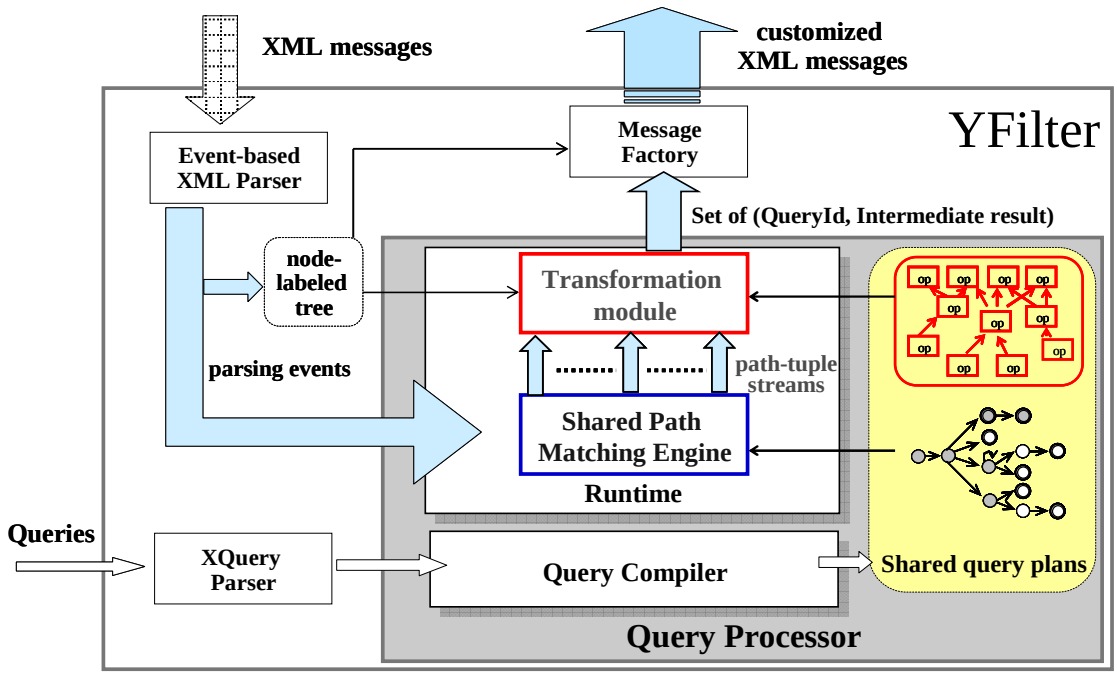


Figure 6.1: YFilter Transformation Architecture

Incoming messages are filtered and transformed on-the-fly for the entire set of queries in the system. These messages need not conform to DTDs or XML schemas but, as described later in this chapter, such conformance can be exploited for query optimization. Internally, a message goes through the following processing steps:

- **Event-based XML parser**: As before, the system runs an incoming message through an event-based XML parser. Parsing events are passed to the runtime system to drive the query execution. In the transformation system, they are also used to incrementally construct a node-labeled tree, which provides materialization of the parsed message for later use. Note that this node-labeled tree is conceptually similar to that presented in Figure 2.2 in Section 2.3, but with two noticeable differences: (1) this tree is constructed gradually as parsing events occur, and (2) it has customized features for high-

performance query processing; in particular, the nodes are assigned integer identifiers according to a pre-order traversal of the tree.

- **Shared path matching engine**: Inside the runtime system, parsing events are passed to a shared path matching engine that runs the NFA as described in Section 4 to match a set of navigation paths. The path matching results are output in a format called *pathtuple streams*, which is explained in the next subsection.
- **Transformation module**: The pathtuple streams are directed to a transformation module, which executes the operator network on those pathtuple streams to generate customized results. Recall that a result is created for each matched query in the GroupSequence-ListSequence format.
- **Message factory**: Finally, the query processing results are fed to the message factory where the processing results are combined with the element tags in queries and the resulting messages are forwarded for delivery.

6.3.2 PathTuple Streams

Algorithms used by the transformation module for customized result generation are developed in the context of a particular output format that the shared path matching engine provides. For a navigation path matched by an incoming message, this engine delivers a stream of “path-tuples” each of which represents a unique match of this path. A path-tuple contains one field per location step in the path, and the value of the field is the identifier of the message node bound to the location step. When multiple paths are matched by a message, the engine delivers its output as streams of path-tuples, one stream for each path.

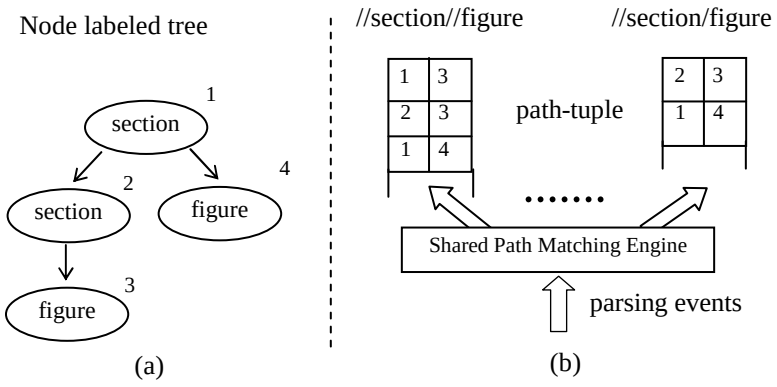


Figure 6.2: An Example of the Path Matching Output

Figure 6.2(a) shows a node-labeled tree for a message fragment, where nodes are annotated with their assigned ids. Path-tuple streams that are output from the engine for different paths are illustrated in Figure 6.2(b). Take the stream for the path "//section//figure". It contains three path-tuples. Each path-tuple contains two node ids, representing a unique combination of the two location step bindings.

The shared path matching engine guarantees that path-tuples in each stream are produced such that the node ids in the last field of the path-tuples appear in monotonically increasing order. This stream order is exploited in the processing algorithms as described in the following sections. It is also important to note that ordering on other fields of path-tuples is not guaranteed by the engine.

6.4 Basic Approaches

In this section, I present three different query processing approaches that differ in the extent to which they exploit the path matching engine. In all of them, a post-processing phase is applied to the output of the matching engine to generate the complete *groupSequence-listSequence* output. Given pathtuple streams, the post-processing is done via query plans

using relational-style operators. In the approaches described in this section, one such query plan is used per XQuery query (i.e., the post-processing phase is not shared). How to share post-processing work is investigated in Section 6.6.

It should be noted that much of the subtlety of developing solutions to this problem arises from the inherent tension between shared processing at the lower level (which is essential for good performance) and customized query result generation. The path matching engine returns the path-tuples in a stream in a single, fixed order to all queries that include the corresponding path. The paths, however, may be used quite differently by the various queries, and thus potential inconsistencies such as unintended duplicates or ordering problems can arise with aggressive path sharing (both of these cases will be discussed in detail shortly). In the following, I describe three approaches in order of increasing path sharing, and focus on how the additional complications raised by increased sharing are addressed. The approaches are additive; that is, the approaches exploiting increased sharing incorporate those that use less.

6.4.1 Shared Matching of "For" Clauses

The first approach uses the path matching engine to process only binding paths (i.e., paths that appear in *for* clauses). In this approach, the navigation part of the binding path from each query is inserted into the engine. Then, during the processing of a message, the output of the engine for each path is directed to the post-processing plans for its corresponding queries. This approach is referred to as *PathSharing-F*. Consider Query 8:

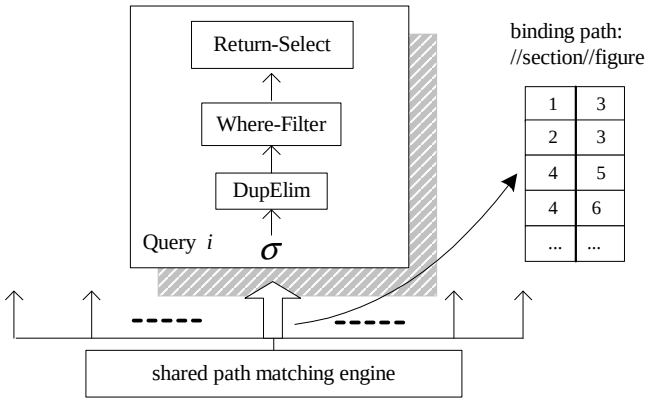


Figure 6.3: A Query Plan using PathSharing-F

```
Query 8:  <figures>
          {
                for       $f  in  $doc//section[@id<=2]//figure
                where     $f/title = "XML processing"
                return    <figure>
                              { $f/image }
                          </figure>
          }
          </figures>
```

Figure 6.3 highlights the post-processing plan for this query under *PathSharing-F*. In the figure, the multiple arrows above the matching engine represent the streams of path-tuples (note that queries that have a common binding path share a common stream). The thick arrow denotes the stream used by Query 8, which contains the path-tuples matching the binding path "//section//figure". In the following, the last field of these path-tuples is referred to as the *binding field*, because they contain the ids of the nodes that are actually bound by the binding paths. These nodes are also referred to as the *BoundNodes*. The box above the thick arrow contains the post-processing execution plan. The operators in this plan are, from bottom-up.

Selection. A selection operator is placed at the bottom of a query plan to evaluate any value-based predicates (i.e., comparisons of the attributes or text data of elements to a constant) attached to a binding path. The evaluation is done for each path-tuple by checking predicates against the nodes referenced by the path-tuple. Selection emits only those path-tuples for which all predicates evaluate to True.

Duplicate Elimination (DupElim). The XQuery specification requires that duplicate nodes bound to a path be eliminated based on the node identity [Boag et al., 2003]. Accordingly, duplicates in the stream for a binding path are defined as path-tuples that contain the same node id in the binding field.

Such duplicates arise when multiple path-tuples in a stream reference the same BoundNode. For example, consider Query 8 and the XML fragment:

> "<section id=1> <section id=2> <figure> <title> XML processing </title> </figure> </section> </section>"

The matching engine outputs two path-tuples for the binding path. The first corresponds to "<section id=1> <figure>" and the second to "<section id=2> <figure>". These two path-tuples reference the same BoundNode, so the second could cause redundant work and produce a duplicate result.

The DupElim operator avoids these problems by ensuring that each BoundNode is emitted at most once. In this case, a simple scan-based DupElim operator can be used because as described in the previous section, path-tuples in the stream are ordered by their binding field. It should be noted, however, that DupElim cannot be pushed before the selection, because it is not known which (if any) of the path-tuples referencing the same BoundNode will pass the selection.

Where-Filter. This operator evaluates the *where* predicates on each path-tuple until a predicate evaluates to False or the entire *where* clause evaluates to True. Path-tuples in the latter case are emitted. For each path-tuple, a predicate path is evaluated with a tree search routine that uses a depth-first search in the sub-tree of the parsed message rooted at the *BoundNode* of the path-tuple. The search routine for a path returns True as soon as any node satisfying the predicate is found.

Return-Select. This operator applies the *return* clause to the *BoundNode*s of the path-tuples that survive the Where-Filter. It uses the tree search routine for each return path. Unlike the Where-Filter, however, the tree search routine here must retrieve *all* nodes matching a return path rather than stopping at the first match.

Return-Select generates results in the *groupSequence-listSequence* format. Each input path-tuple causes the creation of a new group. The ordering of return paths in the query defines the sequence of lists within each group. For each list, the matches of the corresponding return path are placed in the order that they appear in the message.

Recall that the results of a FLWR expression must be ordered in accordance with the order of the variable bindings of the *for* clause. Since the stream for the binding path is ordered in this way, and the remaining processing steps do not change that order, it is guaranteed that the order produced by *PathSharing-F* is correct.

6.4.2 Shared Matching of “Where” Clauses

PathSharing-F only uses the path matching engine to process binding paths. The next approach, *PathSharing-FW*, in addition pushes the navigation part of predicate paths from the *where* clause into the matching engine to exploit further sharing. Recall that predicate paths are defined to be relative to the binding paths. Since the matching engine treats all

paths as being independent, the predicate paths must be first extended by pre-pending their corresponding binding path. For example, consider Query 9:

```
Query 9:  <sections>
          {
                  for       $s  in  $doc//section
                  where     $s/title="XML"
                    and     $s/figure/title = "XML processing"
                  return    <section>
                                  { $s//section//title }
                                  { $s/figure }
                            </section>
          }
          </sections>
```

The first predicate path "/title" is transformed into "//section/title" and the second becomes "//section/figure /title". These extended predicate paths, along with the binding path, are inserted into the matching engine. Note that since common prefixes of paths are shared in the matching engine, the extension of these paths does not add significantly to their processing cost.

As in *PathSharing-F*, the path-tuple streams for each query are then post-processed by a query plan that executes the remaining portion of that query. This arrangement is shown in Figure 6.4. The stream corresponding to a binding path is passed through a selection operator and a DupElim operator as before. The output of the DupElim operator is then matched with the streams corresponding to the predicate paths. The path-tuples resulting from the matching process are piped to a Return-Select that works as described before.

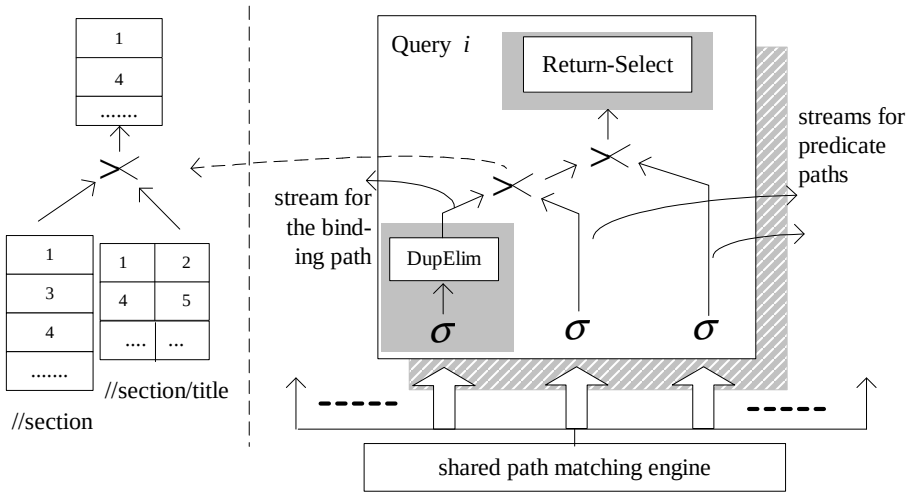


Figure 6.4: A Query Plan Using PathSharing-FW

In *PathSharing-FW*, the Where-Filter of *PathSharing-F* is replaced by a left-deep tree of *semijoins* with the binding path stream as the leftmost input. Recall that the predicate paths are extended by pre-pending them with the corresponding binding path. Thus, the common field on which each semijoin will match is the *binding field*, i.e., the last common field between the binding path tuples and the predicate path tuples. The result of a semijoin, therefore, is a stream containing only those binding path tuples that have matching predicate path tuples. Figure 6.4 shows an example for the leftmost semijoin.

The semijoin operators can be implemented using a simple merge-based algorithm, if it is known that the predicate path streams are delivered in monotonically increasing order of *BoundNode* id. In general, however, there are cases where such ordering cannot be assumed. Consider the execution of Query 9, when applied to the following XML fragment:

"<section> <section> <figure> <title> XML processing </title> </figure> </section> <figure> <title> XML processing </title> </figure> </section>"

In this case, the stream for the predicate path "//section/figure/title" would contain a path-tuple corresponding to "$section_2$ $figure_1$ $title_1$" followed by a path-tuple corresponding to "$section_1$ $figure_2$ $title_2$", where the subscript indicates the first or the second occurrence of the tag name. This stream is not properly ordered by the *binding field* (i.e., section). In such cases, since the binding path stream is ordered properly, PathSharing-FW uses a hash-based implementation of semijoin where the binding path stream is used as the probing stream. Sufficient conditions for determining when the more efficient merge-based approach can be used are discussed in Section 6.5. Note, however, that both approaches order the output correctly, resulting in semantics identical to those provided by *PathSharing-F*.

A final note is that duplicates in predicate path streams are not a concern, because these streams are only used to filter binding path tuples that have passed a DupElim operator.

6.4.3 Shared Matching of "Return" Clauses

The third alternative approach, *PathSharing-FWR*, aims at further increasing sharing by also pushing the return paths into the path matching engine. Return paths differ from predicate paths in that they do not constrain the set of matching binding path tuples so the semijoin approach cannot be used for them. Instead, *outer-join* semantics are required.

Query processing here requires a slightly more specialized operator than a generic outer-join, however, because results must be generated in the *groupSequence-listSequence* format. Thus, this thesis research has implemented an n-way outer-join operator especially for this purpose, which is called *OuterJoin-Select*. As Figure 6.5 shows, OuterJoin-Select takes as its leftmost input, the binding path stream resulting from the semijoins of the *PathSharing-FW* approach. It performs left outer joins on the *binding field* with each of the return path streams. Generation of the results in the required format is performed as part of the outer join

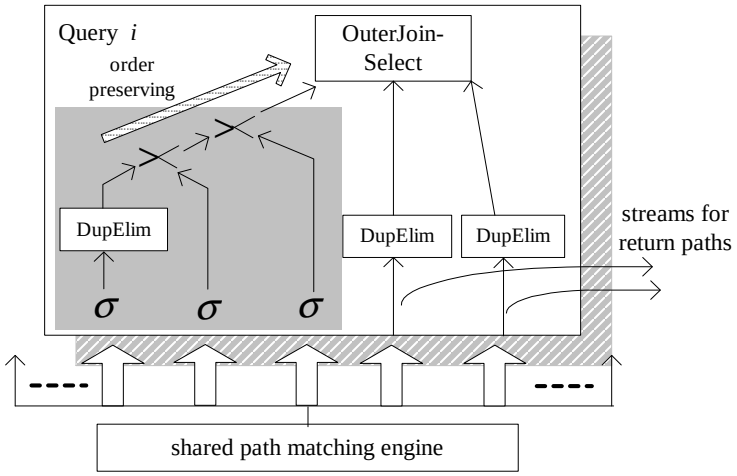


Figure 6.5: A Query Plan Using PathSharing-FWR

processing. Each path-tuple in the binding path stream causes the creation of a new group. The outer join between this path-tuple and a return path stream results in a new list within the group, containing the node ids in the last field in the return path tuples that have matched the binding path tuple. If no such matches are found, an empty list is kept in the group for this return path.

In the implementation of PathSharing-FWR, OuterJoin-Select builds hash tables for each of the return path streams and then probes them in a pipelined fashion using a single scan of the stream emitted by the semijoin tree. In this way, the output of this operator is guaranteed to be ordered by the binding field.

Note from Figure 6.5 that, DupElim operators are required on each of the return path streams to prevent duplicate results from being generated by OuterJoin-Select. Here, the notion of duplicates is defined on the combination of the *binding field* and the last field of the path-tuple, called the *return field*.

Recall that a return path stream is always ordered by the *return field*. If it also arrives ordered by the *binding field*, a scan-based approach suffices for DupElim. Otherwise, hashing is used.

As can be seen, PathSharing-FWR, the approach that exploits path sharing to the fullest extent, requires the most sophisticated post-processing. As mentioned earlier, this complexity results from the tension between shared path matching and result customization. It is important to note that this problem cannot be easily solved in the path matching engine. Consider a path expression that is the binding path in one query and a return path in another. In this case, the path-tuple stream produced for that path expression will be used (by different queries) as two different types of streams. Since the two types of streams have different notions of duplicates, duplicate elimination cannot be done in the engine, but must be done in a usage-specific manner during post-processing. Similar issues arise with the ordering of path-tuples expected by the different uses of the stream.

6.5 Simplifying Post-Processing

Duplicates and stream ordering are two fundamental issues that complicate post-processing for customized result generation. With additional knowledge however, it is sometimes possible to infer cases when duplicates cannot arise, or when path-tuples will arrive in a needed order. In the first case, DupElim operators can be removed from the post-processing plans. In the second case, cheaper scan or merge-based operator implementations can be used in place of the more expensive hash-based ones.

6.5.1 Sufficient Conditions

This thesis research has derived a set of sufficient conditions that enable the detection of some situations where post-processing can be simplified. These conditions involve the presence of "//" axes in queries, and the potential for *recursive elements* (i.e. elements that have the same element name and contain each other) in the messages. The first type requires examining the queries, and the second can be checked by examining a DTD, if present. The claims involving a DTD utilize a *DTD element graph* constructed as follows: Start at the root of the DTD and examine its child elements. If a node for a child element is not in the graph, create one. Then draw a directed edge from the parent element to each child element. Repeat this for all elements.

The conditions are described in the following five claims. Correctness proofs for these claims are given in Appendix E. Consider a path expression p of m location steps, and the stream of path-tuples that match the path, with fields numbered 1..m.

Claim 1: If p contains at most one "//" axis, then there will be no duplicates in the stream of path-tuples matching p when the path-tuples are projected on field m.

Claim 2: If p contains n, $n > 1$ "//" axes, then if the elements of the first $n-1$ location steps containing a "//" axis do not appear on a loop in the DTD element graph, then there will be no duplicates in the stream of path-tuples matching p when the path-tuples are projected on field m.

Claim 3: Partition p into two paths, one consisting of location steps 1 to i, $i < m$, and the other being a relative path consisting of the rest of the path. If claim 1 or claim 2 indicate that no duplicates exist for either path, then there will be no duplicates in the stream of path-tuples matching p when the path-tuples are projected onto fields i and m.

Claim 4: If there is no "//" axis from location steps 1 to *i*, $1 \leq i < m$ of *p*, then the stream of path-tuples matching *p* will be in increasing order when projected onto field *i*.

Claim 5: If *p* contains one or more "//"axes within location steps 1 to *i*, then if for all steps *j*, $j \leq i$ containing a "//" axis, the elements of location steps *j* and *i* do not appear on the same loop in the DTD element graph, then the stream of path-tuples matching *p* will be in increasing order when projected onto field *i*.

6.5.2 Optimization of Post-Processing Plans

The preceding claims enable optimizations of post-processing plans on a query-by-query basis as follows:

- Claim 1 (and 2, if a DTD is present) is used to check if there can be any duplicates in the path-tuple stream for a binding path. Recall that duplicates for binding path tuples are defined on the *binding field,* the last field of binding path tuples. If duplicates are not possible, the DupElim operator for the binding path is removed.
- Claim 3, in conjunction with Claim 1 (and 2, if a DTD is present) is used to check the possible existence of duplicates in the path-tuple stream for a return path. Recall (from Section 6.4.3) that for return paths, duplicates are defined based on the combination of the binding field and the return field. Thus, Claim 3, is tested with *i* set to the location of the binding field. If duplicates are not possible, the DupElim operator for the return path is removed.
- Claim 4 (and 5, if a DTD is present) is used to check if all input streams for a semijoin or OuterJoin-Select are guaranteed to be ordered by the binding field, with *i* set to the location of the binding field. If yes, the merge-based versions of these operators can be

used in place of the more expensive hash-based implementation. These claims are also used to determine if a scan-based DupElim operator can be used for each return path.

Consider the application of these claims for Queries 8 and 9 of the previous section using *Pathsharing-FWR*. Assume that the element "section" is on a loop in the DTD element graph, but the element "figure" is not. For Query 8 (see Section 6.4.1), the tests for Claims 1-3 fail, and in fact, duplicates can arise, as described in Section 6.4.1. The test for Claim 4 also fails because of the "//section//figure" in the binding path. The test for Claim 5, however, succeeds because although the two location steps in the binding path both contain "//" axes and the element "section" is on a DTD element loop, the element "figure" is not on any loop with "section". Therefore all predicate and return path streams are guaranteed to be ordered by the binding field. Thus, cheaper operators can be used for semijoin, Outer Join-Select and the DupElim on the return path stream.

For Query 9 (see Section 6.4.2), if Claim 1 (or 2) and Claim 3 are applied to its query plan, all DupElim operators except the one for the return path "//section//title", can be removed. The remaining DupElim operator results from the presence of two "//"s in the return path and the fact that element "section" after the first "//" is on a DTD loop.

The performance impact of these optimizations can be quite significant, and is studied in the experiments presented in Section 6.7.

6.6 Shared Post-Processing

So far I have presented three ways to share path matching among queries. A common feature of these approaches is that they all require a separate post-processing plan for each query. In this section I describe an initial set of techniques that can further improve sharing by allowing some of the post-processing work to be shared across related but non-identical

queries, in particular, ones that have path expressions (and hence, path-tuple streams) in common.

A prerequisite to the sharing techniques described below is a way to determine which path expressions appear in multiple queries. The technique used here is to associate with each query a set of unique path identifiers corresponding to each of the paths that appear in it. These identifiers are returned by the path matching engine when the paths are initially inserted.

The sharing techniques developed in this research are similar in spirit to techniques developed for shared *Continuous Query* (CQ) processing over (typically non-XML) data streams [Hanson et al., 1999; Liu et al., 1999; Chen et al., 2000; Chen et al., 2002; Madden et al., 2002]. Unlike the *generic* functionality provided in CQ systems, however, the approaches employed here are highly tailored for large-scale XML filtering and customization. For ease of exposition, I focus the discussion on the post-processing plans used by *PathSharing-FWR* with DTD-based optimizations (as described in the previous section), which are shown in the experimental results to outperform the other approaches in most cases.

6.6.1 Query Rewriting

As a first step to enhance sharing among queries, whenever the appropriate DTD is available, path expressions are rewritten into a canonical form before they are inserted into the path matching engine. This rewriting collapses certain expressions that are semantically (but not syntactically) equivalent, allowing their corresponding queries to share a single path-tuple stream for the path. The rewriting focuses on removing superfluous "//" axes. A "//" axis is superfluous if the DTD shows that there is a single path from the element before "//" to the

element after "//". If so, the "//" axis is replaced with the deterministic sequence of '/' steps. For example, a return path "figure//image" can be rewritten to "figure/image" if the DTD shows that an image element can only be the child but not the descendent of a figure element.

6.6.2 Sharing Techniques

Most work on CQ systems considers selection and join operators in a relational (or close to it) framework. In contrast, my research on XML message brokering is focused a subset of XQuery and involves a unique set of operators and a specific data flow through these operators, as presented in Section 6.4. The specialized nature of this work leads to a particular set of sharing techniques, three of which are described below.

Shared GroupBy for OuterJoin-Select: In the implementation as described so far, each OuterJoin-Select operator does its own hashing (or scanning) of the path-tuple streams it consumes for return paths (i.e., all but the leftmost stream). When multiple queries share a common return path, this approach incurs redundant processing. This redundancy can be expensive, because return paths are not constrained by predicates; thus, these streams may carry a large number of path-tuples.

This thesis research proposes to remove this redundancy by placing GroupBy operators before OuterJoin-Selects on those streams that provide return path tuples. A GroupBy operator groups path-tuples in a return path stream by the *binding field*, so that the subsequent OuterJoin-Select can simply get all the return path tuples matching a binding path tuple by obtaining the matching group. Each GroupBy operator is shared by all OuterJoin-Selects that process the corresponding return path. Thus, their overhead is expected to be small. Implementationwise, if the stream of a return path is ordered by the binding field, the

GroupBy is scan based. Otherwise, it is hash based. Duplicate elimination, if necessary, is performed in a scan-based manner in the GroupBy itself.

Having addressed return path processing, I now turn to the post-processing of binding paths and predicate paths.

Selection-DupElim pull up: Shared processing of semijoins among multiple queries is considered first. The common relational optimization of pushing selections below joins makes it difficult to share join processing. Pulling selection up over joins [Chen et al., 2002] avoids this problem. The sharing technique proposed here pulls selections with their subsequent DupElim operators, if present, over semijoins, and turns semijoins into shared joins. Currently this technique is implemented only for queries with a single predicate path.

The technique works as follows. Semijoins are said to have "signatures" consisting of the path ids for their two inputs (a binding path on the left and a predicate path on the right). A shared join is created for all semijoins with the same signature. When converting a semijoin to a join, all path-tuple fields are retained for later use in selections. To be consistent with semijoin semantics, the shared joins are also implemented to preserve the order of the left input stream. The decision on merge- or hash- based implementation carries over from semijoins to shared joins.

Shared selection: Above a shared join operator, selections can be grouped by their signatures [Hanson et al., 1999; Chen et al., 2000; Chen et al., 2002; Madden et al., 2002]. In the XML setting of this research, a predicate signature is a quadruplet (path id, level, attribute name, operator), where the level specifies the location step in the path containing the predicate. For sharing, currently only a single predicate per path is considered. Given this restriction, the signature for a selection above a join is simply the pair of predicate signatures

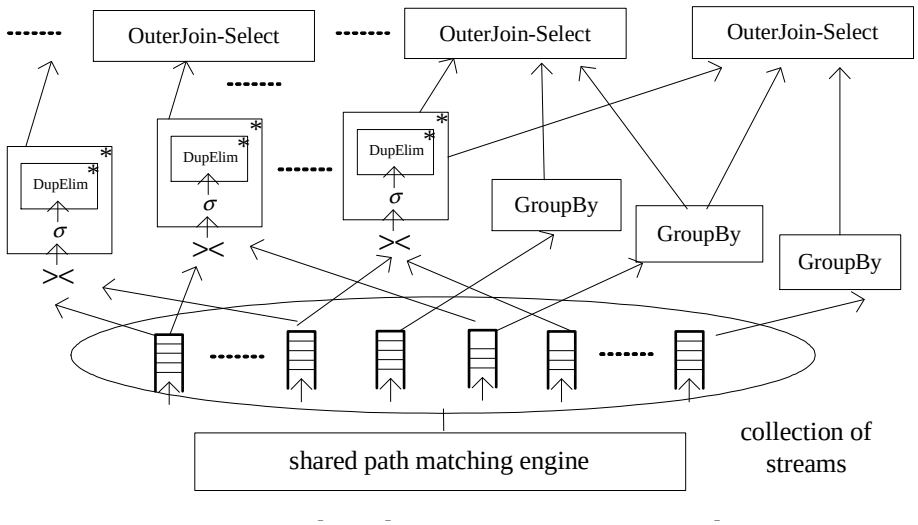


Figure 6.6: Shared Post-Processing Example

from the joined paths. The constant of a selection signature is the pair of constants in the two predicates from the joined paths. Selections with the same signatures are replaced by a shared selection where different constants are merged into a single index. A shared selection can have multiple outputs, one for each constant of the selection signature matched by the XML data.

Shared joins may produce path-tuples containing the same node id in the *binding field*. Fortunately, shared joins preserve the order on the binding field in their output, so scan-based DupElim can be used on the selection outputs.

An example of a shared post-processing plan is given in Figure 6.6. Here a box annotated with '*' means there is a set of such operators. On top of the path matching engine there is a set of merged plans sharing joins and selections, and a set of GroupBy operators shared by OuterJoin-Selects. Each OuterJoin-Select takes the left input from one output of a merged plan and the rest of its inputs from the GroupBys.

6.6.3 Query Plan Construction and Execution

The construction of the shared post-processing plans is done incrementally. When a new query is entered into the broker, a standalone post-processing plan is first constructed for the query. Then its relationship to the current shared plans is determined by examining its path ids and signatures. Operators in the new plan are either merged with existing ones or result in the creation of new branches.

The execution of such large-scale shared query plans is a non-trivial issue. *NiagaraCQ* [Chen et al., 2000; Chen et al., 2002] placed a *split* operator to direct the output of one operator to all the subsequent operators. That operator, however, copies tuples (or pointers to tuples) when multiple subsequent operators require them. This research experimented with a split operator copying tuple pointers in an initial implementation, and found that it imposed a significant performance overhead. *CACQ* [Madden et al., 2002] avoids this problem using *tuple lineage*, which records the operators that a tuple has passed or needs to pass inside the tuple itself. The overhead of tuple lineage, however, increases with the number of queries.

The implementation in this thesis research used an alternative technique that places the pointers to path-tuples in each output of an operator in a data structure called *tpList*, and lets all the subsequent operators share the *tpList*(s) for their input. During query plan construction, each operator allocates one or more tpLists; each subsequent operator must remember which tpList to read from. Most operators have a single tpList. There are two exceptions, however. The path matching engine requires a tpList per path-tuple stream and a shared selection requires a tpList per constant of its signature. The tpLists in the latter cases can be instantiated lazily so they incur overhead only if they are actually used.

During post-processing execution, each operator places the pointer to each output path-tuple to one of its tpLists. Upon completion of an operator, all the subsequent operators read from the desired tpLists and start their execution. A possible disadvantage of this technique is that the scheduler has to check all the subsequent operators even though some tpLists are known to be empty. The experimental results shown in Section 6.7.5 show that this overhead is quite small in practice.

6.7 Experimental Evaluation

The techniques described in the preceding sections have been implemented in the YFilter transformation system using its shared path matching engine. In this section, I present the results of a detailed performance study of this implementation. The performance of the three basic approaches is first compared with and without optimizations when individual post-processing plans are used for distinct queries. Then, the scalability of these approaches and the impact of shared post-processing is examined.

6.7.1 Experimental Setup

The YFilter transformation system was written in Java. All of the experiments were performed on a Pentium III 850 Mhz processor with 768MB memory running IBM J2RE 1.3.0 on Linux 2.4. The JVM maximum allocation pool was set to 600MB, so that virtual memory activity had no influence on the results.

To test the system, generators for both documents and queries are needed. The document generator developed previously for testing the filtering algorithms was used to create XML documents. This generator takes a DTD as input, and produces documents that conform to

that DTD, according to a set of workload parameters. The default settings are used for all those parameters except for the following three.

DocDepth bounds the depth of element nesting in the generated XML documents. The performance study here is less concerned with the absolute document depth, but rather, focuses on the depth of recursive elements. This is because document depth mainly impacts path navigation, while deeply recursive data stresses the post-processing aspects of the transformation solution by requiring DupElim and hash-based operators when "//" axes are used in queries.

The parameter *MaxRepeats* determines the number of times an element can repeat in its parent element. The original generator was modified here so that *MaxRepeats* can be varied on an individual element basis. A large value of *MaxRepeats* produces more matches of a query within a document, generating a larger result set for each matched query.

The parameter *MaxValue* determines the number of values that the data of elements and attributes of elements can take, therefore affecting the selectivity of predicates.

Queries were created using a query expression generator that takes the workload parameters shown in Table 7. This generator ensures that all generated queries are unique. To so do, predicates in the *where* clause are sorted lexicographically. Paths in the *return* clause are also sorted, since two queries that are the same except the ordering of return paths can share most processing with only some trivial reordering at the end. Hashing on the query after path sorting is used to determine if it is unique. Predicates in the generated queries take values from a range of size MaxValue, so this parameter determines the selectivity of predicates. A large value of MaxValue produces fewer matches per query, but also can increase the number of unique queries for scalability evaluation.

Parameter	Values	Description
Q	5,000 – 100,000	The number of distinct queries.
D1	2, 3	The maximum depth of a binding path
PP	1 - 3	The number of predicate paths in a query
RP	1 - 4	The number of return paths in a query
D2	2	The maximum depth of predicate paths or the return paths
DSProb	0 - 0.4	The probability of a "//" axis occurring in any location step in a path expression

Table 7: Workload parameters for query generation

This section reports on experiments with two DTDs: the *Bib* and *Book* DTDs from the XQuery use cases [Chamberlin et al., 2003]. The *Bib* DTD is used to generate non-recursive documents; the *Book* DTD is used to generate documents that can contain multiple levels of recursion. For each DTD, a set of 200 documents were generated using one setting of the workload parameters. For each run, 20 of these documents were used to warm up the JVM runtime compiler. Thus, all reported experimental results represent the average over 180 documents. For each experiment, queries were generated according to a specific query workload setting. For a given experiment, each algorithm was run individually in a separate Java process.

The main performance metric used is ***Multi-Query Processing Time*** (*MQPT*), which is the time from the scan of a parsed document starting until the last result in the *groupSequence-listSequence* format is returned to the calling program. The cost of parsing is not included in my reported results, but was usually below 100 milliseconds.

A profiler was also implemented to report the cost of each operator for a run of an experiment. MQPT times reported here were taken with the profiler turned off. Where appropriate, data from runs with profiling turned on is used to explain the performance results. Due to the overhead of running the profiler, the costs reported in this manner are higher than those observed in the actual experiments.

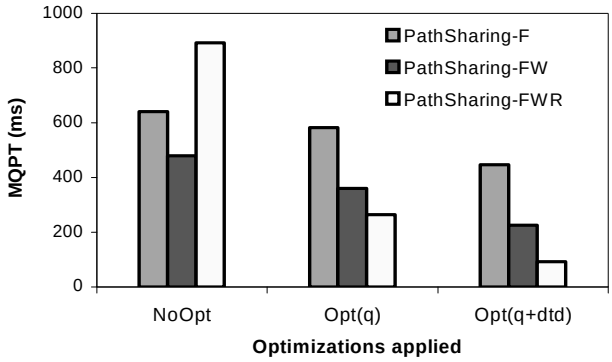


Figure 6.7: MQPT of Three Alternatives (Bib, Q=5000, PP=1, RP=2, DSProb=0.2)

6.7.2 Shared Path Matching – Non-recursive Data

This section reports on tests with the *Bib* DTD, which contains no recursion. For document generation, *DocDepth* was set to 4 because the DTD allows at most four levels of element nesting. *MaxRepeats* was varied such that in each document a bib element contains 20 books and each book has up to five authors or editors. On average, each document contains 149 start/end element pairs. *MaxValue* is set to 10.

Expt. 1 – Basic performance. The first experiment compares the performance of the three approaches for moderate query loads (i.e., Q = 5000). In this experiment, queries were generated using the settings D1 = 2, PP = 1, RP = 2, D2 = 2, and DSProb = 0.2. Under this workload, a single where clause predicate is applied to book elements bound by the for clause. The return clause identifies two types of sub-elements from each remaining book element.

The three approaches were first run with no optimizations. The leftmost group of bars in Figure 6.7 (labeled “NoOpt”) shows their MQPT (in msec). In this case, PathSharing-FW has the lowest cost and PathSharing-FWR has the highest. PathSharing-FW outperforms

PathSharing-F due to the shared path matching for all the predicates. The profiler reports that evaluating all predicate paths using tree search in PathSharing-F takes 386 ms, while for PathSharing-FW, the equivalent work takes only 231 ms (27ms for predicate path matching by the engine, 57ms for selection, and 147ms for semijoins). On the other hand, PathSharing-FW handles return paths using the tree-search based Return-Select operator, at a cost of 212ms, while PathSharing-FWR uses 648ms to perform the equivalent functionality using Outer Join-Selects (244ms) and DupElim for return paths (404ms) (note that there is almost no additional cost for processing the return paths by the engine).

Next, the optimizations described in Section 6.5 were applied. The results are shown in the middle and right groups in Figure 6.7, where Opt(q) indicates optimizations based only on queries and Opt(q+dtd) indicates those also using the DTD. For this latter case, the path rewriting described in Section 6.6.1 was also applied to speed up path matching in the engine and in Where-Filter and Return-Select operators. The following observations can be made:

- The query-based optimizations improve performance for all alternatives, but particularly for those that exploit more path sharing. PathSharing-FWR benefits significantly, outperforming the other two in this case.
- More sophisticated optimizations using the DTD enable further improvements for all three approaches. With these optimizations, PathSharing-FWR outperforms the others by a wide margin.

More detailed results for PathSharing-FWR are shown in Table 8. Three operators, namely, DupElim, semijoin and OuterJoin-Select, particularly benefit from the optimizations. With opt(q), most of the DupElim cost is avoided and the costs of semijoin and OuterJoin-Select are more than halved. When the DTD is also utilized, DupElim is

unnecessary, and semijoin and OuterJoin-Select only each require around 20 ms. Note that the matching engine denoted as PME in the table, is indeed a less dominant component of the overall cost.

Operators	*PME*	*Selection*	*DupElim*	*Semijoin*	*OuterJoin*
No opt	28	61	451	140	235
Opt (q)	27	51	15	67	112
Opt (q+dtd)	9	42	0	18	22

Table 8: Costs (ms) of operators (PathSharing-FWR)

The reduced cost of the three operators is further explained by the change in the resulting query plans, as shown in Table 9. The improvement of DupElim arises because fewer such operators are needed with better optimization. The reduction in time for semijoin and OuterJoin-Select results from the ability to use merge-based implementations more often. For the Bib DTD, since no elements are on a DTD loop, Opt(q+dtd) can completely avoid DupElim and hash based implementations (as described in Section 6.5).

Operators	*Semijoin*		*OuterJoin*		*DupElim*
	#hash	# merge	#hash	#merge	#DupElim
No opt	5000	0	5000	0	15000
Opt (q)	1966	3034	1966	3034	429
Opt(q+dtd)	0	5000	0	5000	0

Table 9: Profile for 5000 queries (PathSharing-FWR)

The above results demonstrate the effectiveness of the optimization techniques. In conjunction with these techni-ques, PathSharing-FWR provides significantly better perfor-mance than the other two alternatives, despite its more complicated post-processing. Thus, the post-processing optimizations help resolve the conflict between shared path processing and customized result generation.

Expt. 2 - Varying the number of predicates. In the next experiment, the number of predicate paths (PP) was varied from 1 to 3. Increasing PP makes each query more selective

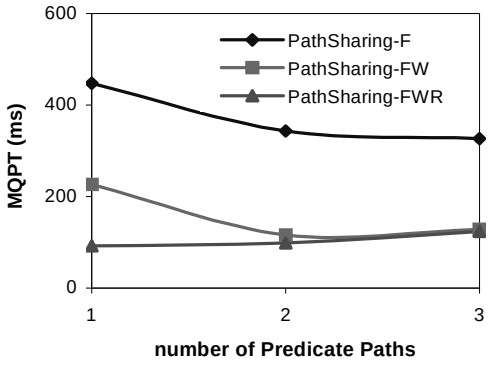


Figure 6.8: Varying PP (Bib, Q=5000, RP=2, DSProb=0.2, Opt(q+dtd))

in addition to requiring more predicates to be evaluated. Figure 6.8 shows the results using Opt(q+dtd).

The main observation is that more predicates reduce the differences among three alternatives. For alternatives using Return-Select, more predicates improve their MQPT because the extra predicates reduce the number of query matches, resulting in much less work for Return-Select. These savings outweigh the modest increase in cost for predicate evaluation. An additional observation is that with three predicates in each query, only 116 matches were found for all 5000 queries, which explains why PathSharing-FW and PathSharing-FWR are so close at that point. In this workload, further increasing the number of predicate paths tends to result in no matches, so the parameter is not further enlarged here.

Expt. 3 - Varying the number of return paths. Figure 6.9 shows the results obtained when the number of return paths in the queries is varied from 1 to 4. Again, only the results for the Opt(q+dtd) case are shown here. In this experiment, the MQPT of PathSharing-F and PathSharing-FW increases linearly because with the fixed query selectivity, more return paths require more executions of the tree search routine. PathSharing-FWR is much less sensitive to the increased workload, because the matching of the return paths is shared

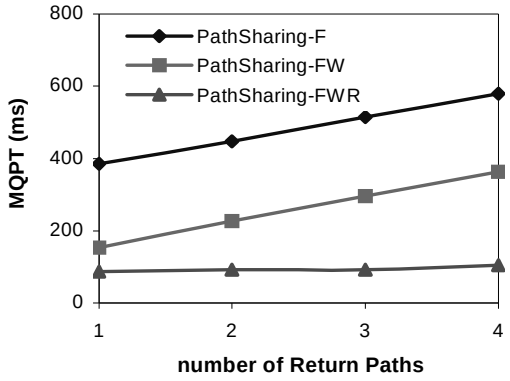


Figure 6.9: Varying RP (Bib, Q=5000, PP=1, DSProb=0.2, Opt(q+dtd))

among 5000 queries. Also, by using a merge-based approach, OuterJoin-Selects are efficient even when the number of streams involved in the outer joins increases.

6.7.3 Shared Path Matching – Recursive Data

In the next set of experiments, the *Book* DTD is used to generate documents with recursive elements. *DocDepth* is set to 5 so that up to four levels of nesting of section elements can be obtained. *MaxRepeats* is set such that there are 12 top-level section elements in each book, and in each section, p (i.e., paragraph), figure, and section elements are allowed to repeat four times. The average document length is 83 start-end element pairs. *MaxValue* is set to 10.

Figure 6.10 shows the MQPT of the three alternatives when queries were generated using the settings: *Q* = 10,000, *D1* = 3, *PP* = 1, *RP* = 2, *D2* = 2, *DSProb* = 0.2. Under this workload, the evaluation of the *for* clause can bind section, paragraph (p), or figure elements to the variable. The results are similar to those of the previous experiments except that with Opt(q), PathSharing-FWR is outperformed by Path Sharing-FW, and with Opt(q+dtd) the advantage of Path Sharing-FWR is less pronounced.

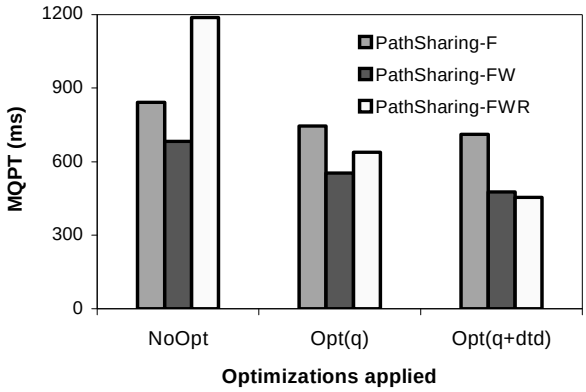


Figure 6.10: MQPT of Three Alternatives (Book, Q=10000, PP=1, RP=2, DSProb=0.2)

The detailed performance of PathSharing-FWR is shown in Table 10. While optimization cuts down the cost of DupElim successfully, the costs of semijoin and OuterJoin-Select remain high. This is due to the recursive *section* elements and a fairly large number of "//" axes in queries (recall that *DS*=0.2 for each location step). In this situation, it is likely that tuples generated for predicate paths and return paths are not ordered by the *binding field*. Consequently, as shown in Table 11, many semijoins and outer joins must be hash based, even when the best optimization is applied.

Operators	*PME*	*Selection*	*DupElim*	*Semijoin*	*OuterJoin*
No opt	36	101	560	225	287
Opt (q)	31	101	82	184	252
Opt (q+dtd)	21	93	30	137	163

Table 10: Costs (ms) of operators (PathSharing-FWR, *Book*)

Operators	*Semijoin*		*OuterJoin*		*DupElim*
	#hash	# merge	#hash	#merge	#DupElim
No opt	10,000	0	10,000	0	30,000
Opt (q)	5963	4037	5963	4037	2833
Opt(q+dtd)	3968	6032	3968	6032	1500

Table 11: Profile for 10000 queries (PathSharing-FWR, *Book*)

To further investigate the impact of "//" axes in the presence of recursive elements, another set of experiments was run with *DSProb* varying from 0.05 to 0.4. The results can be summarized as follows: First, regardless of any optimizations applied, the MQPT for all alternatives increases with the *DSProb*. Second, the difference between Opt(q) and Opt (q+dtd) is more pronounced as *DSProb* is increased. Recall that Opt(q) only checks how many times the "//" axis appears in path expressions. In contrast, Opt(q+dtd) also checks if an element after "//" is allowed to be recursive, and thus can be more effective. Finally, with a very large *DSProb* value, even Opt(q+dtd) has a limited effect. As a result, PathSharing-FWR loses its performance gain over PathSharing-FW, as it requires more DupElims and hash-based outer joins, which offsets the benefit of shared matching of return paths. For example, with *DSProb* = 0.4 (a very high value), PathSharing-FW outperforms PathSharing-FWR slightly (by about 4%).

Note that experiments were also run by varying the number of predicates and number of return paths for the *Book* DTD. The results are similar to those reported for the *Bib* DTD so they are not shown here.

6.7.4 Scalability

Next, experiments were conducted to test the scalability of the approaches in terms of the number of queries (i.e., *Q*). Figure 6.11 shows the MQPT for the three approaches with Opt(q+dtd), using *Bib* documents, as *Q* is varied from 5,000 to 40,000. To create a sufficient number of unique queries here, the *MaxValue* parameter was increased to 100 for both document and query generation; the other parameters are set as in the basic experiment, i.e., Expt. 1.

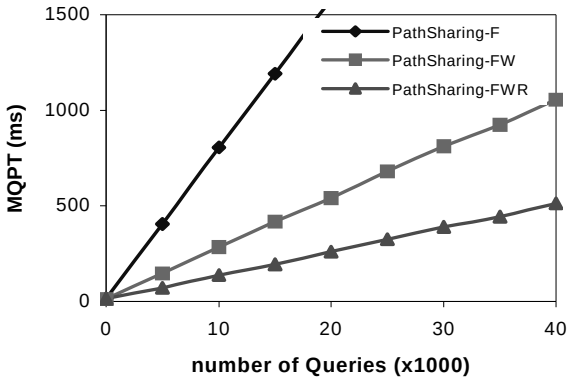


Figure 6.11: Varying Q(Bib, PP=1, RP=2, DSProb=0.2, Opt(q+dtd))

As can be seen in the figure, the MQPT for all three approaches grows linearly with *Q*. Since the solutions studied in this experiment do not share any post-processing, such an increase is to be expected. Note also that the rate of increase is highest for PathSharing-F, which exploits the shared path matching engine the least.

Similar results were obtained using the *Book* DTD, but with an even sharper increase in MQPT due to the additional impact of recursive data on post-processing costs. Table 12 shows the detailed cost breakdown for PastSharing-FWR with Opt(q+dtd) in this case, as *Q* is varied from 10,000 to 50,000. The increasing semijoin and OuterJoin-Select costs become dominant as *Q* increases, while the costs of selection and DupElim also increase. As explained in Section 6.7.3, post-processing is more expensive for the *Book* DTD because of the need for hash-based operators.

Q	**10,000**	**20,000**	**30,000**	**40,000**	**50,000**
Selection	93	191	267	380	498
DupElim	30	62	111	146	183
Semijoin	137	320	484	659	847
OuterJoin	163	364	592	810	1025
Others	…	…	…	…	…
Executor	73	152	182	314	384
Total	**516**	**1111**	**1715**	**2344**	**2985**

Table 12: Costs(ms) as *Q* varies - PathSharing-FWR (*Book* DTD)

6.7.5 On Shared Query Execution

The results reported in the previous section demonstrated the scalability limitations of approaches that share only path matching work. This section examines the additional benefits to be gained by applying the techniques for sharing post-processing described in Section 6.6.

In the following experiments, individual query plans were first generated for PathSharing-FWR with Opt(q+dtd). From these individual plans, shared execution plans were built using the three strategies from Section 6.6.2: pulling selections above joins, grouping selections, and using GroupBy on return paths for outer joins.

The rewriting techniques as described in Section 6.6.1 were also applied the queries to increase commonality. Two effects of this optimization were noticed. First, as expected, it does reduce the number of unique paths. Furthermore, it was observed that some previously unique queries could completely share a query plan because their signatures became identical after this rewriting.

Here, only the results obtained using the (recursive) *Book* DTD are reported (experiments with the *Bib* DTD tell a similar story). Figure 6.12 shows the MQPT of PathSharing-FWR without shared post-processing and with (labeled "Plan Sharing") as the number of unique query plans is varied from 10,000 to 100,000 (note that "*Q*" is roughly 20% higher than this, but those queries sharing query plans with others do not incur extra cost in both algorithms here). As shown in the figure, shared post-processing leads to dramatic reductions in cost and concomitant improvements in scalability; the results here show the PlanSharing approach handling 100,000 unique query plans in only 472ms.

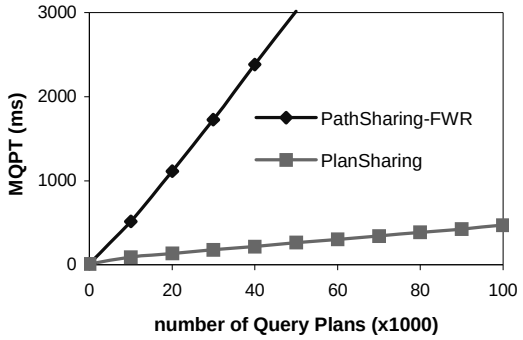


Figure 6.12: Varying number of unique query plans
(Book, PP=1, RP=2, DSProb=0.2, Opt(q+dtd))

Table 13 shows the cost breakdown of PlanSharing. A comparison with Table 12 provides insight into the reduction of the overall cost, which results from four major factors:

- The high cost of semijoins is reduced dramatically, because joins are now shared.
- Grouped selections reduce the selection cost (note that the cost of scan-based DupElim is included in the selection numbers, because it is folded into the selection operator.)
- OuterJoin-Selects are substantially cheaper, because the GroupBy technique removes redundant scanning and hashing at very little cost. Note that OuterJoin-Select is the only operator that exhibits a noticeable increase, as in the current implementation, the outer joins themselves are not shared.
- The Executor's cost is also significantly reduced due to the reduction in query plan size.

Q	**10,000**	**20,000**	**30,000**	**40,000**	**50,000**
(Unique plans)	**(8,232)**	**(16,482)**	**(24,576)**	**(32,736)**	**(40,392)**
Selection	18	18	24	18	21
GroupBy	4	3	5	6	5
Join	18	19	19	21	17
OuterJoin	29	58	81	117	138
Others	…	…	…	…	…
Executor	7	16	22	28	37
Total	**105**	**156**	**212**	**264**	**317**

Table 13: Costs (ms) as *Q* varies - PlanSharing (*Book* DTD)

6.7.6 Summary of Experiments

The experiments reported here have examined the performance of the three alternatives developed for exploiting YFilter's shared path matching engine to provide message broker functionality. These experiments also investigated the performance of a suite of techniques for sharing post-processing among queries. The results can be summarized as follows:

- PathSharing-FWR when combined with optimizations based on queries and DTD usually provides the best performance. This approach is the most aggressive of the three in terms of path sharing. Without optimizations, however, PathSharing-FWR performs quite poorly, due to high post-processing costs.
- Optimization of query plans using query information improves the performance of all alternatives, and the addition of DTD-based optimizations improves them further.
- For non-recursive data, DTD-based optimizations can remove all DupElim and hash-based operators. Recursive data, however, stresses the post-processing of queries containing "//" axes and limits the effectiveness of optimizations.
- Finally, experiments on extending PathSharing-FWR with shared post-processing showed excellent scalability improvements, allowing the processing of 100,000 queries in less than half a second.

6.8 Related Work

Continuous Queries (CQ) systems support shared processing of multiple standing queries over (typically non-XML) data streams. The concept of expression signatures was introduced by *TriggerMan* [Hanson et al 1999]. Using such expression signatures, *NiagaraCQ* [Chen et al., 2000; Chen et al., 2002] incrementally groups query plans. *CACQ* [Madden et al., 2002]

further combines adaptivity and grouping for CQ, and supports the sharing of physical operators among tuples. *OpenCQ* [Liu et al. 1999] uses grouped triggers for CQ condition checking. YFilter's techniques for sharing post-processing, though similar in spirit to those used in some of these systems, are developed particularly for XQuery processing.

In the context of XML stream processing, a number of XQuery processors have been developed. Some recent work uses transducer-based mechanisms for processing path expressions with qualifiers [Olteanu et al., 2003] or XQuery containing FLWR expressions [Ludascher et al., 2002]. *Tukwila* [Ives et al. 2002] represents queries using several individual FSMs that are generated on the fly. The *BEA Stream Processor* [Florescu et al., 2004] executes queries in a pipelined fashion to the extent possible. These approaches, however, are developed for single query processing.

Multi-query processing [Rosenthal and Chakravarthy, 1998; Roy et al., 2000; Sellis, 1988] considers small numbers of queries (e.g., 10's) and uses heuristics to approximate the optimal global plan. In contrast, high-volume XML message brokering presented in this chapter needs to handle sets of queries orders of magnitude larger in a dynamic environment. Thus, scalability of the approach and incremental construction of query plans are two main concerns unique to XML message brokering.

6.9 Summary

This chapter presented a shared processing approach to result customization in the context of high-volume XML message brokering. This approach is the first in the literature that can support transformation of XML messages for large numbers of queries. In particular, this chapter compared three different ways of exploiting the shared path matching engine of YFilter for this purpose. The results of a thorough performance study show that the most

aggressive of the three in terms of path sharing performs best, when combined with optimizations based on the queries and DTD. Moreover, when post-processing is also shared among queries, excellent scalability can be achieved. As a result of applying these techniques, the YFilter transformation system can handle tens of thousands of queries in sub-second response time.

7 Internet-Scale XML Data Dissemination

The preceding three chapters described how YFilter supports efficient filtering and transformation of XML messages for a large set (e.g., tens of thousands) of queries. YFilter, however, does not address the issues of deploying such XML-based services on an Internet-scale (e.g., thousands of information providers and millions of subscribers). In this chapter, I address these issues in the context of incorporating filtering and transformation functionality in a highly scalable system. In particular, this chapter presents the architectural design of ONYX, a distributed system built on an overlay network of brokers running YFilter, identifies the technical challenges in supporting rich data dissemination functionality in this environment, and discusses a suite of techniques that have been developed to address these challenges.

7.1 Introduction

A large number of emerging applications, such as mobile services, stock tickers, sports tickers, personalized newspaper generation, network monitoring, and electronic auctions, have fuelled an increasing interest in *Content-Based Data Dissemination* (CBDD). CBDD is a service that delivers information to users (equivalently, applications or organizations) based on the correspondence between the content of the information and the user data interests. Publish/subscribe systems, especially recent XML-based message brokering systems, are well suited for providing such content-based data dissemination services.

While filtering and transformation of XML messages has aroused significant interest in the database community, little attention has been paid to deploying such XML-based services

on an Internet-scale. In the latter scenario, services are faced with high data rates, enormous query population, variable query life span, and tremendous result volume. Distributed publish/subscribe systems developed in the networking community [Oki et al., 1993; Aguilera et al., 1999; Banavar et al., 1999; Carzaniga and Wolf, 2003; Carzaniga et al., 2004] have demonstrated their scalability in applications such as sports tickers at the Olympics [Gryphon, 2002]. Their services, however, are limited by the data semantics and query expressiveness that they support. Based on these insights, integrating XML filtering and transformation into distributed environments appears to be a natural approach to supporting large-scale XML dissemination.

7.1.1 Challenges

Distributed publish/subscribe systems partition the query population to multiple nodes and direct the message flow to the nodes hosting queries based on the content of messages (which is referred to as *content-driven routing*). Integrating XML into content-driven routing, however, brings the following key challenges.

- As XML mixes structural and value-based information, content-driven routing needs to support constraints over both. The inherent repetition and recursion of element names in XML data also defeats well-known routing techniques (e.g., the counting algorithms [Fabret et al., 2001; Carzaniga and Wolf, 2003]) designed for simpler data models. New techniques for XML-based content-driven routing are needed.
- When XML transformation is introduced to a distributed system, the best venue to perform such transformation is another issue to address.
- The criteria used to partition the query population have an impact on the effectiveness of content-driven routing. The mixture of structure and value-based constraints in queries

and the repetition of element names in XML data complicate the query partitioning problem.

- As the verbosity of XML results in large messages and these large messages need to be parsed at each routing step, alternative formats should be considered for efficient XML transmission.

A number of XML query processors are available for supporting message routing and processing in this environment. Among them, the YFilter system described in the preceding chapters is the first that efficiently supports both filtering and transformation. Recall that YFilter represents a set of queries using an operator network on top of a *Non-Deterministic Finite Automaton* (NFA) to share processing among those queries. Using YFilter for distributed XML dissemination then raises the issues of distributing the NFA-based operator network and efficient scheduling of the operators for both query processing and content-driven routing.

7.1.2 Contributions

In this chapter, I present an initial design of ONYX (*Operator Network using Yfilter for XML dissemination*), a large-scale dissemination system that delivers XML messages based on user specifications for filtering and transformation. The contributions of this work include the following.

- ONYX leverages the YFilter system for content-driven routing. In particular, it uses the NFA-based operator network to represent routing tables, and provides an initial solution to constructing the routing tables from a distributed query population.
- ONYX addresses the issue of how to perform incremental message transformation in the course of routing.

- In order to boost the effectiveness of routing, ONYX provides an algorithm that partitions the query population based on exclusiveness of data interests.
- ONYX explores holistic message processing for sharing the work among various processing tasks at a node (i.e., content-driven routing, incremental transformation, and user query processing). Dependency-aware priority scheduling is used to support such sharing while providing a fast path for routing.
- ONYX supports various formats for efficient XML transmission.
- Last but not least, the research on ONYX includes a detailed architectural design of the system and offers mechanisms for building such a system.

The remainder of this chapter is organized as follows. Section 7.2 presents the system model of ONYX. Sections 7.3, 7.4, and 7.5 describe the core techniques of three main components of ONYX, respectively. Section 7.6 provides a detailed broker architecture design. Section 7.7 presents concluding remarks.

7.2 System Model

In this section, I present the operational features of ONYX. ONYX provides content-based many-to-many data dissemination from publishers to end users. It consists of an overlay network of nodes. Most of the nodes serve as information brokers (or brokers, for short) that handle messages and user queries, while a few of them collaborate to provide a registration service. The overview is illustrated in Figure 7.1.

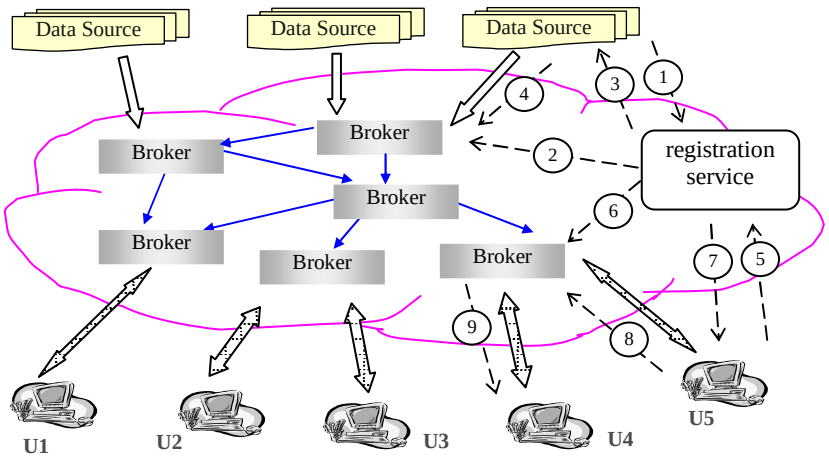


Figure 7.1: Architecture of ONYX

7.2.1 Service Interface

The service interface provided by ONYX consists of several methods (some of which are similar to those in [Altinel et al., 1999]):

Register a data source: A data source registers with ONYX by contacting the registration service and providing information about its location, the schema used, the expected message rate and message size, etc. (as illustrated by message 1 in Figure 7.1). The registration service assigns an ID to the data source, and chooses a broker as the *root broker* for the data source. The choice of the root broker is based on its topological distance to the data source, the bandwidth available, and the data volume expected from that source. After the service forwards the information about the new data source to the root broker (message 2), it returns the assigned ID and the address of the root broker to the data source (message 3).

Publish data: After registration, a data source publishes its data by attaching its ID to each message and pushing the message to its root broker (message 4).

Register a data interest: To subscribe, the user contacts the registration service, and provides his profile including his network address and a query (message 5). The registration service assigns an ID to this profile, and chooses a broker as the *host broker* for this profile based on the user's location and/or the content of his query. At the end of the registration, the service forwards the profile and related information to the host broker (message 6), and returns the profile ID and host broker address to the user (message 7). Thereafter, the host broker will deal with all the user requests concerning that profile.

Update a data interest: Subsequent changes to a profile (including updates and deletion) are sent directly to the host broker (message 8).

Note that users do not need a method to retrieve the messages matching their interests, because those messages are pushed to them from the system (e.g., message 9). Additional methods are provided for data sources to update the schema and other information sent previously.

Fault-tolerance can be achieved by having backup nodes for the registration service and brokers or using other techniques. That discussion is beyond the scope of this work.

7.2.2 Two Planes of Content-Based Processing

ONYX is an application-level overlay network. It consists of two layers of functionality. The lower layer, called the *control plane*, deals with application-level broadcast trees and gives each broker a broadcast tree rooted at that broker that reaches all other brokers in the network. Figure 7.2 shows such a tree in a network consisting of six brokers. Algorithms for constructing broadcast trees have been provided elsewhere (e.g., [Chu et al., 2000]).

This section focuses on the higher layer of functionality in ONYX – c*ontent-based processing*, which is a primary concern of this research. The operations in this layer

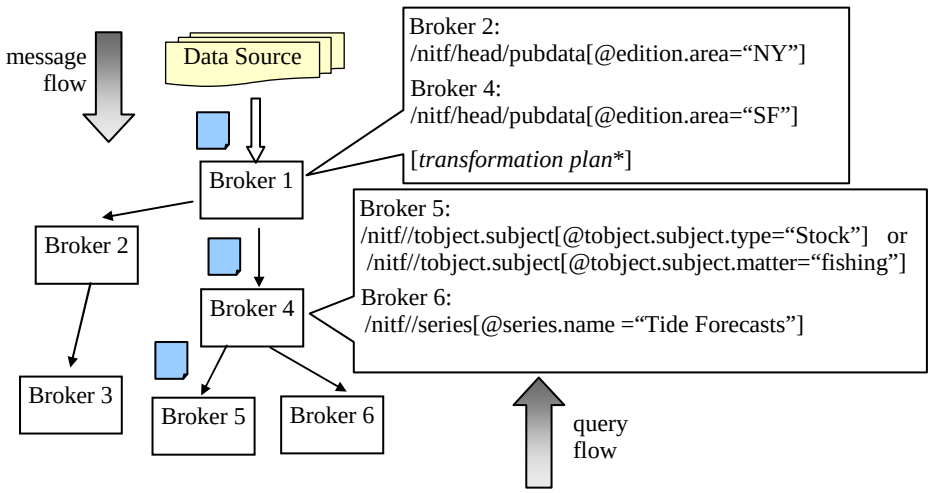


Figure 7.2: Message Routing Based on Content

decomposed into two planes of processing - the *data plane* and the *query plane*. The data plane captures the flow of messages in the system while the query plane captures the flow of queries and query-related updates in the system. As can be seen in this chapter, the duality of data and query is a pervasive feature of ONYX. I now discuss the three tasks performed in this layer – content-driven routing, incremental transformation, and user query processing.

Content-driven routing is necessary to avoid the flooding of messages to all brokers in the network. It builds on top of the broadcast tree described above. The routing is *content-driven* because instead of forwarding a message to all the children in the broadcast tree, a broker sends it to only the subset that is "interested" in the message. This routing scheme, which matches a message's content with routing table entries (or *routing queries*) representing the interests of child brokers, is in sharp contrast to the address-based IP routing scheme.

Figure 7.2 shows an example of routing news articles (written using the NITF DTD [IPTC, 2004]) based on their content. The routing tables for Broker 1 and 4 are shown

conceptually. The table at Broker 1 provides a routing query "*/nitf/head/pubdata* [*@edition.area*= "*NY*"]" for Broker 2, which specifies that Broker 2 and its downstream brokers are only interested in news articles published in the "NY" area. This table contains a similar routing query "*/nitf/head/pubdata*[*@edition.area*= "*SF*"]" for Broker 4. The matching of a new message arriving at Broker 1 with either routing query results in routing the message to the corresponding child.

The building of such routing tables by summarizing the queries of downstream brokers is a subtask in the query plane. The matching of messages against routing queries occurs in the data plane.

Incremental transformation is the second task in the content-based processing layer. Interesting cases of transforming messages during routing include (1) early projection, i.e., removal of data, and (2) early restructuring. An example of early projection is as follows. A data source publishes messages containing multiple news articles. If all the user queries downstream of a link are interested only in a subset of the articles (e.g., those distributed in the area "SF"), messages can be projected onto the articles of interest before they are forwarded along that link using the following query.

```
<batched-nitf>
{
    for     $n   in     $msg/batched-nitf/nitf
    where   $n/head/pubdata/@edition.area ="SF"
    return  $n
}
</batched-nitf>
```

An example of restructuring is message transcoding based on the profiles of wireless users, say, when all users downstream of a link require images and comments to be removed and

tables to be converted to lists. Incremental transformation helps reduce message sizes and avoids repeated work at multiple brokers.

In ONYX, incremental transformation is enabled by attaching transformation queries to the output links of brokers on the path of routing. User queries downstream of a link are aggregated and the commonality in their transformation requirements is extracted to form the transformation query. These subtasks happen in the query plane. The corresponding subtask in the data plane consists of transforming messages using these queries, before the messages are sent to the output links.

User query processing is the task of matching and transforming messages against individual user queries at their host brokers. For the user queries resident at a particular broker, this is the last step of message processing (although the arriving messages may be routed and transformed for other downstream user queries). The subtask in the query plane consists of issues such as indexing of user queries for which the broker is a host broker, and the subtask in the data plane consists of matching messages against these indexes.

Table 14 summarizes the content-based processing tasks in ONYX and their subtasks over the query and data planes.

System Task	**Query Plane**	**Data Plane**
Content-driven routing	build routing tables	lookup in routing tables
Incremental transformation	build transformation plans	execute transformation plans
User query processing	build query plans	execute query plans

Table 14: System tasks over the two planes of processing

In the following sections, I describe three key aspects of ONYX: the query plane, the data plane, which both run at all brokers, and the query partitioning strategy, which is executed separately at the registration service.

7.3 Query Plane

This section focuses on two issues in the query plane: routing table construction and the generation of incremental transformation plans. User query processing is not discussed here, as it is completely handled by YFilter. The solutions addressing those two issues are based on an extension of the YFilter processor. In the following, these solutions are described using a concise model of query representation in YFilter, which allows the reader to ignore many query processing details and to focus on issues relevant to content-driven routing and incremental transformation.

7.3.1 An Operator Network Based Model

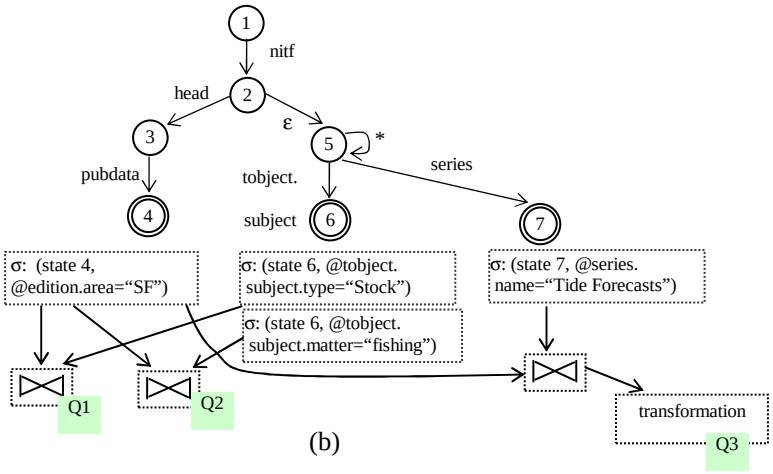
YFilter builds a shared representation for all queries that it contains. At the core, a *Non-Deterministic Finite Automaton* (NFA) is used to represent a set of simple linear paths and support prefix sharing among those paths. While the structural components of path expressions are handled by the NFA, for the remaining portions of the queries, YFilter builds a network of operators starting from the accepting states of the NFA. Each operator performs a specific task, such as evaluation of value-based predicates, evaluation of nested paths, or transformation. The operators residing at an accepting state of the NFA can be executed when that accepting state is reached. Downstream operators in the network are activated when all their preceding operators are finished. In addition, some accepting states and operators are annotated with query identifiers. These identifiers specify that if an annotated accepting state is reached or an annotated operator is successfully evaluated, the queries corresponding to the identifiers are satisfied.

```
Q1:  $msg/nitf[head/pubdata[@edition.area="SF"]]
       [.//tobject.subject[@tobject.subject.type="Stock"]]

Q2:  $msg/nitf[head/pubdata[@edition.area="SF"]]
     [.//tobject.subject[@tobject.subject.matter="fishing"]]

Q3:
  <nitf>
  { for  $n   in   $msg/nitf
    where          $n/head/pubdata/@edition.area ="SF"
       and   $n//series/@series.name ="Tide Forecasts"
    return         {$n/body/body.content}
  }
  </nitf>
```

(a)



(b)

Figure 7.3: Three Example Queries and their Operator Network Representation

Figure 7.3 shows three example queries and their operator network representation. Take Q1 for example. It contains a root element "/nitf" with two nested paths applied to it. Recall that YFilter decomposes the query into two linear paths "/nitf/head/pubdata [@edition.area="SF"]", and "/nitf//tobject.subject [@tobject.subject.type="Stock"]". The structural part of these paths is represented using the NFA (see Figure 7.3 (b)), with the common prefix "/nitf" shared between the two paths. The accepting states of these paths are state 4 and state 6, where the network of operators (represented as boxes) for the remainder

of Q1 starts. At the bottom of the network, there is a selection (σ) operator below each accepting state to handle the value-based predicate in the corresponding path. For example, the box below state 4 specifies that the predicate on the attribute *edition.area* should be evaluated against the element that drove the transition to state 4. To handle the correlation between the two paths (e.g., the requirement that it should be the same *nitf* element that makes these two paths evaluate to true), YFilter applies a NP-Filter (as described in Section 5.2) after the two selections. In this model, all operators that compare two inputs such as NP-Filters and semijoins (as described in Section 6.4) are simply represented using a join (▷◁) operator. In Figure 7.3 (b), the left most join operator is annotated with the query identifier Q1. This means that if the join is successfully evaluated, then Q1 is satisfied.

The representation of Q2 follows the same two paths in the NFA as Q1 and uses the same selection at state 4 to process the common predicate with Q1, but it contains a separate selection at state 6 to evaluate the different predicate in the second path. A distinct join operator is built on these two selections. The representation of Q3 is similar to that of Q1 and Q2 for the *for* and *where* clauses, but contains an additional box for transformation using the *return* clause.

7.3.2 Routing Table Construction

As stated previously, a routing table conceptually consists of routing query-output link pairs, where each routing query is aggregated from user queries downstream of the corresponding output link. In this research, YFilter was chosen for implementing routing tables. The reasons include: (1) fast structure matching of path expressions, (2) a small maintenance cost upon query updates (both features above were shown in Sections 4 and 5), and (3) extensibility for supporting new operations using operator networks. In the following, I present the

representation of routing tables and mechanisms to construct them. For the purpose of routing, only the matching part of a query, i.e., the *for* and *where* clauses of a query written in XQuery, is considered. This part can be converted to a single path expression with equivalent semantics; therefore, it is referred to as the *matching path* of a query.

In the current design, routing queries are represented using a *Disjunctive Normal Form* (DNF) of absolute linear path expressions. If a matching path contains *n* nested paths, it is decomposed into *n*+1 absolute linear paths (possibly with value-based predicates). The routing query constructed for this matching path is the conjunction of the resulting *n*+1 paths. Multiple routing queries can be connected using *or* operators to create a new routing query. Note that an alternative could be to allow any matching path to be a routing query and use *or* operators to connect them. In comparison, DNF relaxes the semantics of nested paths. The motivation of using DNF is that join operators used to evaluate nested paths are relatively expensive, whereas logical *and* operators between path expressions can be evaluated much more efficiently.

Routing table construction from a distributed query population consists of applying three functions, *Map*( ), *Collect*( ), and *Aggregate*( ), to create routing queries in the chosen form.

- *Map*( ) maps the matching path of a user query to the canonical form of a routing query;
- *Collect*( ) gathers routing queries sent from the child brokers into the routing table of a broker;
- *Aggregate*( ) merges the routing queries in the routing table of a broker with those mapped from the user queries at the broker, and generates a new routing query to represent the broker in its parent broker.

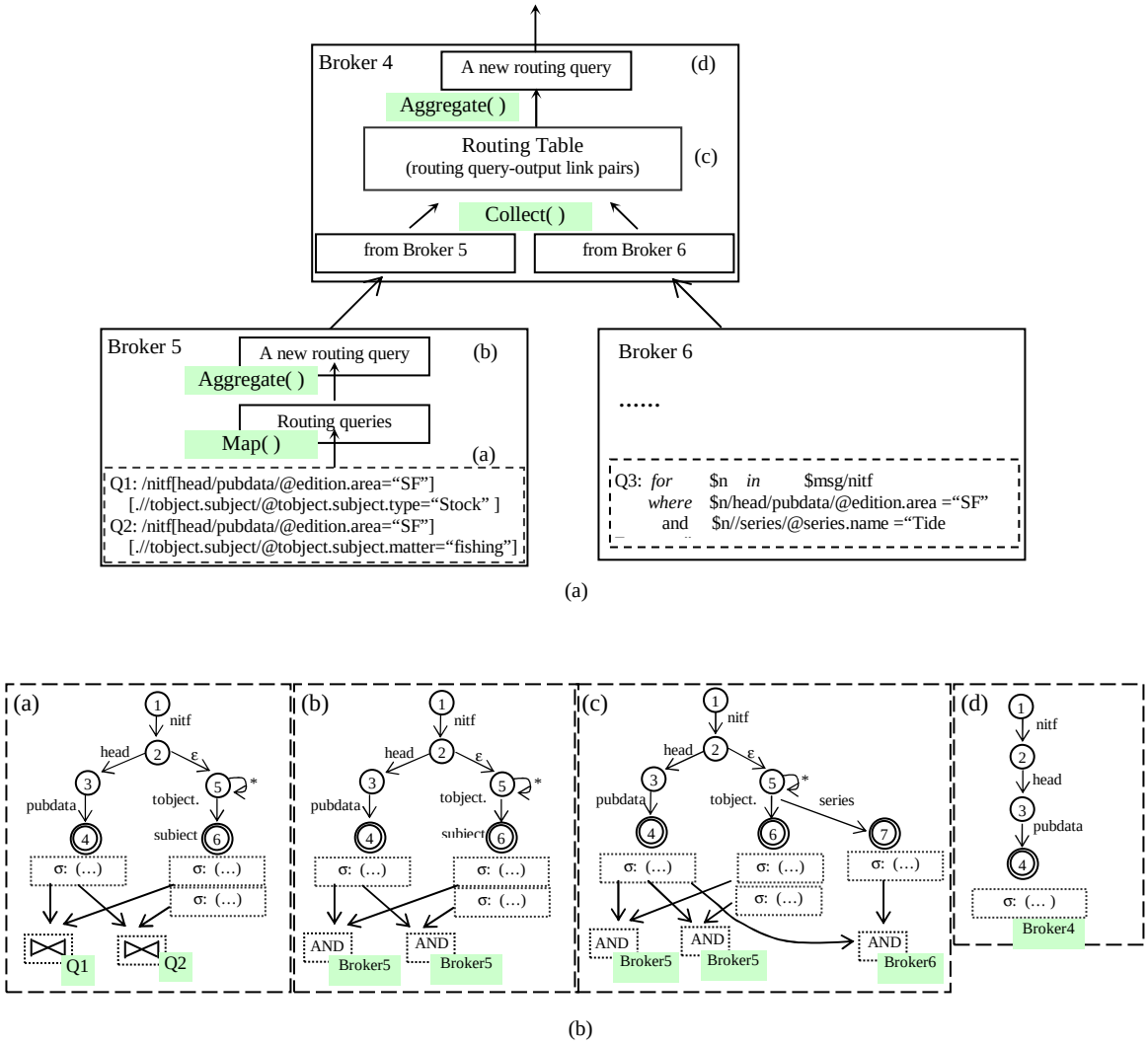


Figure 7.4: Examples of Constructing Routing Tables Using a Disjunctive Normal Form

These three functions are illustrated for Brokers 4 and 5 in Figure 7.4(a). Broker 5 is a host broker with matching paths Q1 and Q2. It uses function *Map*( ) to create a routing query for each of them. Then it applies *Aggregate*( ) to those routing queries to generate a new one that will represent it in its parent (Broker 4). Note that as a leaf, Broker 5 does not contain a routing table. Broker 4 has child brokers Broker 5 and Broker 6, but no user queries. It uses the function *Collect*( ) to merge the routing queries sent from the child brokers into a routing

table, and then applies *Aggregate*( ) to the routing table to generate a routing query that will represent it in its parent.

Construction operations. Next I present the implementation of the three functions using YFilter.

Map( ) takes as input a YFilter operator network representing a set of matching paths. To create the DNF representations of their routing queries, *Map*( ) simply replaces each join operator in the operator network with an *and* operator.

Collect( ) merges routing queries sent from downstream brokers into a routing table of a parent broker. This operation simply merges the YFilter operator networks that represent those routing queries.

Aggregate( ) performs re-labeling on a YFilter operator network. It changes all the identifier annotations (for queries or brokers) to the identifier of this broker, so that the annotated places become marks for routing to this broker. It essentially adds "*or*" semantics to those annotated places, as encountering any one of them can cause routing of messages to this broker. YFilter treats broker identifiers the same as query identifiers, so these identifiers are simply called "targets" in the sequel.

An example is shown in Figure 7.4(b). Box (a) in this figure shows the YFilter operator network built for queries Q1 and Q2 from Broker 5. Box (b) represents the routing query created for Broker 5 after applying *Map*( ) and *Aggregate*( ) to box (a). Box (c) depicts the result of merging box (b) with the routing query sent from Broker 6 (assumed to be the routing query created for query Q3 in Figure 7.4(a)). Box (d), the result of applying *Aggregate*() to box (c), will be explained shortly below.

Sharing among routing queries. Different from the conceptual representation of a routing table (i.e., routing query-output link pairs), ONYX implements a routing table by creating a single combined operator network for all the routing queries. As a result, the common portions of the routing queries will be processed only once. As an example, box (c) in Figure 7.4(b) shows that the path leading to accepting state 4 and the selection operator attached to that state can be shared between the routing query for Broker 5 and that for Broker 6. When the commonality among routing queries is significant, the benefit of sharing can be tremendous.

The *or* semantics introduced to routing queries, however, complicates the issue of sharing. When using separate operator networks for routing queries, a short-cut evaluation strategy can be applied in the evaluation of each routing query. Consider box (b) in Figure 7.4(b) as an operator network created for the routing query for Broker 5. If during execution, one of the two targets labeled as Broker 5 is encountered, the processing for this routing query can stop immediately. In contrast, when using the combined operator network shown in box (c), after a target for Broker 5 is encountered, the processing of the combined operator network has to continue as the target for Broker 6 has not been reached. If care is not taken, some future work may be performed which only leads to the targets for Broker 5. In other words, naïve ways of executing a combined operator network for shared processing may perform wasteful work.

To solve this problem, ONYX employs a runtime mechanism that instructs YFilter to ignore the processing for duplicate targets but not the processing for different targets. This mechanism is based on a dynamic analysis of the operator network which reports the

portions of the combined operator network that will only lead to the targets that have already been reached.

Content generalization. Another issue to address in routing table construction is routing table size (i.e., the size of their operator network representation). Larger routing tables can incur high overhead for routing table lookup, thus slowing the critical path of message routing. They may also cause problems in environments with scarce memory. For these reasons, *content generalization* is introduced as an additional step that can be performed in *Collect*( ) or *Aggregate*( ). Generalizing the routing table essentially trades the filtering power (i.e., the fragment of messages that can be filtered) of the routing table for processing or space efficiency.

An initial set of content generalization methods is proposed in ONYX. Some of the methods generalize individual path expressions with respect to their structural or value-based constraints. Some other methods generalize all the disjuncts in a routing query. For instance, one such method preserves only the path expressions common to all the disjuncts in the new routing query. Consider the routing table shown in box (c) in Figure 7.4(b). When applying *Aggregate*( ) to this routing table, calling this method after re-labelling the identifiers will result in an operator network containing a single path, as shown in box (d). This generalized operator network will be used to represent Broker 4 in its parent.

7.3.3 Incremental Message Transformation

Incremental transformation happens in the course of routing. In this subsection, I briefly describe the extraction of incremental transformation queries from user queries and their placement.

A transformation query for early projection can be attached to an output link at a broker, if (1) its *for* clause is shared by all the user queries downstream of the link, (2) its *where* clause generalises the *where* clauses of all those queries, and (3) the binding of its *for* clause provides all the information that the *return* clauses of those queries require. The last requirement implies that the *return* clauses of the user queries downstream cannot contain absolute paths or the backward axis ".." to navigate outside the binding.

Similarly, a transformation query for early restructuring can be applied to an output link, if conditions (1) and (2) above are satisfied, and (3) the *return* clauses of the downstream queries all contain a series of transformation steps (e.g., removing images and then converting tables to lists), and the first few steps are shared among all those queries. This transformation query will carry out the common transformation steps on matching messages earlier at this broker.

When opportunities for early transformation are identified at host brokers based on the above conditions, incremental transformation queries representing them are generated and propagated to the parent broker. At the parent, these transformation queries are compared and the commonality among them is extracted to create a new transformation query for its own parent and a set of "remainder queries" for its output links. A remainder query is one that combined with the new transformation query constitutes the original transformation query. Each remainder query is attached to the output link where the corresponding original transformation query came from. The new transformation query is propagated up, and the above process repeats.

A final remark is that although the algorithms for routing table construction and incremental transformation plan construction as presented consider all the user queries in a

batch, they can also be applied for incremental maintenance of routing tables or transformation plans. In that case, “delta” routing/transformation queries are constructed and propagated, instead.

7.4 Data Plane

Having described the query plane, I now turn to the data plane, which handles the XML message flow. In the following, I describe two aspects of this plane: holistic message processing for various tasks and efficient XML transmission.

7.4.1 Holistic Message Processing

In ONYX, a single YFilter instance is used at each broker to build a shared, “holistic” execution plan for the routing table, incremental transformation queries, and local user queries (“holistic” means that all these processing tasks are considered as a whole in the data plane). Processing of an XML message using this shared plan is sketched in this section.

The execution algorithm for holistic message processing is an extension of the push-based YFilter execution algorithm. As described in Section 6.3, elements from an XML message are used to drive the execution of NFA. At an accepting state of the NFA, path tuples are created and passed to the operators associated with the state. The network of operators is executed from such operators (i.e., right below accepting states) to their downstream operators. In YFilter, the order of operator execution is based on a First-Come-First-Serve (FCFS) policy among the operators whose upstream operators have all been completed.

In contrast to the earlier work on YFilter, however, the holistic plan contains multiple types of queries, i.e., routing queries, incremental transformation queries, and local user

queries. The first two types are on the critical path of message routing. They should not be delayed by the processing for local queries. Moreover, incremental transformation is useful only if the routing query for the corresponding link can be satisfied, which implies the dependency of transformation queries on the routing queries in execution. For these reasons, a dependency-aware priority scheduling algorithm is proposed here to support shared holistic message processing.

Dependency-aware priority scheduling. In this algorithm, operators that contribute to routing queries are assigned high priority; among other operators, those that contribute to incremental transformation queries have medium priority; and the rest of the operators have low priority. The second priority class, however, is declared to be dependent on the first class with the following condition: an operator in the second class is executed only if at least one incremental transformation query that it contributes to has been necessitated by the successful evaluation of the corresponding routing query. In this implementation, an FCFS queue is assigned to each priority class. In addition, a wait queue is assigned to the dependent class. Priority scheduling works as in a typical OS, except that operators in the dependent class are first placed in the wait queue, and then moved to the FCFS queue when their dependency conditions have been satisfied.

7.4.2 Efficient XML Transmission

Low cost transmission of XML messages is also a paramount concern in a multi-hop distributed dissemination system. XML raises two challenges in this context. First, the verbose nature of XML can cause many redundant bytes in the messages. Second, XML messages need to be parsed at each broker, which can be expensive [Snoeren et al., 2001; Diao et al., 2003]. This section addresses these two challenges.

The inherent verbosity of XML has led to compression algorithms such as XMill [Liefke and Suciu, 2000]. Compression, however, solves only the first of the above challenges but not the parsing problem. A promising approach that is explored in ONYX to counter this problem, is using an *element stream* format for XML transmission. This format is an in-memory binary representation of XML messages that can be input to the YFilter processor without any pre-processing or parsing. The binary format is also more space-efficient than raw XML because the latter has white spaces and delimiters. The "wire size" of an XML message can be further reduced by compressing this binary representation.

Schema-aware representation of XML is also explored for transmission. Given that the control plane can be used to broadcast the schema of a publishing source to all the brokers in the network, ONYX can perform schema-aware XML encoding of messages for transmission between brokers. In particular, ONYX uses a dictionary encoding scheme that maps XML element and attribute names from the schema to a more space-efficient key space. More advanced schema-aware optimizations can be explored to avoid storing parent-child relationships in the binary format, as they can be recovered from the schema.

This experiment compared six XML transmission formats: text, binary (i.e., the element stream format), binary with dictionary encoding, and their corresponding compressed versions. Messages were generated using the YFilter XML Generator based on the NITF DTD. The two parameters - *DocDepth* (that bounds the depth of element nesting in the message) and *MaxRepeats* (that determines the number of times an element can repeat in its parent element) enable the creation of sets of messages with varying degrees of complexity. All the compression was performed using ZLIB, gzip's library, because it outperforms XMill

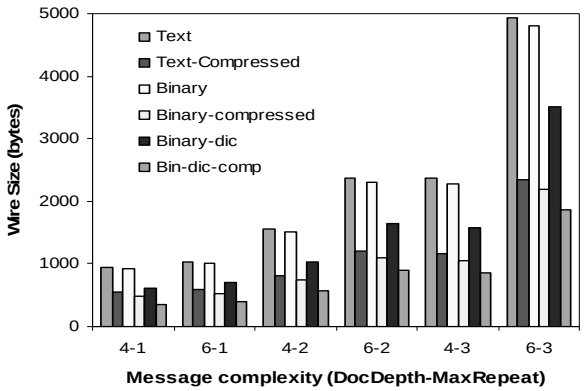


Figure 7.5: Wire size of XML messages

for the relatively small-sized messages (the types of messages that ONYX mostly considers), as reported in [Liefke and Suciu, 2000].

Figure 7.5 summarizes the performance of different XML formats over the first metric, the *wire size*, for messages of different complexities. Although the element stream format does not remarkably outperform the text format, dictionary encoding gives promising results. Compression helps reduce the wire size for all formats significantly.

Figure 7.6 presents the evaluation of these XML formats on the complementary metric of *message processing delay*. While uncompressed formats require only serializing messages at the sender and deserializing them at the receiver, the raw format additionally requires parsing and thus proves to be expensive. Compressed formats have significant costs of compression at the sender and decompression at the receiver.

The choice of XML format for transmission must weigh both the wire size and processing delay metrics to get a combined metric. This decision will invariably be influenced by implementation details like the transport protocol used. For example, in the distributed PlanetLab testbed [PlanetLab, 2005], all the message sizes involved in the above

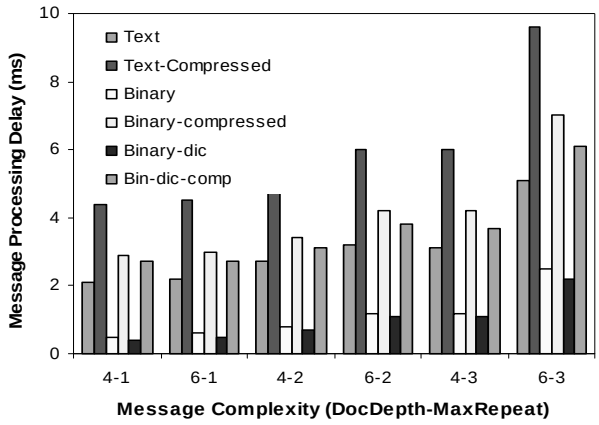


Figure 7.6: Processing delay for XML transmission

experiments gave the same transmission delay using TCP. This was attributed to the connection establishment time dominating in TCP for small message sizes. Thus, the message processing delay turned out to be a more important concern than the message size, making compression rather undesirable. On the other hand, if the DCP protocol [Snoeren et al., 2001] that sends data in redundant streams over UDP can be employed, compression may be useful.

7.5 Query Population Partitioning

Previous work on distributed publish/subscribe [Aguilera et al., 1999; Banavar et al., 1999; Carzaniga and Wolf, 2003] assumes that queries naturally reside on their nearest brokers, without considering alternative schemes for partitioning the query population. This section addresses the effect of query partitioning on the filtering power of content-driven routing, which is captured by the fraction of query partitions that a message can match.

The discussion starts with an investigation of the properties of query partitioning and their effect on content-driven routing. Query similarity within a partition seems to be an

intuitive property, but is not effective in filtering. For example, in the ideal case that all the queries in one partition are “/a/b” and all the queries in the other partition are “/a/c”, a message can still match both partitions by containing the two required elements. Dissimilarity between partitions is another candidate. Consider one partition with two queries “//a” and “//b”, and the other partition with “//c” and “//d”. Though these two partitions have little in common, it is still quite likely that a message matches both partitions. Mutual exclusiveness turns out to be a desired property. For example, if one partition requires “/a/b[@id=1]” and the other requests “/a/b[@id=2]”, the chance that a message satisfies both can be low. The message surely cannot satisfy both if it contains only one “b” element.

The next question is what path expressions can establish such mutual exclusiveness among query partitions. In this regard, three key observations can be made. First, structural constraints alone are not enough (see the first two examples above). This is because an XML schema can not specify that two paths are mutually exclusive in a message. In fact, path expressions exhibit potential exclusiveness if they involve the same structure, and contain value-based predicates that address the same target (e.g., an attribute or the data of a specific element), use the “=” operator, but contain different values (see the third example above). The common part of these paths is called an *exclusiveness pattern*. Second, repetition of element names in XML messages limits the exclusiveness of such patterns. Thus, the best choice of an exclusiveness pattern would be one that can appear at most once in any message, as dictated by the schema. Third, in general the coverage of an exclusiveness pattern in the query population could be rather limited, due to the diversity of user data interests. Thus, using a single exclusiveness pattern for query partitioning could cause the

majority of queries to be placed in a partition called “don’t care”. In that case, a set of exclusiveness patterns should be used.

Partitioning based on Exclusiveness Patterns. To achieve exclusiveness of data interests among query partitions, ONYX uses a query partitioning scheme, called *Partitioning based on Exclusiveness Patterns* (PEP). I briefly describe the two steps of this scheme here, assuming for now that this algorithm can be run over the entire query population in a centralized fashion. (1) *Identifying a set of exclusiveness patterns*. PEP first searches the YFilter representation of the entire query population, and aggregates the predicates contained in the selection operators at each accepting state to exclusiveness patterns. These patterns are sorted by their coverage of the query population (i.e., the number of queries involving them). Then PEP uses a greedy algorithm to choose a set of patterns such that every query involves at least one pattern from the set. Heuristics can be used to perturb this set with other unselected patterns so that more patterns included in the set can appear at most once in a message, but the coverage of the query population is not sacrificed. (2) *Partition creation*. In the second step, K query partitions are created using the M patterns selected in the first step. To do so, the value range of each exclusiveness pattern is partitioned into K buckets, numbering 1, 2, …, K. Then queries are assigned to the $K*M$ buckets based on their values in the contained exclusiveness patterns. As a query must involve at least one of those patterns, it must belong to at least one bucket. If the query involves multiple patterns, it is randomly assigned to one of the matching buckets. Finally, K query partitions are created by assigning the queries in the i^{th} bucket of any pattern to query partition i.

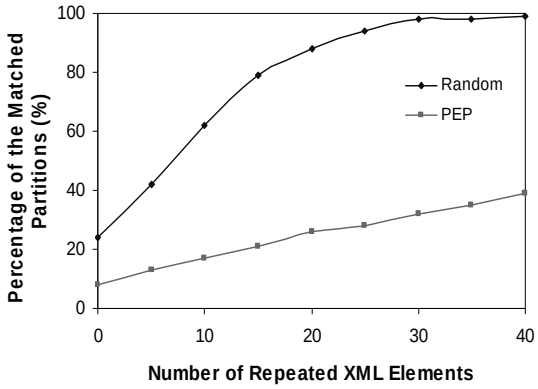


Figure 7.7: Random query partitioning vs. PEP

In the ideal case, where each exclusiveness pattern appears at most once in a message, a message can match at most *M* query partitions, i.e., one bucket per pattern. Thus the filtering power of content-driven routing, i.e., the fraction of query partitions that a message can match, can achieve *M*/*K* (e.g., 10 patterns, 100 partitions, and filtering power ≈ 1/10). If some patterns can appear multiple times in a message, their repetition degrades the filtering power (in many cases linearly).

To study the potential benefit of the PEP scheme, this experiment compared its performance with the random query partitioning scheme that randomly assigns queries to partitions. The case being considered is to assign a population of 1 million queries to 200 partitions. Every query contained two patterns, each chosen uniformly from a set of 10 exclusiveness patterns. PEP exploited these 10 patterns for partitioning. Figure 7.7 shows how the percentage of the partitions that a random message matches varies with the amount of repetition of element names in the XML message. Clearly, the random partitioning scheme ends up matching almost all partitions with messages even with a small amount of repetition of element names. In contrast, PEP leads to many fewer partition matches. Unless

user interests are influenced by geography, a system that assigns user queries to the closest brokers will end up doing random partitioning of queries, leading to many messages being exchanged between the brokers of the system.

An important remark is that in ONYX, PEP is a core algorithm for query placement used by the registration service. In addition to PEP, query placement also involves the decision of mapping query partitions to brokers, and the use of distributed protocols to perform the initial query partitioning and to maintain the partitions as user queries change over time. These issues will be further addressed in future work.

7.6 Broker Architecture

Having described the broker functionality in the query and data planes, I now turn to a discussion of the broker architecture that implements this functionality. This architecture is shown in Figure 7.8. It contains the following components.

Packet Listener. This component listens to each packet arriving at the broker and based on the header, assigns the packet to one of the four flows: catalog packets, XML messages, query packets, and network control packets.

Catalog manager. Catalog packets contain information about a data source. They may originate from the registration service concerning a new data source or from a registered data source to update information sent previously. The catalog manager parses these packets, and stores the information in the local catalog. If the packet is for a new data source, a new entry is added to the catalog including the ID of the data source, information on the data rate, the schema used, etc. If the information relates to a known data source, the existing entry in the catalog describing this data source is updated by the new information. The catalog will be used in other components for message validation, XML formatting, query processing, etc.

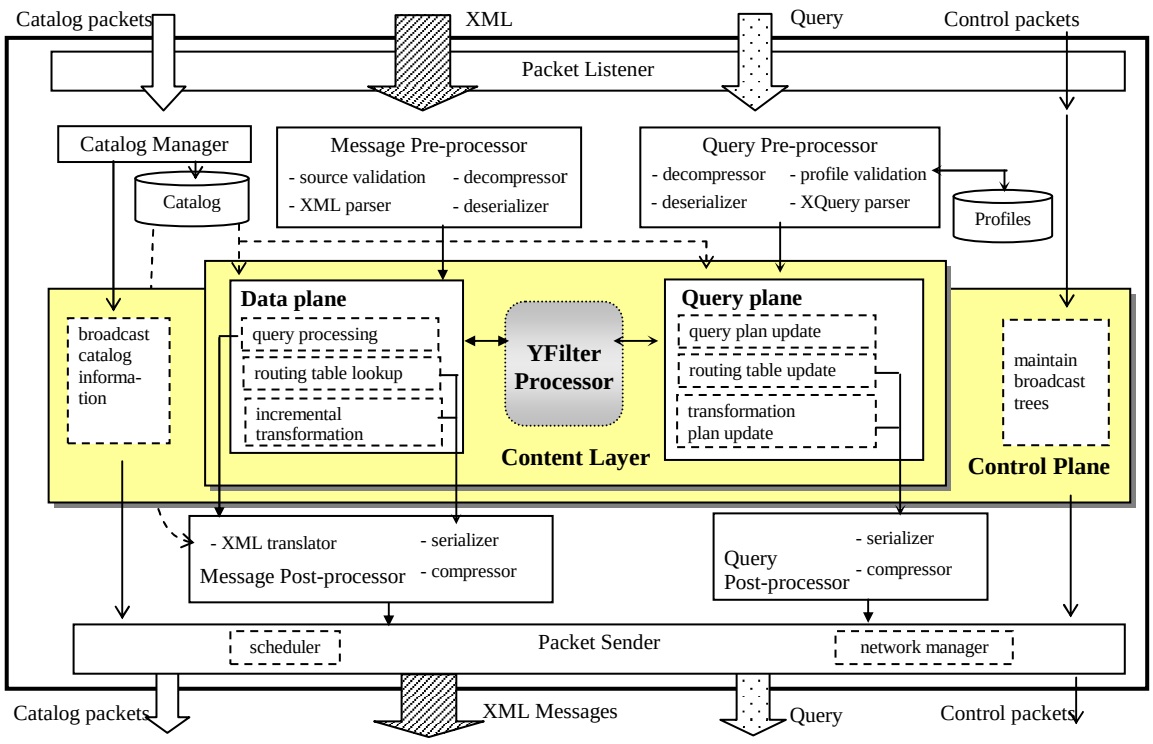


Figure 7.8: ONYX Broker Architecture

Message pre-processor. XML messages can come from data sources as well as other brokers in the system. The messages from a data source carry the source ID and are in the text format. On receiving such a message, the root broker of the data source validates the source ID attached to the message using its catalog. It also parses the message to an in-memory representation for later routing and query processing. If the message comes from an internal broker, source validation is skipped. Depending on the internal representation of XML, the message can be in one of several formats that were discussed earlier, and will need suitable pre-processing like decompression, deserialization, etc.

Query pre-processor. This is analogous to the message pre-processor in functionality, except that it also maintains a database of the profiles for which it is the host broker.

Control plane: Taking the control messages, the control plane maintains the broadcast tree for each root broker in the system. Specifically, it records the parent node and the child nodes of a broker on a particular root broker's broadcast tree. It provides two methods for use of the content layer, one for forwarding messages along a broadcast tree, the other for reverse forwarding of queries. The control plane is also responsible for disseminating catalog information for the purposes of optimizing content-based processing. For example, the schema information can be used to optimize query processing and support schema-aware XML encoding.

Data plane. The broker performs three tasks in the data plane, when receiving an XML message. First, it takes a sequence of steps to route the message: (a) if the broker is the root broker for the message, it attaches its broker identifier to the message; (b) it retrieves its output links in the broadcast tree that is specified by the root broker identifier attached to the message; and (c) it looks up in the content-based routing table to filter those output links. Second, for each output link selected, the broker transforms the message, if a transformation plan is attached to that link. Last, the broker processes the message on local queries to generate results. These three tasks are all realized by the YFilter processor.

Query plane. The query plane exhibits duality with the data plane. If an arriving query is from a user, the local query processing plan is updated. If the query comes from another broker to update the routing table (i.e., it is a routing query) or the incremental transformation plan (i.e., it is an incremental transformation query), the modification of the routing table or the transformation plan will cause a new query to be generated for delivery to its parent broker.

YFilter Processor. YFilter has been described in detail in the previous three chapters. In ONYX, it is leveraged to build a holistic processing plan for all the processing tasks, so that the shared processing among the tasks is maximized. For the query plane, it is extended to support the routing table construction operations (as described in Section 7.3.2). For the data plane, its scheduler is augmented to prioritize the processing for different types of queries while exploiting the sharing among them (see Section 7.4.1).

Message and query Post-processor. The results from the data plane are passed to the message post-processor. Results of local query processing are translated into XML messages for delivery to end users, while results of routing and incremental transformation are serialized (and possibly compressed). Queries generated from the query plane also follow the path of serialization and compression.

Packet Sender. This component attaches a header to each packet, specifying the type of flow, the identifier of the root broker (if the packet is an XML message), and the format used. Then it multiplexes the four types of flows into the output channel, through a scheduler and a network manager that sends packets through TCP, UDP, etc.

7.7 Related Work

As described in Section 3, distributed publish/subscribe systems provide many-to-many communication between publishers and subscribers using the complex predicate-based model. The proposed techniques and reported results particularly relevant to ONYX are described in more detail below. *Siena* [Carzaniga and Wolf, 2003] developed efficient forwarding algorithms for the routers to evaluate each packet against all output link queries and forward the packet along the links with matched queries. It also proposed the *CBCB* (*combined broadcast and content-based*) routing scheme [Carzaniga et al., 2004] that

contains a broadcast routing protocol at a lower layer and a content-based routing protocol at an upper level. *Gryphon* [Aguilera et al., 1999; Banavar et al., 1999] proposed to organize all subscriptions into a Parallel Search Tree and augment each copy of it with router-specific annotations for event forwarding. It also compared different routing schemes; the results of this study indicate that using IP multicast, multi-hop routing is advantageous over flooding only in cases of high selectivity of subscriptions or high locality (i.e., when subscription patterns vary by location); otherwise, nearly all subscription brokers will have some subscription matching each event [Opyrchal et al., 2000]. Mühl et al. compared a number of schemes to distribute subscription updates among a network of brokers. Their results show that merging subscriptions, utilizing publishers' description, and exploiting locality of subscriptions help to reduce the routing table sizes and the control traffic [Mühl et al., 2002]. ONYX has made a number of observations similar to the results mentioned above. To support rich XML structure and queries, ONYX also addresses many complex XML-related issues that arise in query processing, data forwarding, and routing table construction

The transformation functionality in ONYX is related to the transcoding of Web content to suit the profiles of heterogeneous end users, like the users of mobile phones and hand-held computers [IBM, 2004]. However, such profiles usually do not provide expressiveness in querying content as much as the subset of XQuery that ONYX supports. In addition, transcoding is usually performed at either the publisher sites or the edge brokers that end users directly contact. In contrast, ONYX can perform incremental message transformation at internal brokers to share processing work and reduce bandwidth consumption.

7.8 Summary

This chapter presented a design for ONYX, a distributed system providing large-scale XML dissemination services. A detailed architectural description of the system was provided, and a variety of issues including routing table construction, query population portioning, and efficient message processing were addressed in the context of leveraging the YFilter query processor. The fundamental ideas exploited in ONYX are centered on the use of declarative queries (1) to enable the dissemination of high-value content to end users/applications, (2) to bring intelligence to the networking routing fabric (i.e., supporting content-based routing), (3) to provide flexibility in choosing locations to place brokering functionality (e.g., incremental transformation of messages), and (4) to help improve overall performance of the system (e.g., exploiting locality of user interests and using content-based routing as opposed to broadcasting). These ideas have significant potential for research and commercial impact. In addition, the proposed architecture and the various techniques built on YFilter technology provide a basis for developing scalable XML dissemination services.

Many challenges lie ahead in order to achieve Internet-scale deployments of such XML dissemination services. Issues that remain to be addressed include adaptivity in routing table construction for load-balancing, routing table maintenance upon query updates, construction of network and query distribution models for performance analysis, to name just a few. These issues, and other future work in the area of XML message brokering, are the subject of the next chapter.

8 Future Work

Message-based information systems are a rich source of research issues. There are a number of interesting avenues for future work that can be built on the contributions of this dissertation. In the following, I discuss the opportunities for future work in both message-based and other emerging-types of distributed information systems.

New tasks for large-scale data dissemination. Research in the area of large-scale data dissemination can be extended by adding a number of services required by emerging applications.

Stateful publish/subscribe. Currently, query processing in ONYX is performed a single message at a time, with no interaction across message boundaries. A number of emerging applications such as network monitoring and market-trend analysis require information to be aggregated over a stream of messages. Adding support for such information aggregation in ONYX raises a significant research challenge, as the system would need to maintain *state* across messages on a per-query basis in addition to routing the messages to all relevant queries.

*Access to historical data***.** ONYX currently supports queries over current and future data. Many emerging applications require the ability to compare the current observation with observations in the past. For example, users may want to be notified if the traffic pattern this afternoon is similar to that at this time last month. Significant challenges arise in supporting access to historical data in distributed environments where distributed databases are used to store messages. Issues that remain to be addressed include: Where are databases placed? Which subset of the messages does each database store? When a query arrives, does it

initiate the creation of a new database or can some existing database be identified for answering it? Finally, how can queries be continuously evaluated?

*QoS-based publish/subscribe***.** Quality-of-Service (QoS) based publish/subscribe takes into account not only content-based but also time-based constraints. In such services, each user provides a query and a utility function specifying her degree of satisfaction for seeing relevant data delivered within a *time window*. Integrating time constraints into content-based routing raises the difficult problem of how to exploit shared processing and schedule such processing so that user data interests and time constraints can be met simultaneously.

A Comparative study. Another interesting task is a comparative study of the solutions to large-scale information dissemination. A number of dissemination systems have been independently developed in the database and networking communities, with the solutions varying from multicast to content-based routing. It will be of much benefit to devise network, data traffic, and query distribution models and to characterize system constraints so that these solutions can be fairly compared. Such a comparative study would facilitate the convergence of the network and information processing systems addressing the dissemination problem.

Complex event processing for sensor-based networks. Sensor devices such as wireless motes and RFID (*Radio Frequency Identification*) readers are gaining adoption in a growing number of applications for tracking and monitoring purposes. Large-scale deployment of these devices will soon generate an unprecedented volume of event messages. Such messages need to be filtered and combined to identify complex events, aggregated on different temporal and geographic scales, and transformed along concept hierarchies to create new events that reach a semantic level appropriate for end applications. During the transformation in a concept hierarchy, data processing history needs to be collected for end users to evaluate

data quality, which is crucial in areas such as environmental monitoring and disaster prediction.

These requirements represent a distinct query type that is similar to yet broader than the class of queries supported in this dissertation research. To support such queries for complex event processing, significant research challenges remain to be addressed. These challenges include a declarative query language for specifying the requirements, an infrastructure that links various devices to workstations and servers with high-speed connectivity, and algorithms that inject declarative queries into this infrastructure and efficiently execute them. The techniques developed in this dissertation research can be extended to address some of these challenges.

Messaging-based mobile services. Mobile applications constitute a particularly challenging distributed environment: the clients run a multitude of operating systems and can be located anywhere. Information exchange between servers and a huge, dynamic collection of heterogeneous clients has to rely on open, XML-based technologies. In particular, multi-step filtering, incremental transformation, and mutual filtering between servers and clients are all potential areas in which XML query processing can play an important role.

In summary, emerging distributed information systems and applications provide a myriad of interesting and challenging research topics related to large-scale query processing. The query processing techniques developed in this dissertation must be further broadened and enriched to rise to the challenges of these new environments.

9 Concluding Remarks

The Information Technology industry is moving towards building large-scale, flexible, and high-function distributed information systems. XML message brokering, which integrates XML with the publish/subscribe interaction model and exploits declarative XML queries, has emerged as an infrastructure that is able to meet these requirements. In this dissertation, I presented YFilter/ONYX, an XML message brokering system that provides three key components to this new infrastructure:

- A *filtering* engine that matches incoming messages against large numbers of path expressions. By using an NFA-based approach for shared processing, YFilter provides order-of-magnitude benefits over prior work while supporting a wide variety of XML document types and query workloads. The filtering engine has been released and is being used by a growing community of users.
- A *transformation* module that restructures matching messages according to query-specific requirements, resulting in customized result delivery. Built on the YFilter filtering engine, this module uses sophisticated techniques to further share processing among transformation queries. This algorithm is the first in the literature that can efficiently handle tens of thousands of simultaneous queries.
- The ONYX system spreads filtering and transformation functionality into a distributed data dissemination network and augments the network with routing capabilities. In particular, it pushes declarative queries into the network to perform content-based routing and incremental message processing during routing. These routing extensions are built on YFilter technology. The notion of in-network query processing and its efficient

implementation using YFilter enable ONYX to achieve scalability in addition to functionality.

In retrospect, I have exploited three key ideas that have proven beneficial in this dissertation and that can be valuable to other work in the future. First, the shared processing of queries is key to performance and scalability. In this research, novel techniques have been devised to identify and exploit commonalities among filtering and transformation queries. Second, despite the differences in execution models, XML query processing can leverage relational processing for simplicity and performance. Such leveraging requires effective mappings between the two models; a successful example has been the creation of pathtuple streams that this research uses to convert event-based XML processing to tuple-based relational processing, which results in effective optimizations for XML transformation. Third, in large-scale distributed environments for data dissemination, queries bring intelligence to the network routing fabric, provide flexibility in placing brokering functionality, and enable improvements in overall system performance.

As XML becomes the dominant protocol for connecting disparate systems and XML-aware application-oriented networking gains momentum, techniques such as those developed in this dissertation will be of increasing commercial importance. It is my hope that the ideas and techniques developed in YFilter/ONYX can serve as a starting point for many more efforts that will eventually lead to the realization of large-scale, high-performance, and high-function data exchange and information dissemination.

Appendix A: Description of the XFilter Approach

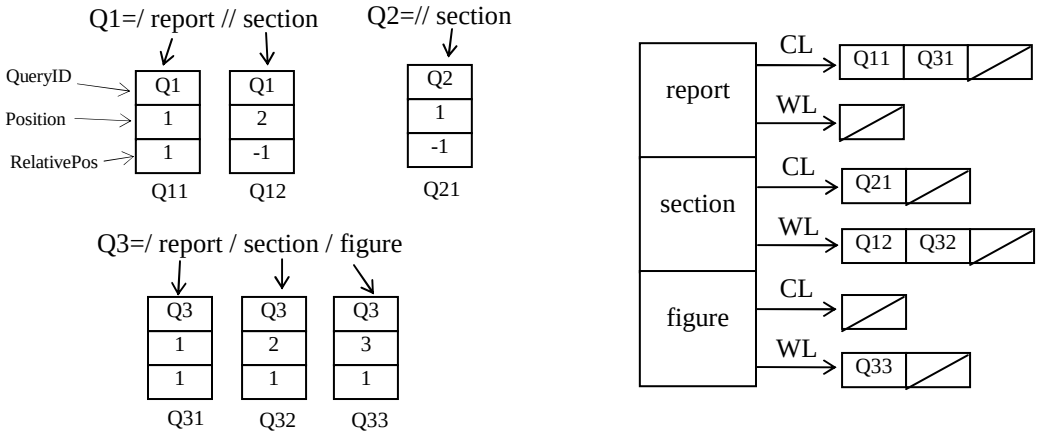
XFilter was the first published XML filtering system that supports efficient structure matching for a large set of path queries. As described in Section 2.4, XFilter creates a Finite State Machine (FSM) for each path query and uses event-based parsing to drive the execution of the query FSMs. This appendix provides more details on XFilter including its query index and execution algorithm.

XFilter builds a dynamic index over the states of query FSMs. To do so, it implements the states of a FSM as *path nodes*. These path nodes represent the element nodes in the query, except wildcard ("*") nodes. A path node contains the following information:

- **QueryId**: A unique identifier for the path expression to which this path node belongs.
- **Position**: A sequence number that determines the location of this path node in the order of the path nodes for the query. The first node of the path is given position 1, and the following nodes are numbered sequentially.
- **RelativePos**: An integer that describes the distance in *document levels* between this path node and the previous path node. This value is marked using "-1" if a path node contains a descendant ("//") axis. Otherwise, it is set to 1 plus the number of wildcard nodes between it and its predecessor node (assuming that the first path node of a query has a pseudo-predecessor node).

Figure A.1(a) shows this representation for three path expressions.

The query index of XFilter is organized as a hash table based on the element names that appear in the path expressions. Associated with each unique element name are two lists: the *candidate list* and *wait list.* Candidate lists identify the path nodes corresponding to the states



(a) path nodes of three queries

(b) query index

Figure A.1: Path Nodes of Queries and a Query Index in XFilter

that the FSM execution is attempting to match at a particular moment. Wait lists contain path nodes that are subsequent to the nodes in the candidate lists. The contents of these lists constantly change as parsing events drive the execution of the FSMs.

The initial distribution of path nodes to candidate lists is an important contributor to performance. Figure A.1(b) shows the most straightforward case, where the path nodes for the initial states in Figure A.1(a) are placed on the candidate lists. For many situations, however, such an approach can be inefficient. The reason is that the first path nodes of queries address elements at higher levels in the documents where the sets of possible element names are small, thus providing poor selectivity (i.e., inadequate to reduce the number of queries that must be considered further). Based on this, XFilter developed the *List Balance* method that attempts to pick more selective path nodes and place them initially in the candidate lists. This method was shown to provide better performance overall than the basic indexing method.

To match the path nodes contained in the Query Index, XFilter uses a *level* value for each path node, which represents the level in the XML document at which this path node should be checked. Because XML does not restrict element types from appearing at multiple levels of a document, it is not always possible to assign this value during query parsing. Rather, this information needs to be updated during the evaluation of the query. XFilter's query execution algorithm is implemented using the following two callback functions.

Start Element Handler: The handler looks up the element name in the Query Index and examines all the nodes in the candidate list for that entry. For each node, it performs a *level check*. The purpose of the level check is to make sure that the element appears in the document at a level that matches the level expected by the query. If the path node does not contain a "//" axis (i.e., its RelativePos value is non-negative), the two levels must be identical in order for the check to succeed. Otherwise, the level for the node is unrestricted, so the check succeeds regardless of the element level.

If the check succeeds, then the node passes and the query is moved into its next state (if it has not been entirely matched). This is done by *copying* the next node for the query from its wait list to its candidate list (note that a copy of the promoted node remains in the wait list). If the *RelativePos* value of the copied node is not -1, its level value is also updated using the current level and its *RelativePos* value to do future level checks correctly.

Note that in the most basic case, there is only one copy of a path node in its candidate list during the evaluation of a query. However, when the same element name appears in a nested manner at different levels of the input document and a path node related to this element name corresponds to a "//" location step, matching of the nested elements with this path node will cause multiple promotions of its subsequent path node. In such cases, multiple copies of the

subsequent path node can exist in its corresponding candidate list to reflect different document levels where it can be matched.

End Element Handler: When an end element tag is encountered, the path nodes promoted when the corresponding “start element” tag was encountered are deleted from the candidate lists in order to restore those lists to the state before reading this element. This “backtracking” is necessary to handle the case where multiple elements with the same name appear at different levels in the document

Appendix B: Description of the Hybrid Approach

The hybrid approach was used in Section 4.4 as a middle point between XFilter and YFilter with respect to the amount of sharing exploited. It supports shared processing of query fragments containing only child (‘/’) axes.

Hybrid works as follows. First, each query is decomposed into substrings containing only “/” operators (i.e., it is split at “*” and “//” operators); a list of nodes is created for the query with one node per substring. Each node contains four data items (QueryId, NodePosition, RelativePos, Level), as path nodes in XFilter. The difference is that RelativePos here specifies the distance in document levels from the end of the previous substring to the end of this substring. Then, the substrings of all of the queries are inserted into a single *Trie* index. Inside the index, a candidate list is allocated in each index node that represents the end (i.e., the last element) of a substring. Similar to XFilter, a candidate list here contains nodes representing those substrings that the current execution attempts to match. Initially, candidate lists only contain the nodes for the first substrings of queries.

During the execution, input elements drive the navigation in the *trie* index as in YFilter, but without any concern for “*” and “//” operators. Each input element initiates a search from the root of the *trie* and also continues searches from index nodes that the navigation reached on the previous input element. As in YFilter, a runtime stack is used for maintaining the list of index nodes representing the current state and for backtracking. When an index node with a non-empty candidate list is encountered, all substring nodes in the list undergo a document level check. For each of those substrings that pass the level check, the expected level of the end of the next substring in the query is updated in the node for the next

substring, and that substring node is copied to its corresponding candidate list. In this way, the matching of a substring in the *trie* index is shared by all queries containing this substring, but the transitions between two substrings are done on a query-by-query basis using document level checking as in XFilter.

Appendix C: Data Structures and Pseudo-code for Inline

In this appendix, I provide the data structures and the pseudo-code used in the Inline approach to incorporating value-based predicate evaluation in shared structure matching. The Inline approach was described in Section 5.1.1.

A.1 Data Structures for Bookkeeping

```
QueryEvaluation[ ]    queryEvalList;
class QueryEvaluation {
    boolean  isMatched;
    PredicateEvaluation[ ]    predEvalList;
}
class PredicateEvaluation {
    int        stepNumber;
    Set        elementIdentifiers;
}
```

A.2 Pseudo-code

```
QueryEvaluation[ ] queryEvalList;
Stack elementIDStack, truePredicateStatck;
Start document handler:
  if queryEvalList has not been allocated
     allocate queryEvalList;
  else
     clear all data structures in queryEvalList;
Start element handler:
  assign an element identifier elementID to this element Element;
```

for each active state

apply rule (1) to (4) to find target states (see section 4.3.4);

endfor

List *truePredicates*;

for each target state

(1) for each predicate P in the local predicate table of the state

retrieve the *P.QueryID*th element *queryEval* from *queryEvalList*;

evaluate *P* using *Element* only if *queryEval.isMatched* is false;

if *P* is evaluated to true

retrieve the *P.PredicateID*th element *predEval* from *queryEval*;

add *elementID* to the set *elementIdentifiers* in *predEval*;

add the pair (*P.QueryID*, *P.PredicateID*) to *truePredicates*;

endif

endfor

if this target state is an accepting state

(2) for each query Q whose identifier is in the ID list at the state

if all predicates contained in *Q* have been satisfied

intersect element identifier sets of all predicates that have the same step number;

if the intersection is non-empty for every level

queryEval.isMatched = true;

endif

endif

endfor

endif

endfor

push *elementID* to *elementIDStack*;

push *truePredicates* to *truePredicateStack*;

End element handler:

pop the top element *truePredicates* from *truePredicatStack*;

pop the top element *elementID* from *elementIDStack*;

for each pair (*P.QueryID*, *P.PredicateID*) of predicate *P* in the list *truePredicates*

(3) retrieve the P.QueryIDth element queryEval from queryEvalList;

if *queryEval.isMatched* is false

retrieve the *P.PredicateID*th element *predEval* from *queryEval*;

remove *elementID* from the set *elementIdentifiers* in *predEval*;

endif

endfor

Note: (1) evaluation of a predicate; (2) final evaluation of a query; (3) undo for a predicate

Appendix D: Data Structures and Pseudo-Code for SP

This appendix provides the data structures and pseudo-code used in the *Selection Postponed* (SP) approach to integrated structure and value-based processing. The SP approach was discussed in Section 5.1.2.

B.1 Data Structures for Bookkeeping

Boolean[ ] *queryEvalList*;

B.2 Pseudo-code

Boolean[ ] *queryEvalList*;

Start document handler:

if *queryEvalList* has not been allocated

allocate *queryEvalList*;

else

clear all data structures in *queryEvalList*;

endif

Start element handler:

assign an element identifier *elementID* to this element *Element*;

for each active state

apply rule (1) to (4) to find target states (see section 4.3.4);

retrieve sequences of elements for the active state by following pointers from the state in the run time stack;

append *Element* to the end of the sequences to obtain new sequences for all target states;

for each target state that is an accepting state

for each sequence of elements

for each query *Q* whose identifier is in the ID list at the state

(1) retrieve the *Q.QueryID*th element of *queryEvalList*;

if *Q* is not matched

for each predicate *P* of *Q*

retrieve an element from the sequence using *P*'s step number and evaluate *P*;

if evaluation fails

break;

endif

endfor

if all pedicates are satisfied

set the *Q.QueryID*th element of *queryEvalList* to true;

endif

endif

endfor

endfor

endfor

Note: (1) selection performed by SP.

Appendix E: Proof of Claims

This appendix provides rigorous proof for the claims presented in Section 6.5.1 that can be used to simplify post-processing plans created for XML transformation queries. Consider a path expression p of m location steps, and the **stream** of tuples that match the path, with fields numbered 1..*m.*

Proposition 4.1: If duplicates occur in field m of tuples in the stream, then in the path expression p which contains m location steps, there exist i and j, $i < j \leq m$, such that both location steps i and j contain a "//" axis and the element in location step i is on a loop in the DTD element graph.

Proof: Let t_1 and t_2 denote the two tuples containing the duplicates, and let n^* be the common node identifier in their last field m, as illustrated in Figure E.1. Since t_1 and t_2 are distinct tuples, there must exist at least one field where the two tuples differ. Let field i be the first field where they differ, $1 \leq i < m$. The shaded fields before field i in Figure E.1 represent the same content in both tuples. The node identifiers in field i in two tuples are denoted as n_{i1} and n_{i2}. Since they are in the same field, I know that n_{i1} and n_{i2} match the same element, i.e. the element in location step i of the path. Also, because $i < m$, n_{i1} and n_{i2} matching field i are both ancestors of n^* matching field m in the document tree. Since all ancestors of n^* lie on a single document path, one of n_{i1} and n_{i2} must contain the other. This shows the element in location step i must be on a loop in the DTD graph.

I then claim the axis in location step i is "//". To see why, let us assume the axis is '/' instead. Then a node in field i must be a child of the node in field i-1. Since I know n_{i1} and n_{i2} contain each other, they must have different parent nodes in field i-1, which conflicts my

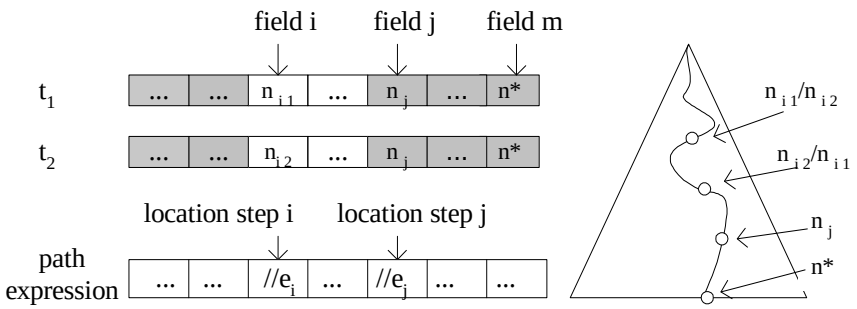


Figure E.1: Two duplicate tuples, their path expression and a document tree

choice of field *i* (recall field *i* is the first field where the two tuples differ). A special case is that field *i* is the first field. If the axis in the first location step is '/', then n_{i1} and n_{i2} both have to be the root node, which also conflicts my choice of field *i*.

In addition, let field *j* be the field immediately after the last field where the two tuples differ, $j <= m$. The shaded fields from field *j* to the end in Figure E.1 represent the same content in both tuples. I claim the axis in location step *j* is also "//". Again assume the axis is '/' instead. Then a node in field *j* must be a child of the node in field *j-1*. Since two tuples have the same node n_j in field *j*, they must have the same parent node of n_j in field *j-1*, which conflicts my choice of field *j*.

Claims 1 and 2 follow immediately from this proposition.

Proof of Claim 3: If claim 1 or 2 claims that there are no duplicates for p_1, I know two matches of p_1 can not have the same node identifiers for the last location step of p_1. In addition, if claim 1 or 2 also claims that are no duplicates for p_2, then I know for any node that matches the last location step of p_1, in the sub-tree rooted at this node, there won't be matches of p_2 that have the same node identifier for p_2's last location step. Combining the above two facts shows that there will no duplicates in the tuples matching the original path when they are projected onto field *i* and *m*.

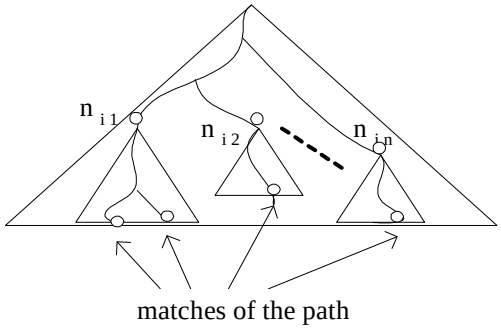


Figure E.2: A document tree with no recursive nodes in field i.

Proposition 4.2: If tuples in the stream matching *p* are not in increasing order when projected onto field *i*, $1 \leq i < m$, then

(a) there exist two tuples whose node identifiers in field *i* identify two nodes that contain each other;

(b) there exists a field *j*, $j \leq i$, s.t.

- the axis of location step *j* in the path expression is "//";
- the element of location step *j* is on a loop in the DTD element graph; and
- the element in location step *i* is on the same loop.

Proof. (a) Let us look at the nodes identified by field *i* of the tuples. Assume there do not exist two nodes that contain each other. Then the sub-trees rooted at these nodes are disjoint, as illustrated by nodes n_{ij} ($j \geq 1$) in Figure E.2.

Also, location steps *i* to *m* are evaluated in these disjoint subtrees. Since the tuples are in increasing order in field *m*, I know that matches of the path expression are returned from one disjoint subtree to another in their document order. This implies that nodes for field *i* are also returned in increasing document order, which conflicts the previous assumption that tuples are not in increasing order in field *i*.

(b) Based on the existence of two tuples whose node identifiers in field i identify two nodes that contain each other, I move to investigate location step i in the path expression which is matched by these nodes. I already know that the element in this location step is on a loop in the DTD graph (otherwise the nodes matching the location step cannot contain each other.) What to be checked is the axis of the location step.

Case 1: Location step i contains a "//" axis. Then this is a special case of claim (b) with $j = i$.

Case 2: Location step i contains a "/" axis. Let t_1 and t_2 denote the two tuples, and n_{i1} and n_{i2} denote the nodes in their field i. Without loss of generality, assume n_{i1} contains n_{i2}. Since the axis of location step i is "/", the parent node of n_{i1} in t_1 must also contain that of n_{i2} in t_2. See Figure E.3 (a) for the illustration. Let field j be the last field before field i whose location step contains a "//" axis. Using the same argument, I know node $n_{j,1}$ in field j in t_1 also contain the node $n_{j,2}$ in field j in t_2. This implies the element in location step j is an element on a loop. Note that such a j location step containing "//" must exit, otherwise by using induction, I reveal conflicting facts that the first location step of the path contains a '/' axis but the two nodes in the first field of the two tuples contain each other.

Last let us check why the elements in location step j and location step i share one DTD element loop. Let e_j and e_i be the two elements in the two location steps. Assume that e_j and e_i are not on any common loop. Given the additional facts that (1) there is a path from e_j to e_i in the document tree (which is in the content of field j to i of either tuple) and (2) both e_j and e_i are elements on loops, I know that in the DTD graph, they are on different loops with one or more directed bridges going from the loop containing e_j to that containing e_i.

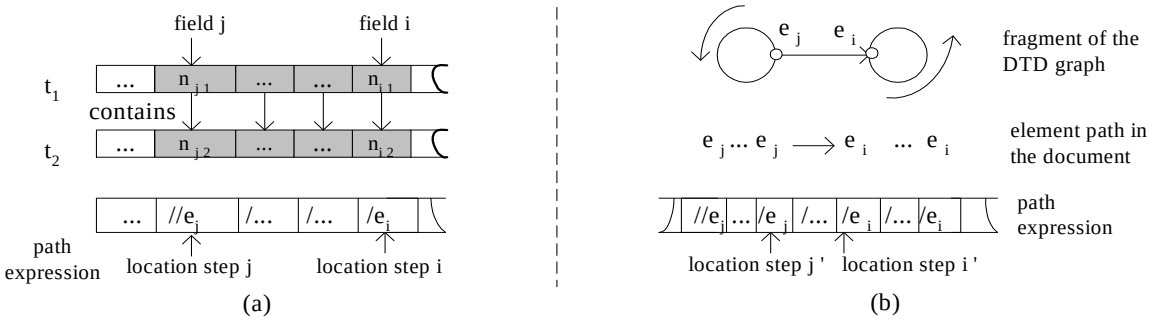


Figure E.3: Two tuples with recursive nodes in field i, their path expression, related DTD graph, and the element path in the document

Assume location step *j'* and location step *i'* are such location steps in the path expression that $j \leq j' < i' \leq i$ and any location step between *j'* and *i'* do not contain element e_j or e_i. Then elements in location steps *j'* to *i'* define a unique bridge between the two loops in the DTD graph. Figure E.3 (b) illustrates the bridge determined by these location steps.

Then it is important to note that, any path from e_j to e_i through this bridge in a document tree must contain one and only one path fragment corresponding to crossing the bridge (note that it goes one direction). Since the content of field *j'* to field *i'* in t_1 matches location steps *j'* to *i'*, this content must be identical to that path fragment. Also, as the content of field *j'* to field *i'* in t_2 matches location steps *j'* to *i'* on the same document path as t_1, it also must correspond to the unique path fragment. This conflicts with the existing knowledge that the nodes in any field between *j* and *i* in the two tuples contain each other. So the assumption that e_j and e_i are not on any common loop is false.

Claims 4 and 5 follow immediately from this proposition.

Bibliography

[Abadi et al., 2003] Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. Aurora: A New Model and Architecture for Data Stream Management. In *VLDB Journal*, 12(2), 120-139, August 2003.

[Aguilera et al., 1999] Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., and Chandra, T.D. Matching Events in a Content-Based Subscription System. In *Proc. of Principles of Distributed Computing* (*PODC'99*), Atlanta, GA, May 1999.

[Alonso et al., 2004] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. Web Services: Concepts, Architectures and Applications. Springer Verlag, Heidelberg, Germany, 2004.

[Altinel and Franklin, 2000] Altinel, M., and Franklin, M.J. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of the 26th Int'l Conference on Very Large Data Bases* (*VLDB'00*), 53-64, Cairo, Egypt, September 2000.

[Altinel et al., 1999] Altinel, M., Aksoy, D., Baby, T., Franklin, J.M., Shapiro, W., and Zdonik, S.B. DBIS-Toolkit: Adapatable Middleware for Large Scale Data Delivery. In *Proc. of the 1999 ACM SIGMOD International Conference on Management of Data*, 544-546, Philadelphia, PA, 1999.

[Apache XML, 1999] Apache XML project. Xerces Java parser 1.2.3 Release. http://xml.apache.org/xerces-j/index.html, 1999.

[Apache Hermes, 2004] Apache WebServices – Hermes. http://incubator.apache.org/hermes/, 2004.

[Arasu et al., 2003] Arasu, A., Babu, S., and Widom, J. CQL: A Language for Continuous Queries over Streams and Relations. In *Proc. of the 9th International Workshop on Database Programming Languages* (*DBPL'03*), 1-19, Potsdam, Germany, September, 2003.

[Ariba., 2005] Ariba Inc. Spend Management Solutions. http://www.ariba.com/, 2005.

[Ballardie et al., 1993] Ballardie, T., Francis, P., and Crowcroft, J. An Architecture for Scalable Inter-Domain Multicast Routing. In *Proc. of the 1993 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications* (*SIGCOMM'93*), 85-95, San Francisco, CA, September 1993.

[Banavar et al., 1999] Banavar, G., Chandra, T. D., Mukherjee, B., Nagarajarao, J., Strom, R. E., and Sturman, D. C. An Efficient Multicast Protocol for Content-Based Publish-

Subscribe Systems. In *Proc. of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 262-272, Austin, TX, May 1999.

[BEA Systems, 2002] BEA Systems, Inc. BEA WebLogic Integration: Application Integration. http://bea.com/products/weblogica/server/index.shtml, 2002.

[BEA Systems, 2005] BEA Systems, Inc. 2005. http://www.bea.com, 2005.

[Belkin and Croft 1992] Belkin, N.J., and Croft, B.W. 1992. Information filtering and information retrieval: Two sides of the same coin? *Communications of the ACM*, 35(12), 29-38.

[Bell et al., 1990] Bell, T.C., Cleary, J.G., and Witten, I.H. *Text Compression*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Bhide et al., 2002] Bhide, M., Deolasse, P., Katker, A., Panchgupte, A., Ramamritham, K., and Shenoy, P. Adaptive Push Pull: Disseminating Dynamic Web Data. *IEEE Transactions on Computers*, 51(6), 652-668, May 2002.

[Birrell and Nelson, 1984] Birrell A.D., and Nelson, B.J. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), 39-59, February 1984.

[Boag et al., 2003] Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., and Siméon, J. XQuery 1.0: An XML Query Language. W3C Working Draft, November 2003. http://www.w3.org/TR/xquery/.

[Bosworth, 2002] Bosworth, A. Data Routing Rather than Databases: the Meaning of the Next Wave of the Web Revolution to Data Management. In *Proc. of the 28th International Conference on Very Large Data Bases* (*VLDB'02*), Hong Kong, China, August 2002.

[Bray et al., 2004] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., and Yergeau, F. Extensible Markup Language (XML) 1.0. W3C recommendation. http://www.w3.org/TR/2004/REC-xml-20040204/, February 2004.

[Bruno et al., 2002] Bruno, N., Koudas, N., and Srivastava, D. Holistic twig joins: optimal XML pattern matching. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data* (*SIGMOD'02*), 310-321, Madison, WI, June 2002.

[Bruno et al., 2003] Bruno, N., Gravano, L., Doudas, N., and Srivastava, D.. Navigation- vs. Index-based XML Multi-query processing. In *Proc. of the 19th International Conference on Data Engineering* (*ICDE'03*), 139-150, Bangalore, India, March 2003.

[Busse et al., 2001] Busse, R., Carey, M., Florescu, D., Kersten, M., Manolescu, I., Schmidt, A., and Waas, F. Xmark: An XML benchmark project. http://monetdb.cwi.nl/xml/index.html, 2001.

[Carges, 2005] Carges, M. Taking SOA from "Pilot to Production" with Service Infrastructure. InfoWorld SOA Executive Forum, keynote presentation. http://www.infoworld.com/event/soa/InfoWorld_SOA_Mark_Carges.ppt, 2005

[Carriero and Gelernter, 1989] Carriero, N., and Gelernter, D. Linda in Context. *Communications of the ACM*, 32(4), 444-458, 1989.

[Carzaniga et al., 2004] Carzaniga, A., Rutherford, M.J., and Wolf, A.L. A Routing Scheme for Content-Based Networking. In *Proc. of IEEE INFOCOM 2004*, Hong Kong, China, March 2004.

[Carzaniga and Wolf, 2003] Carzaniga, A., and Wolf, A.L. Forwarding in a Content-Based Network. In *Proc. of the 2003 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications* (*SIGCOMM'03*), 163-174, Karlsruhe, Germany, August 2003.

[Cetintemel et al., 2000] Cetintemel, U., Franklin, M.J., and Giles, C.L. Self-adaptive user profiles for large scale data delivery. In *Proc. of the 16th International Conference on Data Engineering* (*ICDE 2000*), Los Alamitos, CA, USA, 622-633, 2000.

[Chamberlin et al., 2003] Chamberlin, D., Fankhauser, P., Florescu, D., Marchiori, M., and Robie J. XML Query Use Cases. W3C Working Draft. http://www.w3.org/TR/xmlquery-use-cases/, November 2003.

[Chan et al., 2002] Chan, C., Felber, P., Garofalakis, M., and Rastogi, R. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of the 18th Int'l Conference on Data Engineering* (*ICDE'02*), 235-244, San Jose, CA, February 2002.

[Chan et al., 2002(2)] Chan, C.Y., Fan, W., Felber, P., Garofalakis, M.N., and Rastogi, R. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proc. of the 28th International Conference on Very Large Data Bases* (*VLDB'02*), Hong Kong, China, August 2002.

[Chand et al., 2003] Chand, R., and Felber, P. A Scalable Protocol for Content-Based Routing in Overlay Networks. In *Proc. of the IEEE* International *Symposium on Network Computing and Applications* (*NCA'03*), Cambridge, MA, April 2003.

[Chandrasekaran et al, 2003] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M.A. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the 1st Biennial Conference on Innovative Data Systems Research* (*CIDR'03*), Asilomar, CA, January 2003.

[Chen et al., 2002] Chen, J., DeWitt, D.J., Naughton, J.F. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proc. of*

the 18th Int'l Conference on Data Engineering (*ICDE'02*), 345-354, San Jose, CA, February 2002.

[Chen et al., 2000] Chen, J., Dewitt, D.J., Tian, F., and Wang, Y. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of the 2000 ACM SIGMOD Int'l Conference on Management of Data*, 379-390, Dallas, Texas, May, 2000.

[Chesnais et al., 1995] Chesnais, P.R., Muchlo, M.J., and Sheena, J.A. the Fishwrap Personalized News System. In *Proceedings of the IEEE 2nd International Workshop on Cummunity Networking Integrating Multimedia Services to the Home*, Princeton, NJ.

[Chu et al., 2000] Chu, Y. Rao, S.G., and Zhang, H. A Case for End System Multicast. In *Proc. of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1-12, Santa Clara, CA, June 2000.

[Cisco Systems, 2002] Cisco Systems, Inc. Internet Protocol (IP) Multicast. http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ipmulti.htm, 2002.

[Cisco Systems, 2005] Cisco Systems, Inc. Application-Oriented Networking. http://www.cisco.com/en/US/products/ps6455/index.html, 2005.

[Clark and DeRose, 1999] Clark, J., and DeRose, S. XML Path Language (XPath) - Version 1.0. http://www.w3.org/TR/xpath, November, 1999.

[COR Financial Solutions, 2004] COR Financial Solutions Ltd. Salerio e2e middleware. http://www.corfinancialsolutions.com/salerio.htm, 2004.

[Cowan and Tobin, 2004] Cowan, J. and Tobin, R. XML Information Set (Second Edition). W3C Recommendation 4 February 2004. http://www.w3.org/TR/xml-infoset/, 2004.

[DataPower Technology, 2005] DataPower Technology, Inc. http://www.datapower.com/, 2005.

[Diao et al., 2002] Diao, Y., Fischer, P.M, Franklin, M.J., and To, R. YFilter: Efficient and Scalable Filtering of XML Documents. In Proceedings of the 18th International Conference on Data Engineering, 341-342, San Jose, CA, February, 2002.

[Diao et al., 2003] Diao, Y., Altinel, M., Zhang, H., Franklin, M.J., and Fischer, P.M. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Transactions on Database Systems* (*TODS*), 28(4), 467-516, December 2003.

[Diao and Franklin, 2003] Diao, Y., and Franklin, M.J. Query Processing for High-Volume XML Message Brokering. In *Proc. of the 29th Int'l Conference on Very Large Data Bases* (*VLDB'03*), 261-272, Berlin, Germany, September 2003.

[Diao et al., 2004] Diao, Y., Rizvi, S., and Franklin, M.J. Towards an Internet-Scale XML Dissemination Service. In *Proc. of the 30th Int'l Conference on Very Large Data Bases* (*VLDB'04*), 612-623, Toronto, Canada, August 2004.

[Diaz and Lovell, 1999] Diaz, A.L., and Lovell, D. XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator, September 1999.

[Dilley et al., 2002] Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., and Weihl, B. Globally Distributed Content Delivery. *IEEE Internet* Computing, 50-58, September-October, 2002.

[Fabret et al., 2001] Fabret, F., Jacobsen, H.A., Llirbat, F., Pereira, J., Ross, K.A., and Shasha, D. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *Proc. of the 2001 ACM SIGMOD International Conference on Management of Data*, 115-126, Santa Barbara, May 2001.

[Florescu et al., 2004] Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M.J., and Sundararajan, A. The BEA Streaming XQuery Processor. In *VLDB Journal*, 13(3), 294-315, 2004.

[Florescu and Kossmann, 2004] Florescu, D. and Kossmann, D. XML Query Processing. Tutorial for ICDE 2004. http://www.dbis.ethz.ch/research/publications/50.ppt, 2004.

[Foltz and Dumais, 1992]Foltz, P.W., and Dumais, S.T. Personalized Information Delivery: An Analysis of Information Filtering Methods. *Communications of the ACM*, 35(12), 51-60, 1992.

[Ganglia, 2005] The Ganglia System. http://ganglia.info/, 2005.

[Gelernter and Carriero, 1992] Gelernter, D., and Carriero, N. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2), 96-107, 1992.

[Goldman and Widom, 1997] Goldman, R., and Widom, J. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of the 23rd International Conference on Very Large Data Bases* (*VLDB 1997*), 436-445, Athens, Greece, August, 1997.

[Gnome, 2001] LIBXML: the XML C parser and toolkit of Gnome. http://xmlsoft.org/, 2001.

[Green et al., 2003] Green, T. J., Miklau, G., Onizuka, M., Suciu, D. Processing XML Streams with Deterministic Automata. In *Proc. of Int'l Conference on Database Theory* (*ICDT'03*), 173-189, Siena, Italy, January 2003.

[Green et al., 2004] Green, T. J., Gupta, A., Miklau, G., Onizuka, M., Suciu, D. Processing XML Streams with Deterministic Automata and Stream Indexes. In *ACM Transactions on Databases* (*TODS*), 29(4), December, 2004.

[GridICE, 2005] The GridICE project. http://infnforge.cnaf.infn.it/gridice/index.html, 2005.

[Gryphon, 2002] The Gryphon project. http://www.research.ibm.com/gryphon/gryphon.html, 2002.

[Gupta and Suciu, 2003] Gupta, A. K., and Suciu, D. Streaming processing of XPath queries with predicates. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, 419-430, San Diego, CA, June 2003.

[Halverson et al., 2003] Halverson, A., Burger, J., Galanis, L., Krishnamurthy, R., Rao, A.N., Tian, F., Viglas, S., Wang, Y., Naughton, J.F., and DeWitt, D.J. In *Proceedings of the 27th International Conference on Very Large Data* Bases (*VLDB'03*), 225-236, Berlin, Germany, September, 2003.

[Hanson et al., 1999] Hanson, E.N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parthasarathy, S, Park, J.B., and Vernon, A. Scalable Trigger Processing. In *Proc. of the 15th Int'l Conference on Data Engineering* (*ICDE'99*), 266-275, Sydney, Australia, March 1999.

[Hopcroft and Ullman, 1979] Hopcroft, J. E., AND Ullman, J. D. *Introduction to Automata Theory, Languages and Computation*. Addition-Wesley Pub. Co., Boston, MA, 1979.

[IBM, 2000] International Business Machines. TSpaces: Intelligent Connectionware. http://www.almaden.ibm.com/cs/TSpaces/, 2000.

[IBM, 2002] International Business Machines. WebSphere MQ Series. http://www-306.ibm.com/software/integration/wmq/, 2002.

[IBM, 2005] International Business Machines. WebSphere Application Server Network Deployment. http://www-306.ibm.com/software/webservers/appserv/was/network/, 2005.

[IBM, 2004] International Business Machines. WebSphere Transcoding Publisher. http://www-306.ibm.com/software/pervasive/transcoding_publisher/, 2004.

[IPTC, 2004] Internal Press Telecommunications Council. News Industry Text Format. http://www.nitf.org/, 2004.

[Ives et al., 2002] Ives, Z.G., Halevy, A.Y., and Weld, D.S. An XML Query Engine for Network-Bound Data. In the *VLDB Journal*, 11(4), 380-402, December 2002.

[Jagadish et al., 2002] Jadadish, H.V., Al-Khalifa, S., Chapman, A., Lakshmanan, L.V.S., Nierman, A., Paparizos, S., Patel, J.M., Srivastava, D., Wiwatwattana, N., Wu, Yuqing, and Yu, Cong. TIMBER: A Native XML Database. The *VLDB Journal*, 11(4), 274-291, 2002.

[Jannotti et al., 2000] Jannotti, J., Gifford, D.K., Johnson, K.L., Kaashoek, M.F., and O'Toole, J.W.Jr. Overcast: Reliable Multicasting with an Overlay Network. In *Proc. of the 4th Symposium on Operating System Design and Implementation (OSDI'00)*, San Diego, CA, October 2000.

[Jiang et al., 2003] Jiang, H., Wang, W, Lu, H., and Yu, J.X. Holistic Twig Joins on Indexed XML Documents. In *Proc. of the 29th Int'l Conference on Very Large Data Bases* (*VLDB'03*), 273-284, Berlin, Germany, September 2003.

[Kay, 2001] Kay, M. Saxon: the XSLT processor. http://users.iclway.co.uk/mhkay/saxon/, 2001.

[Le Hors et al., 2004] Le Hors, A., Le Hégaret, P., Wood, Lauren, Nicol, G., Robie, J., Champion, M., and Byrne, S. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation 7 April, 2004. http://www.w3.org/TR/DOM-Level-3-Core/, 2004.

[Lakshmanan and Sailaja, 2002] Lakshmanan, L.V.S., and Sailaja, P. On Efficient Matching of Streaming XML Documents and Queries. In *Proc. of the 8th Int'l Conference on Extending Database Technology* (*EDBT'02*), 142-160, Prague, Czech Republic, March 2002.

[Ley, 2001] Ley, M. DBLP DTD. http://www.acm.org/sigmod/dblp/db/about/dblp.dtd, 2001.

[Liefke and Suciu, 2000] Liefke, H., and Suciu, D. XMILL: An Efficient Compressor for XML Data. In *Proc. of the 2000 ACM SIGMOD Int'l Conference on Managemetn of Data*, 153-164, Dallas, Texas, May, 2000.

[Liu et al., 1999] Liu, L., Pu, C., Tang, W. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and data Engineering* (*TKDE*), 11(4), 610-628, July 1999.

[Ludascher et al., 2002] Ludascher, B., Mukhopadhyay, P., Papakonstantinou, Y. A Transducer-Based XML Query Processor. In *Proc. of the 28th Int'l Conference on Very Large Data Bases* (*VLDB'02*), 227-238, Hong Kong, China, August 2002.

[Luo et al., 2005] Luo, C., Thakkar, H., Wang, H., and Zaniolo, C. A Native Extension of SQL for Mining Data Streams. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data* (*SIGMOD'05*), 873-875, Baltimore, MD, June 2005.

[Madden et al., 2002] Madden, S., Shah, M.A., Hellerstein, J.M., and Raman, V. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data* (*SIGMOD'02*), 49-60, Madison, WI, June 2002.

[McCanne et al., 2003] McCanne, S., Jacobson, V., Vetterli, M. Receiver-Driven Layered Multicast. In *Proc. of the 1996 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications* (*SIGCOMM'96*), 117-130, Palo Alto, CA, August 2003.

[McHugh and Widom 99] McHugh, J. and Widom J. Query Optimization for XML. In *Proceedings of the 25th International Conference on Very Large Data* Bases (*VLDB'99*), 315-326, Edinburgh, Scotland, September, 1999.

[Megginson, 2000] Megginson, D., Simple API for XML (SAX). http://www.saxproject.org/about.html, 2000.

[Microsoft, 2004] Microsoft Corporation. BizTalk Server. http://www.microsoft.com/biztalk, 2004.

[Motwani et al., 2003] Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proc. of the 1st Biennial Conference on Innovative Data Systems Research* (*CIDR'03*), Asilomar, CA, January 2003.

[Mühl et al., 2002] Mühl, G., Fiege, L., Gärtner, F.C., and Buchmann, A. Evaluating Advanced Routing Algorithms for Content-Based Publish/Subscribe Systems. In *Proc. of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems* (*MASCOTS'02*), Fort Worth, TX, October 2002.

[NASA, 2003] National Aeronautics and Space Administration. XML Group Resources Page. http://xml.gsfc.nasa.gov/, 2003.

[NASDAQ, 2005] The NASDAQ Stock Market. http://www.nasdaq.com/, 2005.

[Nestorov et al., 1997] Nestorov, S., Ullman, J.D., Wiener, J.L., and Chawathe, S.S. Representative objects: Concise representations of semistrctured hierarchical data. In *Proc. of the 13th International Conference on Data Engineering* (*ICDE 1997*), 79-90, Birmingham U.K., April, 1997.

[NetLogger, 2002] The NetLogger Toolkit. http://www-didc.lbl.gov/NetLogger/, 2002.

[Nguyen et al., 2001] Nguyen, B., Abiteboul, S., Cobena, G., and Preda, M. Monitoring XML data on the Web. In *Proc. of the 2001 ACM SIGMOD International Conference on Management of Data* (*SIGMOD'01*), 437-448, Santa Barbara, May 2001.

[OASIS WSN TC, 2005] OASIS Web Services Notification (MSN) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn, 2005.

[Oki et al., 1993] Oki, B., Pfleugl, M., Siegel, A., and Skeen, D. The Information Bus: An Architecture for Extensible Distributed System. In *Proc. of the 14th ACM Symposium on Operating System Principles* (*SOSP'93*), 58-68, Asheville, North Carolina, December 1993.

[Olteanu et al., 2003] Olteanu, D., Kiesling, T., and Bry, F. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. In *Proc. of the 19th Int'l Conference on Data Engineering* (*ICDE'03*), 702-711, Bangalore, India, March 2003.

[Opyrchal et al., 2000] Opyrchal, L., Astley, M., Auerbach, J., Banavar, G., Strom, R., and Sturman, D. Exploiting IP Multicast in Content-Based Publish-Subscribe Systems. In *Proc. of IFIP/ACM Int'l Conference on Distributed Systems Platforms*, 185-207, New York, NY, 2000.

[Oracle, 2005] Oracle Streams Advanced Queuing. http://www.oracle.com/technology/products/aq/index.html, 2005.

[Oracle-PeopleSoft, 2005] Oracle. PeopleSoft enterprise. http://www.oracle.com/applications/peoplesoft-enterprise.html, 2005.

[Ozen et al., 2001] Ozen, B., Kilic, O., Altinel, M., and Dogac, A. Highly personalized information delivery to mobile clients. In *Proc. of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access* (*MobiDe 2001*), 35-42, Santa Barbara, CA, 2001.

[PlanetLab, 2005] PlanetLab. http://www.planet-lab.org, 2005.

[QuoteMedia, 2005] QuoteMedia, Inc. Dynamic Market Data Solutions. http://www.quotemedia.com/, 2005

[Raggett et al., 1999] Raggett, D., Hors, A.L., and Jacobs, I. HTML 4.01 Specification. W3C recommendation 24 December 1999. http://www.w3.org/TR/REC-html40/, 1999.

[Rodriguez, 1998] Rodriguez, P., Ross, K.W., and Biersack, E.W. Improving the WWW: Caching or Multicast? *Computer Networks and ISDN Systems*, 30(22-23,25), 2223-2243, November 1998.

[Rosenthal and Chakravarthy] Rosenthal, A. and Chakravarthy, U.S. Anatomy of a Modular Multiple Query Optimizer. In *Proc. of the 14th Int'l Conference on Very Large Data Bases* (*VLDB'88*), 230-239, Los Angeles, CA, September 1988.

[Rogue Wave Software, 2004] Quovadx Inc. Rogue Wave Software. http://www.roguewave.com/, 2004.

[Roy et al., 2000] Roy, P., Seshadri, S., Sudarshan, S., and Bhobe, S. Efficient and extensible algorithms for multi-query optimization. In *Proc. of the 2000 ACM SIGMOD Int'l Conference on Management of Data*, 249-260, Dallas, Texas, May, 2000.

[Salton, 1989] Salton, G. Automatic Text Processing. Addison-Wesley Co., Boston, MA, 1989.

[Schreier et al., 1991] Schreier, U., Pirahesh, H., Agrawal, R., and Mohan, C. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. of the* 17th *International Conference on Very Large Data Bases* (*VLDB 1991*), 469-478, Barcelona, Spain, September, 1991.

[SECSG, 2004] SouthEast Collaboratory for Structural Genomics. http://www.secsg.org/, 2004.

[Segall et al., 2000] Segall, B., Arnold, D., Boot, J., Henderson, M., and Phelps, T. Content Based Routing with Elvin4. In *Proc. of AUUG2K*, Canberra, Australia, June 2000.

[Sellis, 1988] Sellis, T.K. Multiple-Query Optimization. *ACM Transactions on Database Systems* (*TODS*), 13(1), 23-52, Mar. 1988.

[Shah et al., 2003] Shah, S., Dharmarajan, S., and Ramamritham, K. An Efficient and resilient Approach to Filtering and Disseminating Streaming Data. In *Proc. of the 29th Int'l Conference on Very Large Data Bases* (*VLDB'03*), 57-68, Berlin, Germany, September 2003.

[Shah et al., 2002] Shah, R., Jain, R., and Anjum, R. Efficient Dissemination of Personalized Information Using Content-Based Multicast. In *Proc. of IEEE INFOCOM 2002*, New York, NY, June 2002.

[Snoeren et al., 2001] Snoeren, A.C., Conley, K., and Gifford, D.K. Mesh-Based Content Routing using XML. In *Proc. of the 18th ACM Symposium on Operating Systems Principles* (*SOSP'01*), Banff, Canada, October 2001.

[Solace Systems, 2005] Solace Systems, Inc. http://www.solacesystems.com/, 2005.

[Stoica et al., 2002] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet Indirection Infrastructure. In *Proc. of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communities* (*SIGCOMM'02*), 73-88, Pittsburgh, PA, August 2002.

[Stonebraker et al., 1990] Stonebraker, M., Jhingran, A., Goh, J., and Potamianos, S. On rules, procedures, caching and views in data base systems. In *Proc. of the 1990 ACM SIGMOD Int'l Conference on Management of Data* (*SIGMOD 1990*), 281-290, Atlantic City, NJ, May, 1990.

[Sun Microsystems, 2000] Sun Microsystems, Inc. JavaSpaces Service Specification. http://www.sun.com/software/jini/specs/jini1.1html/js-title.html, 2000.

[Sun Microsystems, 2001] Sun Microsystems, Inc. Java XML pack. Winter 01 update release. http://java.sun.com/xml/ downloads/javaxmlpack.html, 2001.

[Sun Microsystems, 2002] Sun Microsystems, Inc. Java Message Service (JMS). http://java.sun.com/products/jms/, 2002.

[Sybase, 2005] Sybase, Inc. Financial fusion message broker. http://www.sybase.com/products/industrysolutions/messagebroker, 2005.

[Taleo, 2005] Taleo, Co. Talent Management Drives the Enterprise. http://www.taleo.com/en/default.php, 2005.

[Terry et al., 1992] Terry, D.B., Goldberg, D., Nichols, D.A., and Oki, B.M. Continuous queries over append-only databases. In *Proc. of the 1992 ACM SIGMOD Int'l Conference on Management of Data* (*SIGMO 1992*), 321-330, San Diego, CA, June, 1992.

[TIBCO Software, 2002] TIBCO Software, Inc. Enterprise Application Integration Solutions. http://www.tibco.com/solutions/tibco_eai.pdf, 2002.

[Thompson et al., 2004] Thompson, H., Beech, D., Maloney, M., and Mendelsohn, N. XML Schema Part 1: Structures. http://www.w3.org/TR/xmlschema-1, October 2004

[Tian et al., 2004] Tian, F., DeWitt, D., Pirahesh, H., Reinwald, B., Mayr, T., and Myllymaki, J. Implementing a Scalable XML Publish/Subscribe System Using a Relational Database System. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data*, 479-490, Paris, France, June, 2004.

[Tolani et al., 2002] Tolani, P.M., and Haritsa, J.R. XGRIND: A Query-Friendly XML Compressor. In *Proc of the 18th Int'l Conference on Data Engineering* (*ICDE'02*), 225-234, San Jose, CA, March 2002.

[UserLand Software, 2005] UserLand Software. UserLand RSS Central. http://rss.userland.com, 2005.

[W3C, 2002] World Wide Web Consortium. Web Services activity. http://www.w3.org/2002/ws/, 2002.

[Watson, 1997] Watson, B.W. Practical optimization for automata. In *Proc. of the 2nd International Workshop on Implementing Automata*, 232-240, Berlin, Germany, 1997.

[Widom and Finklestein, 1990] Widom, J., and Finklestein, S.J. Set-oriented production rules in relational database systems. In *Proc. of the 1990 ACM SIGMOD Int'l Conference on Management of Data* (*SIGMOD 1990*), 259-270, Atlantic City, NJ, May, 1990.

[Wutka, 2000] Wutka Consulting, Inc. DTD parser. http://www.wutka.com/dtdparser.html, 2000

[Yahoo!, 2005] Yahoo! Inc. My Yahoo! http://my.yahoo.com/, 2005.

[Yan and Garcia-Molina, 1994] Yan, T. W., and Garcia-Molina, H. Index Structures for Selective Dissemination of Information Under Boolean Model. *ACM Transactions on Database Systems* (*TODS*), 19(2), 332-364, June 1994.

[Yan and Garcia-Molina, 1999] Yan, T. W., and Garcia-Molina, H. The SIFT Information Dissemination System. *ACM Transactions on Database Systems* (*TODS*), 24(4), 529-565, December 1999.

[Zhang et al., 2001] Zhang, C., Naughton, J.F., DeWitt, D.J., Luo, Q., and Lohman, G.M. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, 425-436, Santa Barbara, CA, June, 2001.