XML Message Filtering and Matching in Publish/Subscribe Systems

By

Liang Dai

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirement for the degree of

MASTER OF COMPUTER SCIENCE

School of Computer Science

at

Carleton University

Ottawa, Ontario

September, 2009

Your file Votre référence
ISBN: 978-0-494-60245-4
Our file Notre référence
ISBN: 978-0-494-60245-4

Abstract

In publish/subscribe systems, XML message filtering performed at application layer is an important operation for XML message multicast. As a specific case of content-based multicast in application layer, XML message multicast depends on the data filtering and matching processes and the forwarding and routing schemes. As the XML data emerges in transition, XML message filtering and matching becomes more and more desirable. BFilter, proposed in this thesis, conducts the XML message filtering and matching by leveraging branch points in both the XML document and user profile. It evaluates user profiles that use backward matching branch points to delay further matching processes until branch points match in the XML document and user profile. In this way, XML message filtering can be performed more efficiently as the probability of mismatching is reduced.

Acknowledgements

The author wishes to thank both my Co-Supervisors Professor Shikharesh Majumdar and Professor Chung-Horng Lung for the great assistance and insightful feedback. As well, the author wishes to thank Alcatel-Lucent and Ontario Centres of Excellence for their financial support throughout the project.

Table of Contents

List of Figures

List of Acronyms

DFA	Deterministic Finite Automata
GTP	Generalized-Tree-Pattern
MEDYM	called Match-Early and Dynamic Multicast
NFA	Nondeterministic Finite Automata
ODMRP	On-Demand Multicast Routing Protocol
PUB/SUB	publish/subscribe system
SAX	SAX Event-based XML Parser
SPBM	Scalable Position-Based Multicast protocol
SRMP	Source Routing-based Multicast Protocol
TOP	Tree-of-Path (TOP) encoding
XML	Extensible Mark-up language
Xpath	XML Path Language
Xquery	XML Query Language

Chapter 1: Introduction

1.1 Introduction and Motivation

In publish/subscribe (pub/sub) systems, application layer multicast is widely used for data dissemination to subscribers. The construction and maintenance of the overlay structure are the main issues in application layer multicasting.

In network layer multicast proposals, packets are replicated and forwarded by network routers. Application layer multicast uses an overlay on top of the physical network for communication between hosts. Application layer multicast has some advantages over network layer multicast. First, it does not need to make changes at the router level, which makes implementation more flexible. Second, it can arbitrarily group receivers from any location in the network. Network layer multicast, however, uses IP addresses to restrict receivers to certain subnets. This means that the receivers within a subnet are usually grouped geographically, which is not appropriate in the case of pub/sub systems, in which the receivers (subscribers) of a particular interest can be located anywhere in the network.

Application layer multicast has some disadvantages. First, the packets are sent along the application overlay layer from a source to a destination, instead of following the shortest path at the network layer. Thus, the path traversed by a packet may be longer in comparison to network layer multicast. Moreover, duplicated packets may occur at some links. Thus, the main challenge in application layer multicast is for end systems to construct effective overlay structures.

In event-based pub/sub services, a subscriber registers a subscription to the pub/sub service and receives published messages that match the subscription. Intuitively, the source (publishers) can allow their subscribers to retain whatever they want, and send all the data to all subscribers. This approach is definitely not efficient because there are too many duplicated data packets that reduce system throughput and waste network resources as well as increase the processing overhead for the intermediate nodes.

Generally speaking, there are two ways to carry out multicast in the context of pub/sub systems [2, 5, 9, 10, 12, 16, 18, 24, 33, 34]. The first is to find the subscriber by using the subscription information, and then send appropriate data to these subscribers. Data matching can be performed either at the source or at some centralized brokers. The second method is to perform data matching on the fly. In this way, the source simply pushes the data into the network that has a multicast tree composed of routers or brokers. The routers or brokers on the tree have filters to dispatch proper subsets of data to their children. The children in turn perform data matching and dispatching and forward the matched data to their children. This continues until the filtered data reaches the subscribers.

The first approach described above may use keyword-based multicast and distributed hash table-based multicast. Keyword-based multicast groups subscribers using the keywords in their subscriptions [2, 16, 21, 22, 25, 28, 31]. Distributed hash table-based multicast uses hash functions to assign keys to subscribers by using their subscriptions [6]. These methods are efficient in terms of delivery speed. However, the keyword-based approach is less expressive because the subscriptions contain only keywords. The distributed hash table approach is not content-aware. In these methods,

data matching is based on the key or keywords but not the content. The second approach delivers data according to the content. The subscription description is used to perform the matching. The subscription can be presented either in an n-tuple containing *n* information spaces, or in XPath expressions [1, 8, 18, 19]. An XPath expression is used for addressing portions of a XML file. A path expression is written as a sequence of steps to get from one XML tag to another tag or set of tags. The steps are separated by "/" (i.e., path) characters. For instance, if a subscription contains A, B and C, it may be {A, B, C} in the n-tuple expression, but in an XPath expression it would be A/B/C or A/C/B or B/C/A and so on. Obviously, XPath is more expressive than n-tuple because the two subscriptions, A/B/C and A/C/B, are definitely different in XPath, but they cannot be distinguished by using n-tuple.

A XML file is a tree-based structure for describing information. The data content is available between a start tag and an end tag. The pair of tags not only scopes the data it contains, but also describes the data, possibly with some constraints on the tags. One XML document has one root tag pair. The root tag pair can have child tag pairs and the children can have their own child tag pairs, and so on. This structure forms a tree with one single root. As a XML file is structured, it naturally applies filters in the hierarchy to perform data matching and delivery. XML-based multicast can properly match and deliver messages to subscribers. However, because it is more difficult to index and identify the elements in the XML file, compared to the content-based message format, which can be considered to be an n-dimensional array containing keywords, the filtering process in each node is time consuming. Hence, the performance of XML-based multicast depends heavily on the approach used to process the XML message.

This thesis examines different filtering approaches used in the area of application layer multicast and proposes a novel XML message filtering algorithm—Bfilter. Bfilter realizes the tree structure in both XML documents and user requests with nested paths. It conducts the XML message filtering and matching process by identifying branch points in both XML documents and user requests. The evaluation of user profiles uses backward matching branch points to delay further matching, so that the probability of a mismatch is reduced and XML message filtering can be performed more efficiently.

1.2 Contributions

Current best practices of XML message filtering are not efficient for complex queries. A complex query is a query that consists of nested paths. A simple query has no nested path. Yfilter [17] deals with a complex query by decomposing it into simple queries that have no nested path. The simple queries are evaluated separately. A post-processing is needed to identify whether all simple queries of the complex query match. A failure of a simple query match will not disrupt the processing of other queries. Afilter [29] matches queries backwards. It emphasizes suffix commonality to reduce mismatches but only deals with simple queries. Gfilter inherits backward matching from Afilter and uses a more efficient indexing approach, but the decomposition of a complex query is needed in the indexing structure.

This thesis proposes a novel algorithm, Bfilter, to handle complex queries without decomposition. Bfilter's contribution is to emphasize branch point matching during the bottom-up process to reduce the possibility of a mismatch, allowing the matching of complex queries to be processed more efficiently. Experimental results from the current implementation demonstrate that BFilter is faster for complex queries. However Bfilter

can be slower than Yfilter when the number of queries is very large. Modifications to the current implementation can overcome this shortcoming, and will be discussed in chapter 5.

1.3 Thesis Organization

In Chapter 2, background information on application layer multicast is presented. Application layer multicast is categorized into three groups, each of which is described in detail. The last part of this Chapter discusses the three topical XML filters—Yfilter, Afilter, and Gfilter. Chapter 3 describes Bfilter in detail and presents the design and implementation of Bfilter. Chapter 4 illustrates the performance results of Bfilter. Finally, Chapter 5 summarizes the thesis and discusses directions for future research.

Chapter 2: Background

There are two important operations performed in a publish/subscribe system: XML message filtering and XML message multicast. This research focuses on techniques for XML message filtering. Before discussing the filtering technique proposed in this thesis, a brief overview of the existing works for both XML message multicast and XML message filtering is provided in this chapter. Application layer multicast can be briefly categorized into content-based multicast and keyword-based multicast. Since XML message multicast belongs to the category of content-based multicast and it is performed at the application layer, a discussion of general application layer multicast and content-based multicast are also included.

Section 2.1 presents a number of general issues regarding application layer multicast and investigates the multicast problem from a system point of view and considers different types of networks and network topologies. Section 2.2 presents multicast in the context of content-based systems where throughput and bandwidth are the main issues. It discusses how to forward a message to a large number of subscribers with a particular interest. Section 2.3 focuses on XML message filtering. Three popular XML filters—Yfilter, Afilter, and Gfilter, are discussed in detail.

2.1 General Issues in Application Layer Multicast

This section presents three general issues in the context of application layer multicast: packet duplication and delivery delay, bandwidth utilization in overlay construction, and wireless ad hoc environment multicast.

2.1.1 Packet duplication and delivery delay

Unlike network layer multicast, application layer multicast replicates packets within the end hosts. In Figure 2.1, the rectangles represent routers and the circles are end hosts. In the case of network layer multicast, when a packet is forwarded from end host 1 to end hosts 2, 3 and 4, it is replicated at routers A and B. However, in application layer multicast, it is replicated at end host 1. In this type of multicast, a so-called delivery tree must be built beforehand. In this case, end host 1 is the root, and end hosts 2 and 3 are its children. End host 4 is the child of 2. In other words, each node in the tree is responsible for replicating packets for all its children. Two problems arise with application layer multicast. The first is that multiple identical packets may be passing through a link.

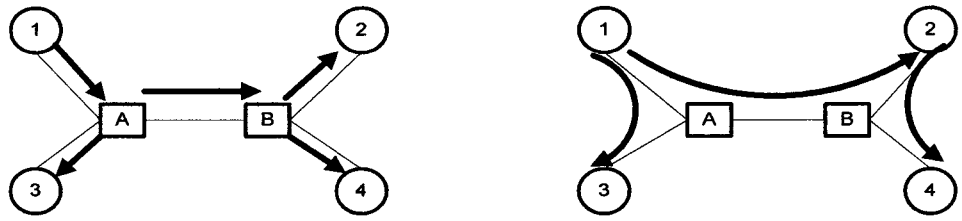


Figure 2.1 Network Layer Multicast versus Application Layer Multicast [2]

Figure 2.1 shows that the same packets pass along link 1-A twice. The second problem is that the path on the tree from the source to the destination is usually longer than the path directly obtained by a shortest-path routing algorithm. For instance, a packet going through 1-A-B-2-B-4 to end host 4 uses a longer path in comparison to 1-A-B-4.

To deal with these two problems and obtain a better solution, Banerjee et al. proposed a protocol that is based on hierarchical clustering of the application layer multicast peers [2]. In the hierarchical arrangement of hosts, all participating end hosts

are at layer 0. They are geographically divided into clusters; the size of each cluster must be within a predefined range. From each cluster, an end host located at the centre or close to the centre of the cluster is elected as a leader. All leaders form a layer; for example, layer 1. Again, the hosts at layer 1 are further divided into clusters and the leaders are selected to form another layer, and so on. Finally, only one is left, which is the root of the delivery tree.

Figure 2.2 illustrates the hierarchical arrangement. Topological clusters are represented by the dashed area. Whenever a packet needs to be delivered, it is forwarded only to the cluster leaders that know all of their members.

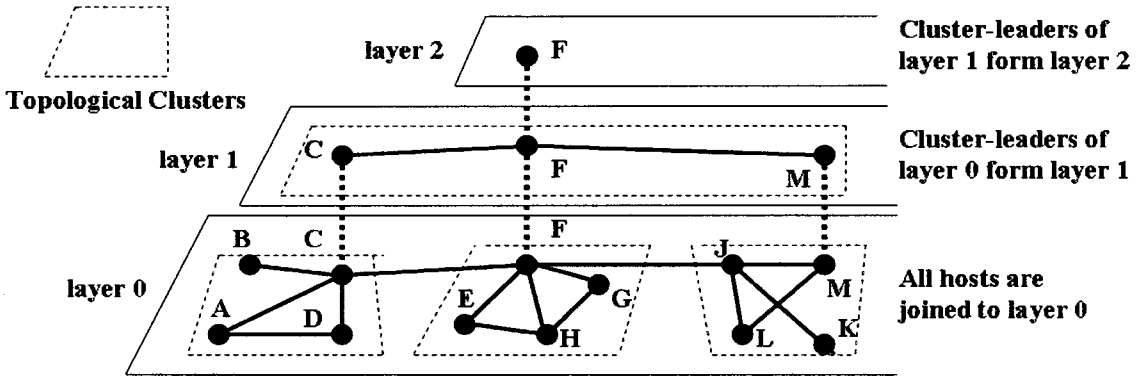


Figure 2.2 Hierarchical Clustering Arrangement [2]

In Figure 2.2, if host F wants to deliver a packet to hosts J and L, F first finds out to which cluster the destination belongs. In this particular case, M is the leader of J and L. Therefore, F forwards the packet to M, and M multicasts the packet to J and L. This architecture also allows other hosts to participate or leave this network by inserting themselves into or deleting themselves from the tree. As the packet goes through the leaders of higher layer(s) to the leaders of lower layer(s) until the destinations are reached, it avoids matching operations at each hop down to the destinations. Therefore, the packet can be delivered properly and efficiently. The results from [2] show that for a

group of 32 hosts or more, this technique has few packet duplications on links and low end-to-end latencies.

As application layer multicast shifts the functionality of multicast from the routers to the end hosts, the packets traverse through the overlay tree rather than along the shortest path via the network layer. Application layer multicast may duplicate packets on physical links and incur larger end-to-end delays. Chu et al. studied this problem in the context of the Narada protocol [16]. In this fully distributed protocol, the end hosts self-organize into an overlay structure and optimize the performance of the overlay structure. Narada builds an overlay network in two steps. The first step is to build a mesh containing all the members of the network. The mesh should have the following two desirable properties: the path between any pair of members should not be too long in comparison with the shortest path between them, and each member should not have too many neighbours in the mesh. The second step is to build a minimum spanning tree on this mesh. As the mesh contains all members of the network, all sources can build their own spanning tree on the mesh. To build and maintain the mesh and its trees, information from the network layer is taken into account.

From the simulation results based on a large multicast group, Chu et al. [16] concluded that transferring some multicast functionalities from the end hosts to the routers may significantly increase performance, compared with using only application layer multicast.

2.1.2 Bandwidth utilization

To build an overlay network, another important problem is determining how to utilize the network bandwidth properly and take it into account in the overlay

construction. A network is said to have a uniform bandwidth if all links have about the same amount of bandwidth. Ideally, a delivery tree is composed of links with the highest bandwidth available. This is not only because the links with high bandwidth can ensure high throughput, but also that higher bandwidth links are usually more stable and reliable. Obviously, in order to build such an overlay network with appropriate link bandwidth, additional research is needed. Bhargava et al. introduced a dynamic overlay network approach called Pagoda [5]. Pagoda can be used to dynamically organize nodes with non-uniform bandwidth. The overlay is constructed and maintained using a sophisticated algorithm in Pagoda. For nodes with uniform bandwidth, any node insertion or deletion in an overlay can be executed in O (log n) time, and for nodes with non-uniform bandwidth, any node insertion and deletion can be executed in O ($\log^2$ n) time [5].

2.1.3 Multicast in ad hoc wireless networks

In wireless networks, multicast faces additional problems due to the nature of the wireless environment. This section presents some of the research efforts in mobile ad hoc networks.

Mobile ad hoc networks suffer from frequent link failures. In an ad hoc network, each node can communicate with others as a host and can also deliver packets to other hosts as a router. A node participates in the network in a temporary manner and can withdraw suddenly from the network without informing its neighbours. In this type of network, it is difficult to construct a stable overlay structure. A single link failure may destroy the entire established structure. In this case, the overlay structure has to be built on demand and the structure has to be tolerant of link failures during packet delivery. In

other words, multiple packets have to be delivered along multiple routes to ensure that at least one packet can arrive at the receivers.

Moustafa et al. proposed a novel on-demand multicast routing protocol named Source Routing-based Multicast Protocol (SRMP) [23]. This protocol applies the source routing mechanism that constructs a mesh to connect group members, providing robustness for the unstable linkage between mobile nodes. For each multicast process, SRMP establishes a multicast mesh starting from multicast receivers. The mesh construction takes two important factors into account: signal strength at the chosen links and power consumption. During the operation, a request phase and a reply phase are used. During the request phase, routes to the multicast group are found, and it is used when a source node wants to join a group. The source node broadcasts a Join-request message to the multicast group. The reply phase starts when each multicast receiver has received the Join-request message. The receivers reply with a Join-reply message if some predefined criteria are satisfied.

Yoneki et al. introduced another approach to dealing with multicast in ad hoc network-based pub/sub systems, which dynamically constructs an event dissemination structure to route events from event publishers to subscribers [32]. This approach uses a message-dissemination mechanism and extends the On-Demand Multicast Routing Protocol (ODMRP) [22]. ODMRP supports optimized data dissemination mechanisms in terms of network topology, network capabilities and mobility. ODMRP is an on-demand routing technique, the same as the approach in [23]. It also uses a mesh-based approach to provide reliable connectivity among multicast members. It creates a group of forwarding nodes between the source and the multicast receivers. These nodes forward all packets

they receive to all interested multicast receivers. For any particular destination, ODMRP establishes multiple routes to ensure packet delivery. Yoneki et al. incorporated a content-based message-dissemination mechanism in ODMRP by inserting content-based data into ODMRP packets to guide packet delivery. Content-based subscriptions at broker nodes are aggregated and summarized into a compact data format using the Bloom filter [6], a simple space-efficient probabilistic data structure that is used to test whether an element is a member of a set. The publisher broker carries out multicast according to the propagated subscriptions.

To consider the geographic location of nodes involved in a multicast, Transier et al. proposed the Scalable Position-Based Multicast protocol (SPBM) for ad hoc networks [30]. This protocol uses the geographic position of nodes to form group memberships and to forward data packets. There are two building blocks in this protocol: a group management scheme and a forwarding strategy. The group management scheme is responsible for disseminating the membership information for multicast groups. The forwarding strategy is used to make packet-forwarding decisions. A forwarding node selects one of its neighbours as next hop by considering whether the node can send the packet to the geographic position of the destination. If it has no neighbour that forwards to the destination, a recovery strategy is used to back up and find another path to the destination. As forwarding decisions are based only on local knowledge, there is no need to create and maintain routes from the sender to the destination.

2.1.4 Summary

To summarize, this section showed that overlay structure construction as well as maintenance of the structure are the main issues for application layer multicast in both

wired and wireless networks. The approaches discussed above focus on the effects of the overlay structures on packet duplication and delivery delay, rather than specific techniques for data forwarding and matching, such as XML message filtering and matching. Once we investigate the details of forwarding and matching techniques according to different subscription expressions, other issues will have to be considered, as will be seen in later discussions.

2.2 Multicast in Content-Based Routing

As discussed in Chapter 1, there are two ways to accomplish multicast in the context of pub/sub systems. The first is to find the subscriber by using the subscription information and send appropriate data to these subscribers. The second method is to perform data matching on the fly. The first approach uses keyword-based multicast and distributed hash table-based multicast, which are efficient in terms of delivery speed but less expressive. The second approach is content-based multicast, which delivers data according to the content. The subscription description is used to perform the matching. The subscription can be presented either in an n-tuple containing n information spaces or in XPath expressions. These types of multicast technique face a primary challenge in terms of delivery delay and the construction and maintenance of multicast trees (or forwarding trees). Langerman et al. proposed an O (n log n) dynamic programming algorithm to calculate the location of the filters that can reduce the filtering delay [21].

As distributed hash table-based multicast is efficient in terms of data delivery, and filtering-based multicast supports highly expressive subscriptions, Perng et al. integrated these two models [26] in a way that merges the advantages of both systems. Instead of using the distributed hash table approach, this technique uses an encoding scheme that

maps each distinct event into an n-bit pattern. Each bit is mapped to an attribute in the universal information space. The i-th bit set to 1 indicates that the event is related to the i-th attribute. Each subscription contains an n-bit pattern and each bit that is set indicates an interest for the corresponding attribute. Thus, a subscription is a conjunction of the indicated interests.

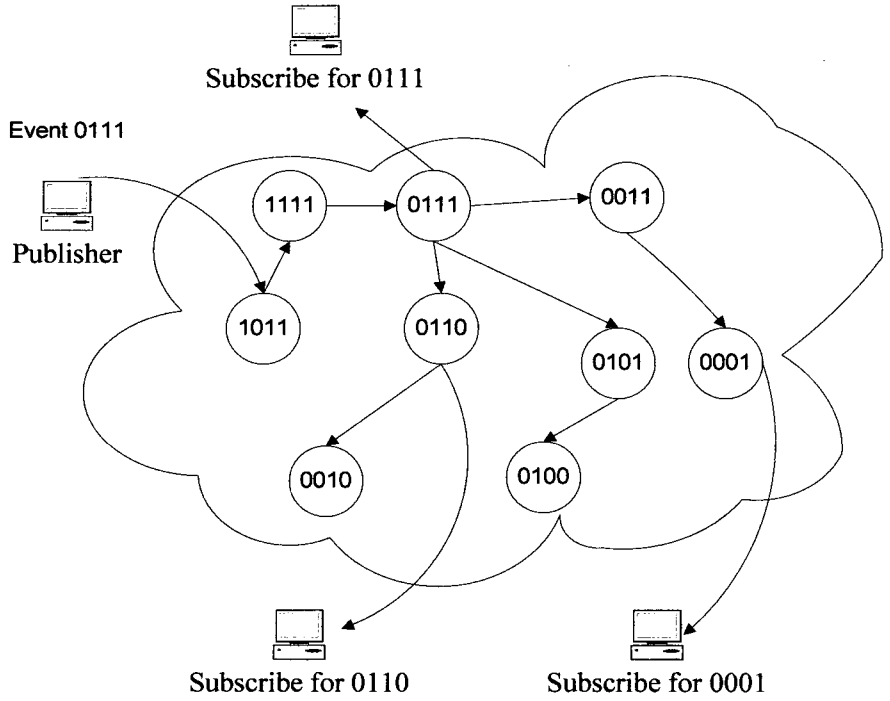


Figure 2.3 Dissemination of Event 0111 [26]

An event matching a subscription means that for every set bit in the subscription, the bit in the event pattern is also set. To form an identifier hierarchy, the parent/child relationship follows a rule, such as the identifier 0011 is a parent for both 0001 and 0010. In other words, a parent identifier contains at least all the attributes of a child identifier. This hierarchy is mapped to the nodes in the overlay network, as shown in Figure 2.3. Each node (represented by a circle) holds some subscriptions (only one is shown in the figure) and is also responsible for forwarding messages to interested clients.

A publisher that publishes an event with identifier 0111 first reaches the node hosting 0111, which is called the rendezvous node. A rendezvous node maintains a sub-tree that contains all subscribers with child identifiers. In Figure 2.3, the rendezvous node multicasts the event to its three children, host 0110, 0101 and 0011, respectively. The children, in turn, send the event to their own children. The proposed protocol provides efficient and scalable content-based services in a dynamic network [26].

Chand et al. proposed a novel protocol for content-based routing in overlay networks that ensures a message can be received by all subscribers that have registered matching subscriptions [10]. This technique attempts to optimize network bandwidth usage. In this protocol, a subscription aggregation approach is used to minimize the size of the routing tables by removing redundant subscriptions. For example, node A has two subscriptions, say S1 and S2 from nodes B and C, respectively, and assume S1 contains S2 in the sense that any event matching S2 also matches S1. S2 is redundant and does not need to be propagated upstream to A's parent in the tree.

This protocol is used in a scalable XML-based data dissemination system and the result shows that the size of the routing tables is small even with large populations of subscribers.

In summary, content-based multicast intends to achieve better performance in terms of forwarding and matching speed, and delivery quality in the sense that the subscribers can obtain more accurate messages of interest by using more expressive subscriptions. Unfortunately, it seems that achieving both at the same time is difficult. Typically, we obtain one at the expense of the other. Many approaches that try to effectively balance the two have been proposed. These will be discussed in next section.

2.3 XML Message Filters

This section focuses on XML message filtering. A XML filter is composed of software components containing user requests or queries, data structures and an underlying filtering algorithm. User requests are usually represented by XPath notation. The currently proposed filtering algorithms [1, 4, 7, 11, 18, 20, 29] have different capabilities for handling user requests, which will be discussed in subsequent sections.

2.3.1 Overview of XML Message Filtering

A XML file is a tree-based structure for describing information. As a XML file is structured, it naturally applies filters in the hierarchy to perform data matching and delivery. XML-based multicast can properly match and deliver messages to subscribers. However, since it is more difficult to index and identify the elements in a XML file, the filtering process in each node is time consuming. Hence, XML-based multicast depends heavily on the approach used to process the XML message. ONXY [18], which is derived from YFilter [17], is one of the most popular techniques used in XML-based multicast. In ONXY, the subscriber interests are aggregated at each node in a distribution tree rooted at a publisher. In turn, the published information needs to be matched at each intermediate node along the paths in the tree, which is time consuming.

To overcome this problem, Fenner et al. proposed a technique called XTreeNet, which utilizes a combination of the pub/sub and the query/response model to obtain better performance [19]. XTreeNet uses a content descriptor (CD) to describe the information of subscriptions and published events. A distribution tree is established for each CD that is rooted at each publisher.

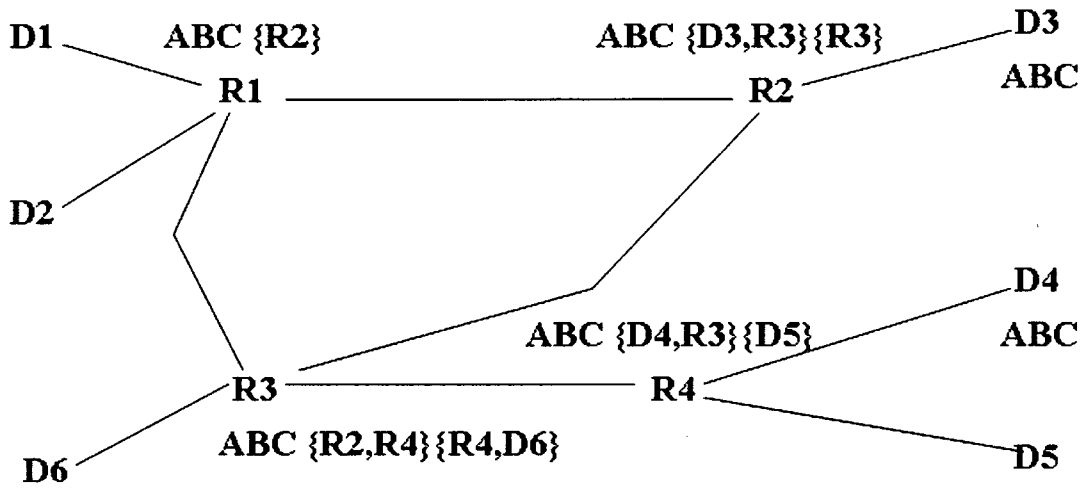


Figure 2.4 XTreeNet: An Example [19]

For example, in Figure 2.4, D_i's represent end nodes and R_j's represent routers. Each router stores tuples for different CDs. Each tuple has three items. The first item is the CD to which the tuple is related. The second item is a bracketed list called the publishers interface list, containing interfaces towards the publishers for this CD. The last bracketed list is the subscribers interface list, containing interfaces towards the subscribers for this CD. For instance, R2 has a tuple as (ABC {D3, R3} {R3}). The CD is ABC. The publisher interfaces are D3 and R3. Thus, D3 is one of the publishers. Another publisher can be reached from R3, which is the next hop towards the publisher. From the publishers interface list of R3, the publisher D3 and D4 can be reached through R2 and R4, respectively.

R2's subscribers interface list indicates that R3 is the next hop towards the subscribers. In the tuple stored in R3, the subscribers interface list shows that D6 is one subscriber and R4 is the next hop towards another subscriber. The subscribers interface list of R4 shows another subscriber, D5. The subscribers interface list stored in R1 is empty, meaning that there is no subscriber for this CD from R1.

In the figure, node D3 produces a data item with CD = <ABC>, and router R2 is pre-selected as the coordinator that constructs a hash value based on the CD. Router R2 then floods all the overlay routers in the network with the tuple (hash ID, <ABC>, coordinator-id). The state stored in all the routers includes the tuple (hash ID, <ABC>, publisher interface list). If node D4 publishes the same CD of <ABC>, router R4 sends a publisher-join event through the topology to the coordinator R2 via R3. When R3 receives the publisher-join, it adds its link to R4 to the publisher interface list. Thus, a tree of publishers is formed. As there is no explicit matching of data for the subscriber interests, data and query forwarding are very efficient.

When YFilter is used to filter documents, if there are a large number of subscriptions it may fail to deliver packets due to the limited bandwidth. Burstein et al. proposed an approach to overcome this weakness in YFilter [8]. They described a peer-to-peer scheme that exploits the bandwidth of participating subscribers for data dissemination by building an unstructured overlay network. This protocol solves the problem of matching high-volume document streams against a huge set of subscriptions. In this proposed scheme, a publisher edge filter (the node adjacent to the publisher) constructs the dissemination tree structure on the fly before sending the document.

Figure 2.5 shows a multicast tree rooted at a filter. The tree structure is forwarded together with the document. At the filter, each document is accompanied by the IP addresses of all its intended recipients. Each node extracts the IP addresses of its intended children in the dynamic multicast tree while receiving the document, and delivers the document to its downstream children. In Figure 2.5, for example, the filter sends router 2 document D with the IP addresses of router 5, 6, 10, 11 and 12. Router 2 sends the

document to its two children, routers 5 and 6, with the IP address of router 10 and the IP addresses of routers 11 and 12 respectively.

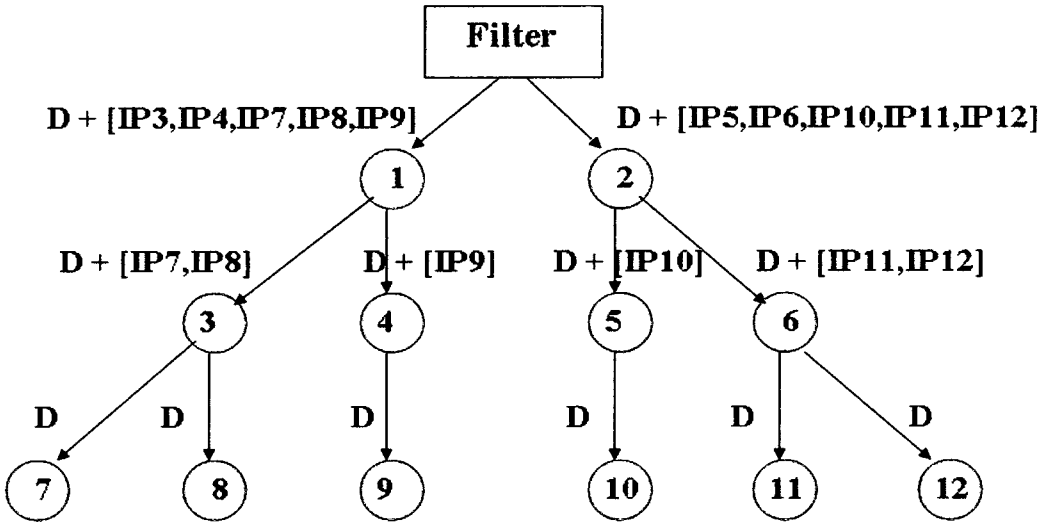


Figure 2.5 The Multicast Tree Built by the Filter [8]

As mentioned in Section 2.2, Chand et al. proposed a novel approach for content-based routing in overlay networks for optimizing network bandwidth usage [10]. Later, they presented a XML content network architecture called XNET [12]. This approach ensures that all registered subscriptions can be properly matched at all times and the network can recover from router or link failures.

In XNET [12], a message traverses a link only if there are some subscribers of interest downstream. This optimizes the link usage. Each node in the network contains a routing table. The routing table contains a set of subscriptions in which the neighbour nodes of the table's owner are interested. When subscribers register or cancel a subscription, the nodes of the overlay must update their routing tables accordingly. XNET also uses the same subscription aggregation approach used in [10] to reduce the size of the routing table. XNET implemented mechanisms such as a crash/recover scheme to deal with router or link failures.

Cao et al. proposed an approach called Match-Early and Dynamic Multicast (MEDYM) to overcome the weakness of Yfilter [9]. In MEDYM, when an event is published, it is first matched against subscriptions from remote servers to obtain a list of destination servers with matched subscriptions. The event is routed to the destination servers through a dynamic multicast tree. Based on its destination list, a server dynamically computes the next-hop servers to which to forward the message, as well as the new destination list for each of the next-hop servers. A dynamic multicast tree is constructed on the fly.

This section has shown that pure XML-based multicast as used in ONXY incurs a performance penalty in data delivery and matching. A number of approaches are available to obtain better performance, either by modifying the Yfilter technique or by using some technique from content-based multicast.

2.3.2 Yfilter

Xfilter [1] is a popular algorithm based on deterministic finite automata, which uses linked lists to store user requests and handle each request individually. It is capable of handling XPath relationship notations, such as ancestor/descendant (represented by '//' in XPath) as well as wildcard '*'. Because it stores requests separately, the same segments of a different request cannot share storage space and have to be matched individually, which results in poor performance.

Yfilter [17] overcomes the disadvantage of Xfilter by using nondeterministic finite automata. It uses a tree to store the user requests, which allows the same segments of different requests to share the same path in the tree. Thus, the number of active states that match an incoming event will decrease. However, the ancestor/descendant

relationship introduces more matching states, which may result in the number of active states increasing exponentially [29].

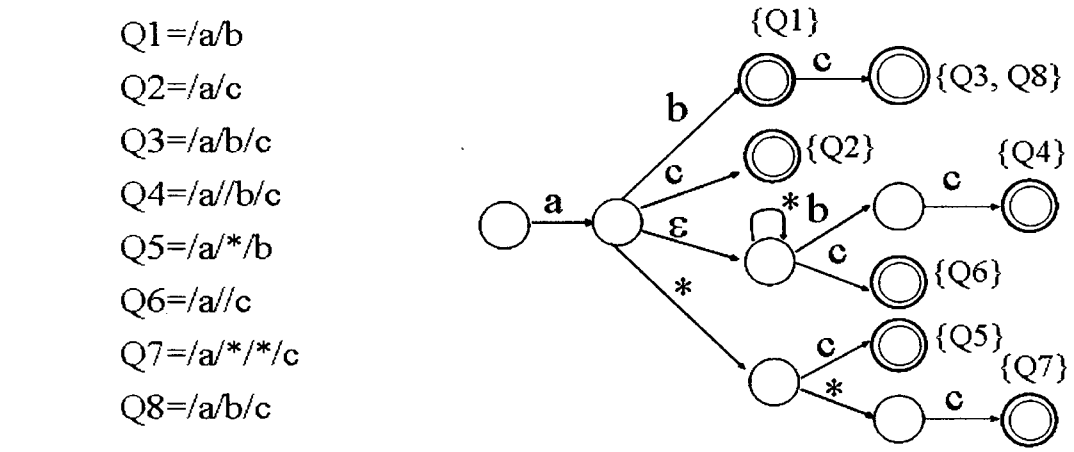


Figure 2.6 NFA Representation for Queries [17]

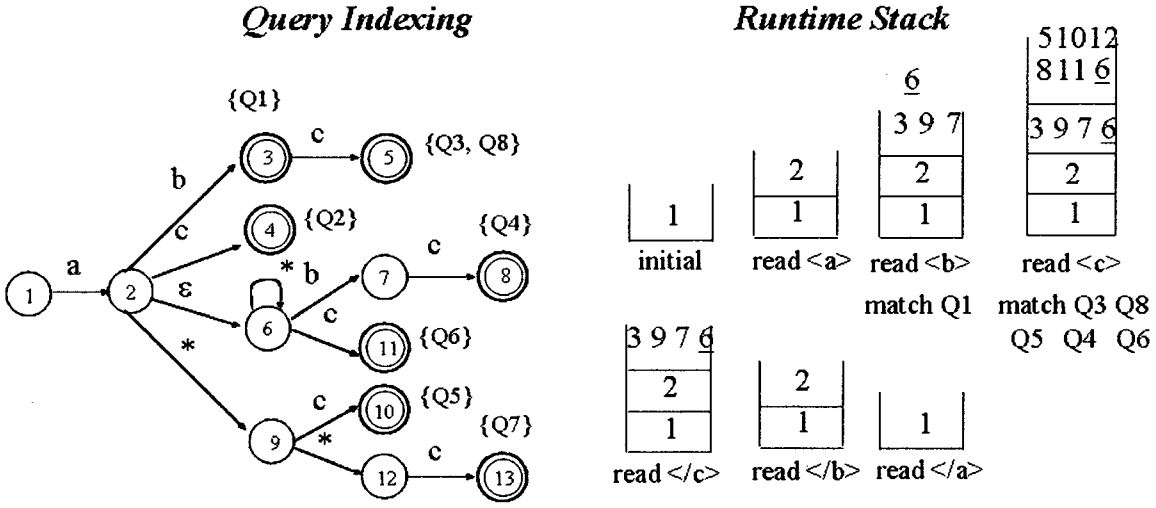


Figure 2.7 Yfilter Matching Process [17]

As shown in Figure 2.6, Yfilter constructs a NFA to hold all queries. The labels of the queries form a tree so that the common prefixes of the paths are represented only once in the structure. On the left side, the eight queries are represented using XPath notation.

Their corresponding NFA structure is shown on the right side. A state is denoted by a circle. The two concentric circles mean that this state is an accepting state for one or more queries. The queries' IDs are shown in accepting states.

In Figure 2.6, the wildcard "*" matches any element. The symbol "ε" means that the transition needs no input. Queries Q1, Q3 and Q8 share the same states from transition a and b. Because Q3 and Q8 are identical they share the same accepting state.

The implementation of NFA uses a "hash table-based" approach, as shown in Figure 2.7. Runtime stack, a stack mechanism, is used to hold the states and enable backtracking when an accepting state is reached. Detailed explanation of Figure 2.7 is described below.

When a start tag is read in the NFA, execution follows all matching transitions from all currently active states. For each active state, it first checks if the tag name is in the state's hash table; if it is, the corresponding state ID is added to a set of target states. It then checks whether the wildcard "*" exists in the hash table. Because the wildcard matches any tags, the corresponding state ID will be in the "target states" if it is present in the hash table. If the state itself is a "//-child" (i.e. an ancestor/descendant relationship in XPath), the state ID will be added to the set. Finally, it checks the "ε" symbol. If the symbol is in the hash table, the //-child state indicated by the corresponding state ID is processed recursively following the previous steps.

When all the currently active states have been checked, the final set of target states is pushed onto the top of the run-time stack. The target states are active states for next input tag. When an end tag is arrived, the NFA execution simply pops the top set of states off the stack.

In Figure 2.7, when the XML fragment starts to read in, the root state is pushed onto the Runtime stack as the initial state. When the first start tag a is read, Yfilter finds the initial state, matches the tag name and pushes the next state ID onto the Runtime stack, which is 2. At this point, the current active state is 2. As start tag b arrives, Yfilter finds b is in the current state. Therefore, the corresponding state 3 is added into the set of the target state because the state 3 is the accepting state for Query 1. Therefore, Query 1 is matched in this case because no predicate needs to be evaluated in this simple example.

In a real case, a predicate evaluation will start for Query 1 by backtracking the Runtime stack for each matched path for Query 1 and use the path to verify the predicates on each level of the query with corresponding start tags from the XML document. Because the "*" also exists in the current active state, the corresponding state 9 is added into the set of target state as well. Finally, the "ε" symbol exists at this point, so a recursive check has to be done. Following the transition, the b tag is present in state 6 and its corresponding state is 7. Because state 7 has neither "*" nor "ε", the recursion is finished at this point. The final target state will be 3, 9, 7 and 6 and they are pushed onto the Runtime stack. Because 6 is a "//-child" state, it is identified using underscore.

When start tag c is fired, the same process is executed for each of the current active states 3, 9, 7 and 6. From 3, the state 5 is added into the target state; from 9, the states 10 and 12 are added into the target state; from 7 and 6, the states 8 and 11 are added. Again state 6 is automatically added into the target states as a "//-child" state. Here we can see that state 5 is the accepting state for Queries 3 and 8; state 10 is the accepting state for Query 5; state 8 is the accepting state for Query 4; and state 11 is the accepting state for Query 6. The predicate evaluations should be processed for all these queries. At

this point, the set of target states is pushed onto the Runtime stack. When the end tags c, b and a are read, the top active states are popped out from the Runtime stack.

Here we can see that the Yfilter processes matching forward, in the sense that it reacts to the start tags following pre-order traversal and does matching whenever an accepting state is reached.

Post-processing is another feature of the filter. All the queries shown in Figures 2.6 and 2.7 are simple, without nested paths. To deal with queries with nested paths (complex queries), Yfilter decomposes a query into simple queries and matches them separately. For example, a complex query /a/*[c]/*/c will be split into /a/*/c (Q5) and /a/*/*/c (Q7). After NFA execution has been done for the entire document, post-processing starts to verify if Q5 and Q7 are both matched and the first "*" must be matched in the same place (same start tag) in the document.

In general, Yfilter utilizes the prefix commonality of user requests. A user request is added to the indexing tree from left to right in sequence, starting from the root. The current implementation of Yfilter can handle user requests with predication and a nested path in one level. A request with a nested path has at least one branch point. Each branch has a path attached to it. The attached paths themselves cannot have a branch point for one level nested path. Yfilter uses a post-procedure to handle nested path requests because the filtering algorithm can only process requests that are simple in nature. A request with a nested path is first decomposed into simple requests. These requests are indexed and their ownership to the original one is recorded. Then they are processed as normal requests. At the end of the matching process, the post-procedure verifies whether all the decomposed requests of a nested path request have been matched.

2.3.3 Afilter

In contrast to Yfilter, Afilter emphasizes suffix commonality to reduce mismatches. Both algorithms match queries that are simple in nature.

The objective of Afilter [29] is to overcome the disadvantage of Yfilter by constructing user requests as a directed graph (AxisView). The transition between two states is conditional for each request. It depends on the type of relationship between the corresponding element names for each state. In this way, the ancestor/descendant relationship can be treated as a self transition. The outgoing state is simply a waiting state. Thus the increment in the number of active states is not significant.

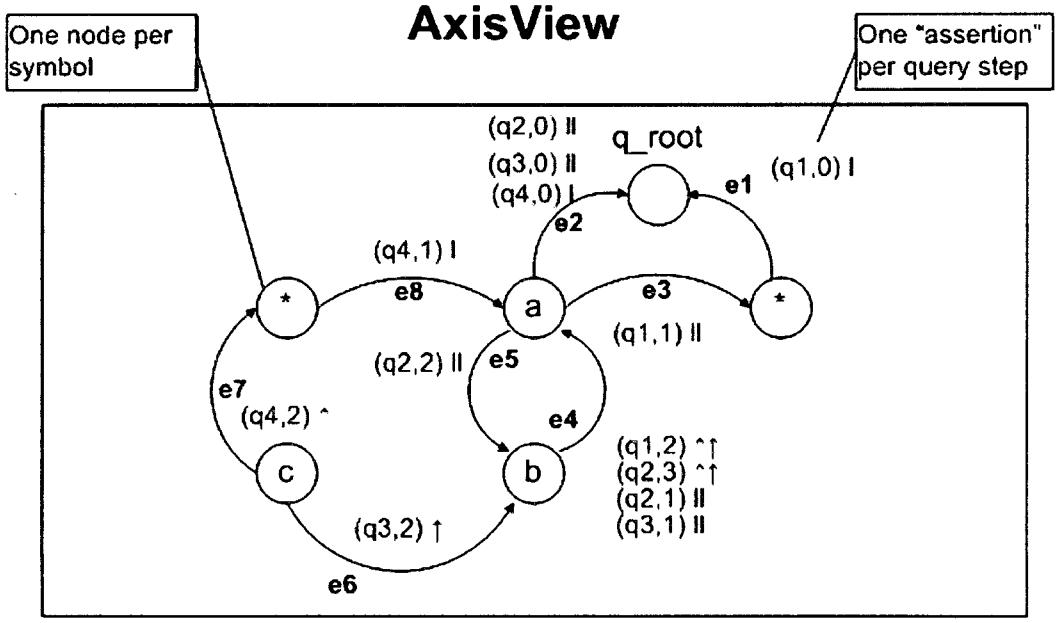


Figure 2.8 AxisView of Afilter [29]

Afilter uses a triggering mechanism to delay the matching process until a trigger condition is met. Each node in the directed graph corresponds to a label and each edge corresponds to a set of axis tests. Each edge is annotated with a set of axis assertions for verification. If an edge between a and b has a parent/child relationship for query q, an

assertion (q, s)| will be added for this edge. If b is the last label of q, the assertion will be (q, s)↑. If the relationship is ancestor/descendant, the assertion is (q, s)|| if the label b is not the last one; otherwise, the assertion is (q, s)↑↑. The two symbols ↑ and ↑↑ indicate that the assertion is a trigger assertion because the corresponding label is the last one for the query. The s in the brackets represents the step index of the query, which starts from 0.

Figure 2.8 shows the directed graph for queries q1 = //d//a//b, q2 = //a//b//a//b, q3 = //a//b/c, q4 = /a/*/c. For example, q1's last label is b and it has an ancestor a. Therefore, edge e4 has a trigger assertion (q1, 2)↑↑.

Unlike Yfilter, which uses a stack to store matched active states, Afilter uses a set of named stacks to store matched states. The states are called objects because they also maintain pointers that allow tracing of previous matching states. Therefore, the state stacks actually store the matching paths. Afilter implements these data structures to backtrack the matching path whenever an end of request is triggered during the process of matching.

Figure 2.9 shows how Afilter works for the example in Figure 2.8. Figure 2.9(a) shows that an empty Stack Branch has stacks corresponding to each node in the AxisView; the status of the Stack Branch after <a><d><a><b> is shown in Figure 2.9 (b).

When a start tag of a XML document is read, a named state is pushed onto the stack with the same name as well as the "*" stack, because the "*" stack matches all of them. At the same time, the pointers are created to point at the topmost state object of its

parents' stack. The pointers are used to backtrack the path element when a trigger assertion occurs at the end of a query.

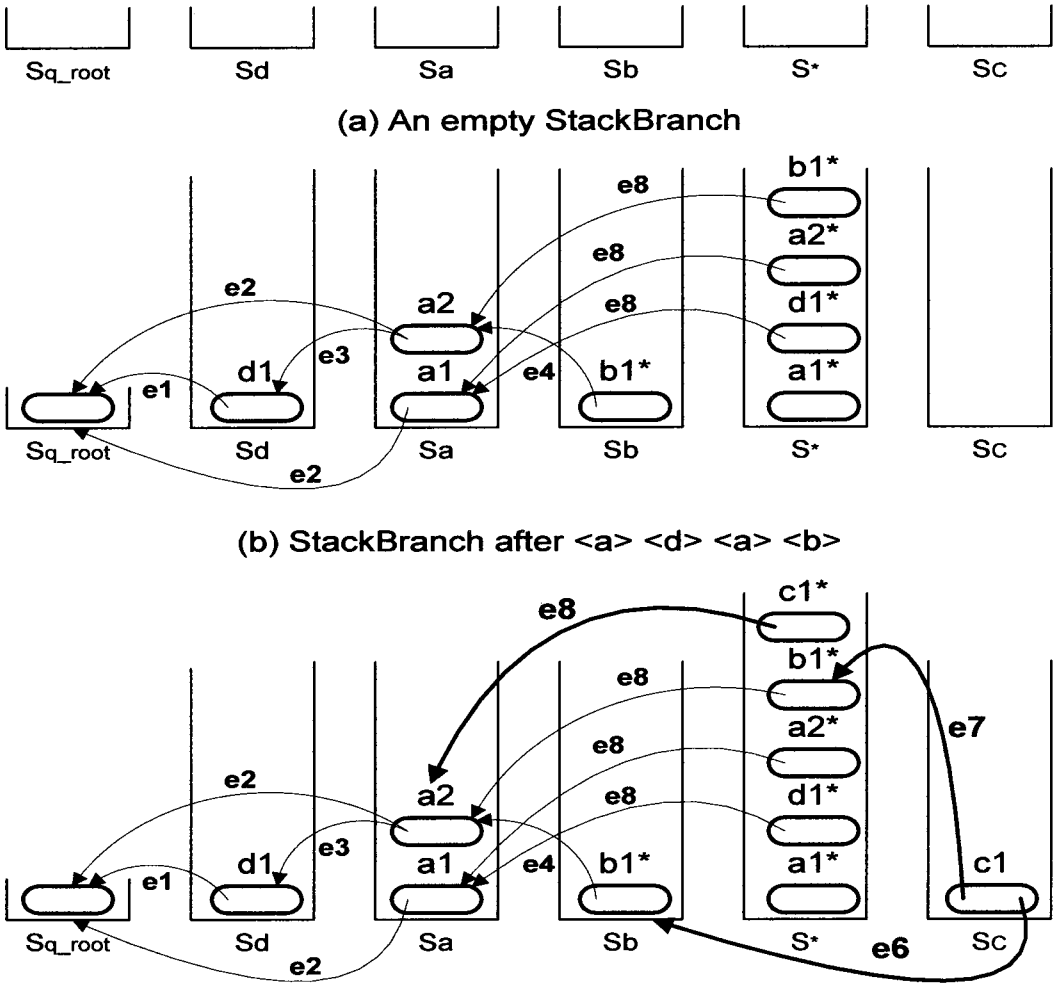


Figure 2.9 Runtime Description of Afilter [29]

For instance, when the first a is read, a1 and a1* are pushed onto both Sa and S*; Figure 2.9 (c) shows the status after tag c is read from the previous status in Figure 2.9 (b). First the c1 is pushed onto Sc. This new stack object has outgoing pointers e6 and e7, corresponding to the edges in the AxisView (Figure 2.8). The two pointers point to stacks

Sb and S*, respectively. Then c1* is pushed onto S* and e8 is created to point at the top of Sa. Because c1 raises a trigger assertion (q3, 2)↑ and (q4, 2)↑, the matching process starts for q3 and q4 by backtracking the pointers along the path until the root. In the case of q3, matching starts from c1 to b1 in stack Sb following e6, then arrives at a2 following e4 and finally reaches the root through e2. Therefore, the path a2 – b1 – c1 is evaluated for q3 in a backward manner. As soon as an end tag of an element is read, the corresponding stack object must be popped and deleted from the Stack Branch.

Optionally, Afilter uses PRLabel-tree [29] to leverage prefix sharing by caching path expressions based on the commonalities in their prefixes, and uses SFLabel-tree [29] to enhance suffix sharing by caching path expressions based on their overlapping suffixes. Afilter does not explicitly deal with requests with nested paths at this point.

2.3.4 Hybrid Algorithm

The hybrid algorithm [7] enhances performance from the viewpoint of engineering. It is intended to speed up the filtering process by taking advantage of both Yfilter and Afilter. As mentioned above, Yfilter emphasizes the prefix commonality of user requests, but suffers overhead in terms of ancestor/descendant relationship '//'and wildcard '*'. On the other hand, Afilter is not very sensitive with regard to the type of relationship and wildcard. The hybrid algorithm pre-processes the user requests and separates them into two groups. One group contains the so-called simple requests and the other contains the complex requests.

Neither Afilter nor the hybrid algorithm explicitly deals with nested paths. A simple request here means that the request has no wildcard or ancestor/descendant relationship; otherwise, it is a complex request. The complex requests are sent to Afilter

to take full advantage of it. The simple requests are sent to a simplified Yfilter. Because the simple requests have no '//' and '*', there is no 'ε' state and a matching process follows a single path in the NDA tree. Therefore, a NDA tree is not necessary; instead, a normal tree with multiple matching elements stored in each node will be sufficient. This structure is a Deterministic Finite Automata, which also realizes prefix commonality, like Yfilter. In the implementation, the hybrid algorithm creates only the core Afilter algorithm, without additional techniques like prefix tree and suffix tree. It uses compact runtime stacks to store the matching states instead of the set of named stacks used in Afilter, which is believed to have better memory storage.

The performance of the hybrid algorithm depends on the percentage of simple requests in the total user requests. In one extreme case, if all requests are simple, they will be processed only within the DFA tree. The performance will be a little better than Yfilter because of the overhead of manipulating a complex NFA tree. In another case, if all requests are complex, the hybrid algorithm is the same as Afilter, except for the implementation modification. However, the cost of switching the two underlying structures is unknown. Moreover, the cost of expanding its capability to handle a nested path is not predictable. This is because the decomposed requests may be either simple or complex. Some of them may be sent to Afilter and others may be sent to a simplified Yfilter; therefore, verifying the post-processing may be costly.

2.3.5 Gfilter

Recently, Chen et al. proposed a novel XML message filtering algorithm called Gfilter [14]. Gfilter fully inherits its previous work, Afilter, which focuses on optimizing the path matching performance via a bottom-up approach, benefiting from the heuristics

that the probability of mismatching at the leaf is higher than at higher levels. This can be seen in Figure 2.10. The top-down approach (Yfilter) processes matching in pre-order traversal, using the start tag of each element from the XML document; the bottom-up approach (Gfilter) executes matching during post-order traversal using the end tag of each element.

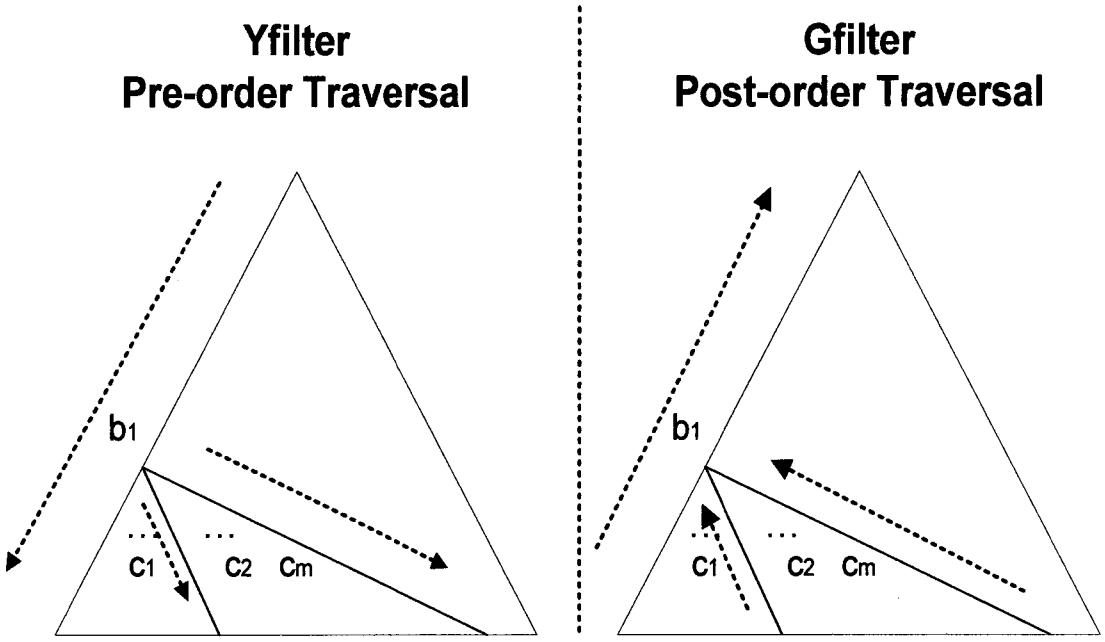


Figure 2.10 Top-down and Bottom-up Matching Order [14]

Moreover, Gfilter further improves the post-processing performance using the Tree-of-Path (TOP) encoding scheme to compactly represent the path matches for the XML document. TOP encoding scheme is derived from the previous work of Gfilter [13]. It dynamically maintains two tables, PCTable and ADTable, to hold the current satisfied suffix in each step of bottom-up matching. An example is provided to explain how it works next.

In Gfilter, the query is represented by the Generalized-Tree-Pattern (GTP). GTP was introduced by Chen el al. [15]. It is a structure to represent XQuery for carrying out tree pattern matching against XML documents. For example, the GTP in Figure 2.11 represents XQuery:

```
FOR $b in //A//B[//C],
    $d in $b/d
LET $c = $b//c
RETURN $b,$d,$c
```

In this XQuery, $b, $d and $c are three variables. In the FOR clause, $b represents all possible matches in the expression "//A//B[//C]". Node B is the binding node. $d gives a further constraint that any match for $b must have node d as a child of the binding node of $b. The last node d is the binding node for $d. The LET clause represents a group binding of $b and $d where the node is *c*. Here, this query returns a set of matching nodes for the XML document on the left side of Figure 2.11. The matching set would be {b1, d2, c1}, {b1, d2, c2}, {b2, d1, c1} and {b2, d1, c2}. Although GPT has a different expression of the query with XPath, the matching process is the same. The equivalent XPath expression for the query in the figure is //A//B[/D]//C.

In Figure 2.11, query //A//B[/D]//C will first be decomposed into sub-queries //A//B//C and B/D. The decomposition results are simple queries so that TOP encoding can apply. The decomposition also allows so-called suffix sharing. Because Gfilter processes a match from the bottom to the top, suffix sharing can reduce its number of active states during matching. This is the same idea as with Yfilter. The structure of the TOP encoding of queries forms a bunch of branches, and the start matching node is the root of each branch.

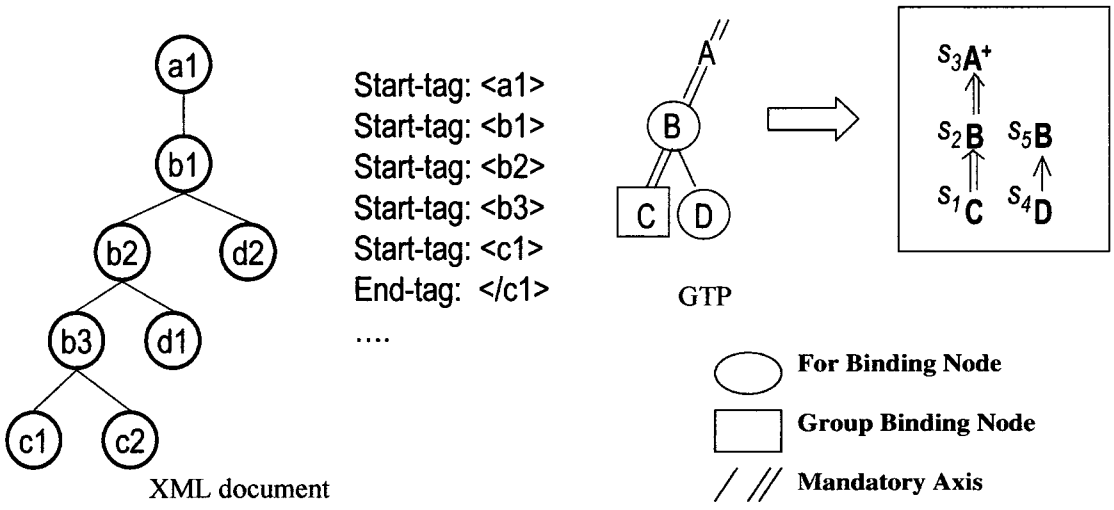


Figure 2.11 An Example of XML Document and GTP [14]

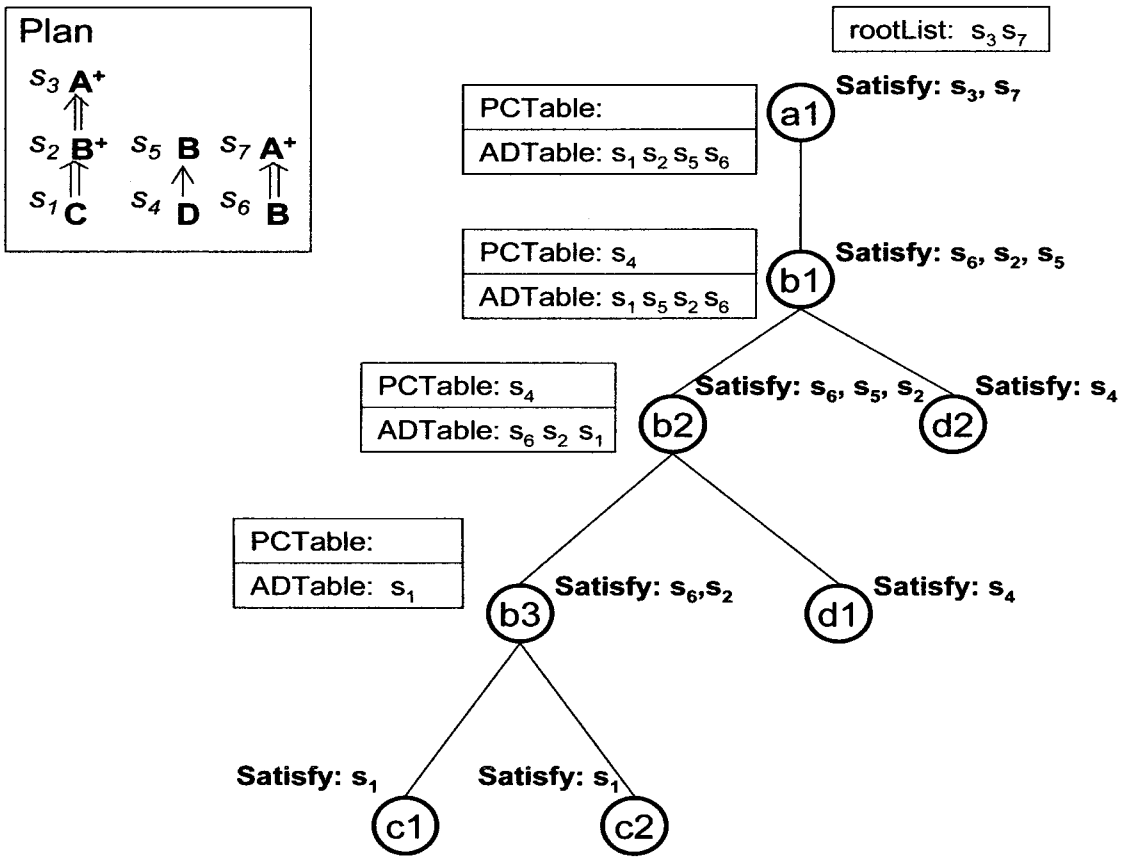


Figure 2.12 The TOP Encoding [14]

Figure 2.12 shows TOP encoding and how it works in the matching process. In this figure, the XML document tree is the same as that in Figure 2.11. The plan in the top left corner is constructed from some GTP queries before the matching process starts. During each step, the PCTable and the ADTable are created to hold the current satisfied suffix nodes. The PCTable holds the children and the ADTable holds the descendants. When the end tag of document node b3 arrives, the ADTable has satisfied suffix node s1 and the PCTable is empty because no descendant suffix node is satisfied. When the end tag of b2 is reached, the PCTable has satisfied suffix node s4. Two satisfied suffix nodes, s6 and s2, are added to the ADTable and the s1 is automatically appended into the ADTable from the lower node. This process continues until the end tag of the root of the XML document is read. At the root node, the satisfied nodes are s3 and s7, which indicates the queries rooted at these nodes have been matched. The plus sign with a suffix node means that it can hold the roots of some queries. Suffix node s2 with label B has a plus sign. Whenever a suffix node with a plus sign is reached during the bottom-up process, matching is started based on the content in the PCTable and the ADTable. For the complex query in Figure 2.11, when its root s3 with label A is satisfied, the post-processing of matching starts to verify that the root of separate query B/D, namely s3, is in the ADTable. Otherwise the query is not matched.

Gfilter can perform better than Yfilter when a XML document has repetitive nodes because of Gfilter's efficient backward matching process. For the example in Figure 2.11, the XML document has three continuous nodes named b1, b2 and b3. Gfilter evaluates these three nodes only once in the post-processing for the example query //A//B[/D]//C. But YFilter has to evaluate theses nodes three times during the forward

matching. This is due to the full path matching performed by Yfilter. If the data is recursive (e.g., the document in Figure 2.11 has three b nodes), Yfilter has a potentially exponential number of path matches [29]. However, Gfilter postpones matching until the root of a query is reached.

The next chapter will describe Bfilter, which is similar to Gfilter. Bfilter also leverages a bottom-up approach to enhance performance. The difference is that Bfilter emphasizes branch point matching between the XML document and complex queries to reduce mismatching. For the query and XML document in Figure 2.11, when the end tag of b3 arrives, Bfilter will set the flag as false for the query because the sub-tree of its branch b is not satisfied. When the end tag of b2 is read, Gfilter detects that branch point b2 matches the branch point B in the query; the flag will be set as true after the sub-tree is evaluated, because a child element d1 and descendant c1 and c2 exist. When the end tag of b1 is read, matching will not take place because the B branch of the query is already satisfied. This is the same as with Gfilter in terms of delaying the matching process until it is necessary. However, if the branch point does not match, b is not a branch, or its sub-tree has no required elements, the matching process will never take place in Bfilter.

Chapter 3: Bfilter

Bfilter is a novel XML message filtering algorithm for XML message filtering and matching. Bfilter effectively deals with user profiles with complex queries by leveraging branch points in both XML document and user profile. It uses backward matching branch points to evaluate user profiles. A matching process is delayed until branch points in both the XML document and user profile match. Thus, XML message filtering can be performed more efficiently as the probability of mismatching is reduced.

Yfilter, one of the most popular filtering algorithms, uses NFA to index XPath queries. It utilizes the prefix commonality of queries for greater efficiency. Contrary to Yfilter, Afilter emphasizes suffix commonality to reduce mismatches. Both algorithms match queries that are simple in nature. To deal with queries with nested paths, Yfilter decomposes a query into simple queries and matches them separately. Post-processing is then needed to verify whether the whole query has been matched. Derived from Afilter, Gfilter also matches queries backwards to emphasize suffix commonality. But it still needs post-processing to group separated sub-queries for complex queries. Bfilter not only matches queries backwards, but also matches branch points backwards. The match of branch points is preconditioned for further matching of a complex query. Since Bfilter treats a complex query as a whole, no post-processing is needed for complex queries.

3.1 Overview of the Bfilter Technique

As mentioned earlier, Bfilter is intended to deal with complex user queries that have nested paths. In Bfilter, a complex request is treated as a unit without being

decomposed, and no post-processing is needed. It uses backward matching of branch points to reduce the probability of mismatching.

Bfilter realizes the tree structure of XML documents and user requests with nested paths (see Figure 3.1). A tree is composed of three parts: branch points that include the root; branches connecting two branch points; and branches that have only one end attached at a branch point while the other end is free. A branch that connects two branch points is known as a Transit Branch; other branches are called Tangling Branches.

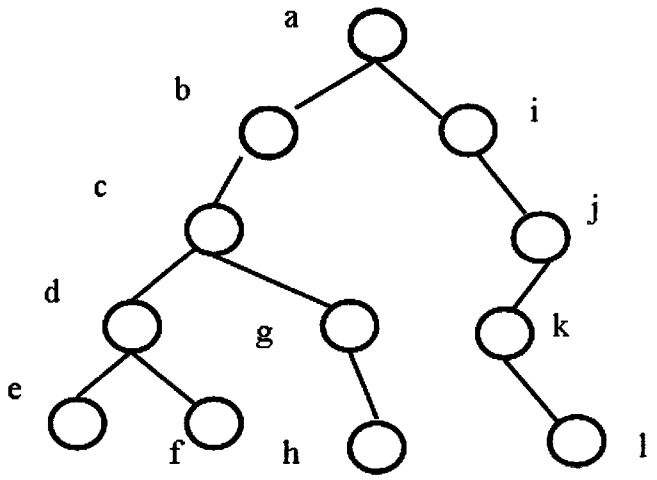


Figure 3.1 A Tree Structure

Branch Points: a, c and d
Transit Branches: a/b/c and c/d
Tangling Branches: d/e, d/f, c/g/h and a/i/j/k/l

In Figure 3.1, there are two transit branches and four tangling branches. The two transit branches are a/b/c and c/d. The transit branch a/b/c connects branch point a and c. The transit branch c/d connects branch point c and d. The four tangling branches are d/e, d/f, c/g/h and a/i/j/k/l. If a request matches a document, the branch points in the request must match the corresponding branch points in the document.

Bfilter matches queries from the end to the front of a request. The matching process starts only when both a document branch point and a request branch point have been matched. Only when the branch point is matched will its tangling branches and the transit branches (if they exist) are checked. In other words, the matching of branches is delayed until the branch point they attach to is matched.

Given any query with nested path Q and a XML document, let the probability of matching of Q be P (Q). Because Yfilter matches Q from the front to the end of Q, we have:

$$P(Q) = P(\text{front}) \times P(\text{rest} \mid \text{front}),$$

where P(front) is the probability of matching of the front part of Q and P (rest | front) is the probability of matching of the rest of Q when the front is matched.

Similarly, Afilter matches Q from the end to the front of Q:

$$P(Q) = P(\text{end}) \times P(\text{rest} \mid \text{end}),$$

where P (end) is the probability of matching of the end part of Q, and P (rest | end) is the probability of matching of the rest of Q when the end is matched.

Because Bfilter matches Q from the last branch point to the front of Q, we have

$$P(Q) = P(\text{last branch}) \times P(\text{rest} \mid \text{last branch}),$$

where P (last branch) is the probability of matching of the last branch point of Q and P (rest | last branch) is the probability of matching of the rest of Q when the last branch is matched.

In real messages, a tag name has a higher probability of appearance at a higher level than at a lower level in a request. In other words, the probability of a match of the front part of a request is higher than that of its end part, which indicates that a match for

the front part of a request may not lead to successful match for the end part. This observation motivates a backward matching (bottom-up) approach used in XML filters including Afilter, Gfilter as well as Bfilter. It implies that, for a particular query, if the match of its last element(s) is successful during backward matching, the probability of a match of its remaining part is higher than the case in which the match of its front element(s) is successful during forward matching. This heuristic can be explained through the following example, in which a user is requesting information about the Mechanical and Aerospace Engineering department of Carleton University in the domain of Canadian secondary schools: Canada/University/Carleton/ Mechanical and Aerospace Engineering. An example of a XML document of Canadian Post-Secondary Schools is shown below:

```
<Canada>
    <University>
        <Carleton>
            < Mechanical and Aerospace Engineering/>
        </Carleton>
        <Toronto>
            <Mechanical Engineering/>
        </Toronto>
    </University>
    <College>
        <Algonquin>
            <ESL Language School/>
        </Algonquin>
    </College>
</Canada>
```

Because the top level tag of the message "Canada" will appear in all messages, the probability of a match of the first element of the request is 100%. The second level tag "University", is likely to appear in about half of the messages in the domain because, unlike in the example, where the message includes both "University" and "College" at

the second level, some messages may contain only "University" or "College". So the probability of a match of the second element of the request is about 50%. As the third tag of the message, "Carleton" will appear in fewer messages than the tag "University". Thus the probability of a match of the third element of the request will be lower than the element "University". Similarly the probability of a match of the fourth element "Mechanical and Aerospace Engineering" will be lower than that of the element "Carleton". Thus, the probability of a match of an element at the higher level is usually higher than that at the lower level. So P (front) is greater than P (end). Furthermore,

P (last branch) = P (branch point matched and it is a branch point in XML document)

= P (branch point matched) × P (it is a branch point in XML document).

Note that P (branch point matched) is approximately the same as P (end) because the branch point is the last one which is close to the end element of the request. On the other hand, the number of branch points in any XML document is less than half of the total number of tags. This is because the number of branch points is at most half of the total number of nodes in a tree. The worst case is when the tree has only two nodes, one is the root and one is the leaf. In this case, the number of branch points is 1 and the total number of nodes in the tree is 2. We assume the worst case that the number of branch points is the half of the total number of nodes, so P (it is a branch point in XML document) is 0.5. Thus P (last branch) will be less than P (end). Therefore,

P (rest | last branch) > P (rest | end) > P (rest | front).

The three matching approaches, branch point backward matching (Bfilter), backward matching (Afilter and Gfilter) and forward matching (Yfilter), start matching from the last branch point, the last element and the first element of a request respectively.

After the first matching step succeeds, the probabilities of matching for the three approaches correspond to P (rest | last branch), P (rest | end) and P (rest | front) respectively. Mismatching is a scenario where the matching fails after the first step(s) succeeds. The probabilities of mismatching for the three approaches are given by 1 - P (rest | last branch), 1 - P (rest | end) and 1 - P (rest | front) respectively. Thus, the probability of mismatching of branch point backward matching is less than that in backward matching, and the probability of mismatching of backward matching is less than that in forward matching. In other words, the probability that mismatching will occur in Bfilter is expected to be the lowest of the three. Mismatching causes the filter to spend time on evaluating requests that will ultimately fail. The lower the probability of mismatching, the greater the likelihood that the processing of un-matched requests will be stopped earlier in the matching process. Reducing the mismatching probability is supposed to increase efficiency of filtering. In order to compare the three algorithms, Appendix A provides a case study that demonstrates the matching processes executed in Yfilter, Afilter and Bfilter.

3.2 Design of Bfilter

3.2.1 System design overview

Bfilter is a complex filtering system built on top of some Yfilter components. The high-level architecture of Bfilter is illustrated in Figure 3.2. When the user queries are read in, the XPath Parser is responsible for parsing them into query objects and sending these queries to the Query Index Tree. Query Index Tree stores these queries in a data structure that uses a hash table-based approach to implement NFA.

SAX XML Parser is event-based XML parser [27]. When parsing a XML document, the "Start document" and "end document" events are generated at the beginning and the end of the document respectively. A "start element" event is generated when a start tag is read. This event contains the tag's name and attributes. An "end element" event is generated when an end tag is read. This event is related to the "start element" event generated by the start tag corresponding to this end tag. It contains the name of the corresponding start tag and closes the segment marked by the start and end tag pair in the document. A "characters" event is generated when content between a pair of tags is read. This event carries the content as a string.

The event-based parser is used by all the filtering systems discussed so far in this thesis. The challenge of using this type of parser is that the XML document reads tag by tag using a pre-order traversal. The filtering system which uses the event-based parser can not trace back to the previously parsed part of the incoming document. However, the advantage of using this type of parser is that the matching process can start at the same time the document is being read. Moreover the document does not need to be stored and parsed in the filtering system, which is critical for large documents. Bfilter uses a SAX event-based parser by implementing event handlers corresponding to SAX events.

As discussed earlier, Bfilter leverages branch points in both the XML document and the user profile. It uses backward matching branch points to evaluate user profiles. The XML document *Branch Point Detector* is responsible for reporting a branch point when a document is read, by using the algorithm described in the next section. When a branch point occurs in a document, it is pushed into the *Document Branch Stack.* The *Queries Branch Points Stack* holds the branch points of queries in the form of a *Query*

Index Tree. The *Query Index Tree* has a Runtime Stack to store the states of NFA. During the matching process, the *Query Index Tree* will push the current active states for all queries onto the Run Time Stack. This is the same as Yfilter, which can be seen in Section 3 of Chapter 2. The difference is that the *Query Index Tree* also stores the branch point information of all queries in the hash table-based structure, so the queries' branch points can be identified from the current active states in the hash table-based structure. The branch points found will be pushed onto the *Queries Branch Points Stack.*

The identification of branch points in a *Query Index Tree* is processed when a start tag is read. The identification of branch points in a XML document is also processed when a start tag is read; so both of these processing are within the handler of a "start element" event. Bfilter's matching algorithm matches the branch points in the two stacks when an end tag is read. It is triggered by the "end element" event.

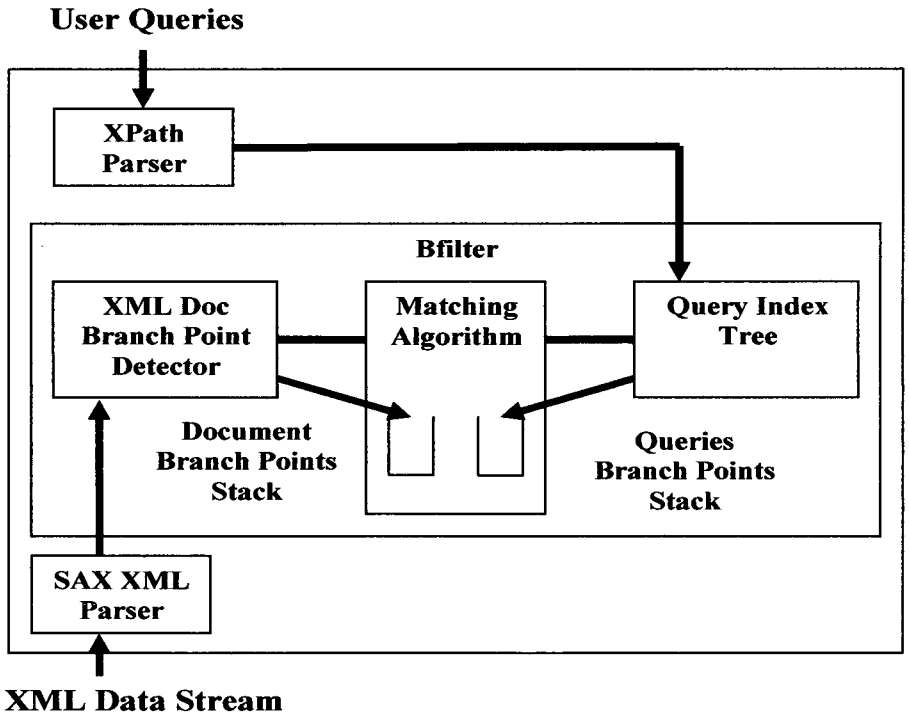


Figure 3.2 Architecture of Bfilter

In Figure 3.2, the *SAX XML Parser* is a component reused from Yfilter. The *XPath Parser* is derived from Yfilter. In Yfilter, a complex query is decomposed into simple queries. The *XPath Parser* reconstructs the decomposed queries into one unit so that the *Query Index Tree* can index them as a whole. The *Query Index Tree* is derived from Yfilter's Query Index by providing additional functionalities to index complex queries and store branch point information. The main functionalities of Bfilter that differ from Yfilter are listed and described below.

Handle the stacks:

As each tag is read in, it is pushed onto (start tag) or popped from (end tag) the runTimeBranch Stack. The elements in the runTimeBranch Stack are used to find branches of the document and match the tangling branches. As each branch point is detected in the document or the queries index, it is pushed onto the corresponding documentBranches Stack or the queryBranches Stack. When an element is popped out from the runTimeBranch Stack (the end tag of the element is read in), if a branch point that is associated with this element exists in the two stacks, the branch point is popped out from them, and the matching is performed for this branch point.

Detect branch points in document:

If a sequence of end tags follows a start tag that is at the same level as the previous one, the current top element on the runTimeBranch Stack is a branch point. In Figure 3.3, the example document stream is a/b/c/d/d/c/e/e/b/f/g/g/f/a. A tag name with an underscore means that the tag is an end tag. When a start tag is read in, a node representing this tag is pushed onto the runtime stack. When an end tag is read, the corresponding node is popped out from the runtime stack. At the bottom of the figure, it

shows the status changes of the runtime stack from left to right. When the first tag, *a*, is read in, node *a* is pushed onto the runtime stack. Now the runtime stack has one node – *a*. When tag *b* is read, node *b* is pushed onto the top of the stack. After tags *c* and *d* are read in and node *c* and *d* are pushed onto the stack, the end tag of *d* is read in, so the *d* node is popped out from the stack. Then node *c* is popped out when end tag *c* is read in. When start tag *e* is read in, the branch point *b* is detected because *e* immediately follows a pop-out of *c*. In the same way, the root *a* is detected when start tag *f* is read in after node *b* is popped out from the stack.

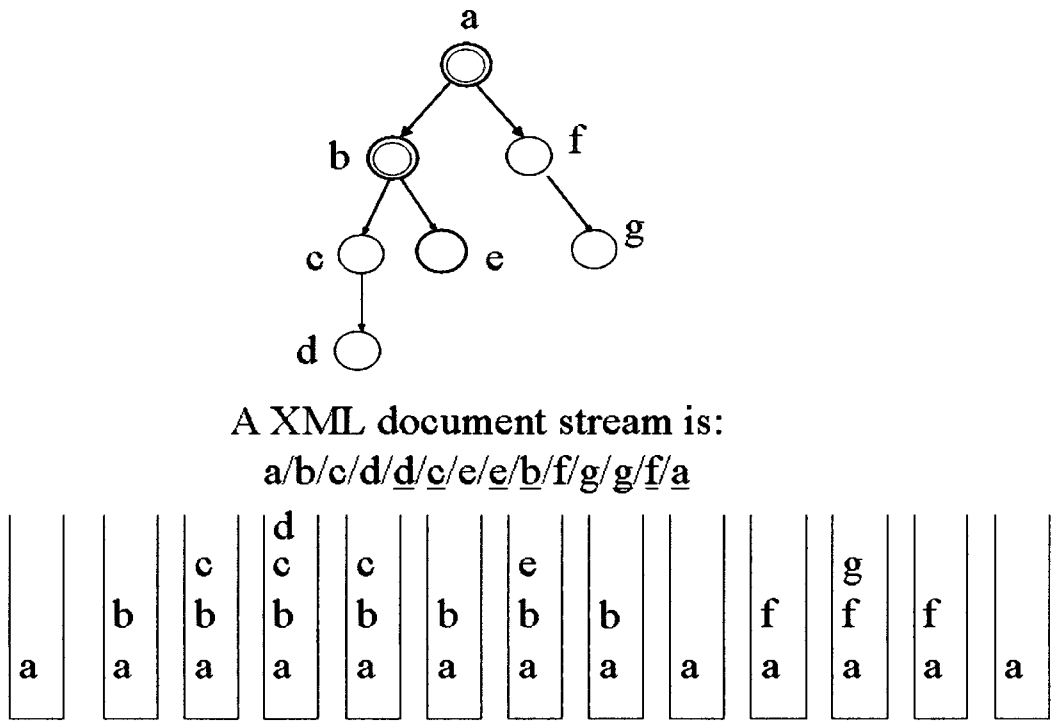


Figure 3.3 Example of Branch Point Detection

Process branch point matching:

Bfilter treats a complex query as a whole during the matching process. In Bfilter, a complex query is represented by using sub-queries that are separated by branch points of the complex query. Each sub-query is rooted at a branch point and has one or more

paths. A sub-query has only one branch point which is its root. So a complex query is represented as a linked list of sub-queries in which the first sub-query rooted at the root of the complex query is at the head of the list. For instance, if a complex query, Q, has three branch points, namely the root, branch point 1 and branch point 2, it will be represented by sub-query 0, sub-query 1 and sub-query 2 rooted at the root, branch point 1 and branch point 2, respectively. Notice that a simple query is represented as one sub-query that has only one path. In Bfilter, when we say a query, it usually indicates a sub-query not a complex query. The detailed description of a query's representation in Bfilter is provided in the following section.

When a branch point is popped out from the documentBranches Stack and the queryBranches Stack, if the branch point exists in both stacks, it means that this branch point is a match between the document and the queries that have this branch point. Therefore the matching is performed for these queries. Otherwise, there are two possibilities: the branch point exists only in the documentBranches Stack or in the queryBranches Stack. In the first scenario, the element popped out is not a branch point in the queries; therefore nothing needs to be done. In the second case, the queries that have this branch point as root are simply marked as unmatched and no further matching is needed. When processing a match for a query that has this matched branch point, if its descendants are not matched, the query is marked as unmatched. Otherwise, the matching for all its branches will be performed.

As discussed earlier, Bfilter processes matching for a query only when its branch point finds a match in the document. The order of branch point matching is conducted from the lower level to the higher level, but the determination of a match is always made

at the higher level. Whenever a match is determined for a query, no matter whether the result is matched or not, all candidates of the query match are deleted. If the result is unmatched, all the descendants of the query are marked as unmatched.

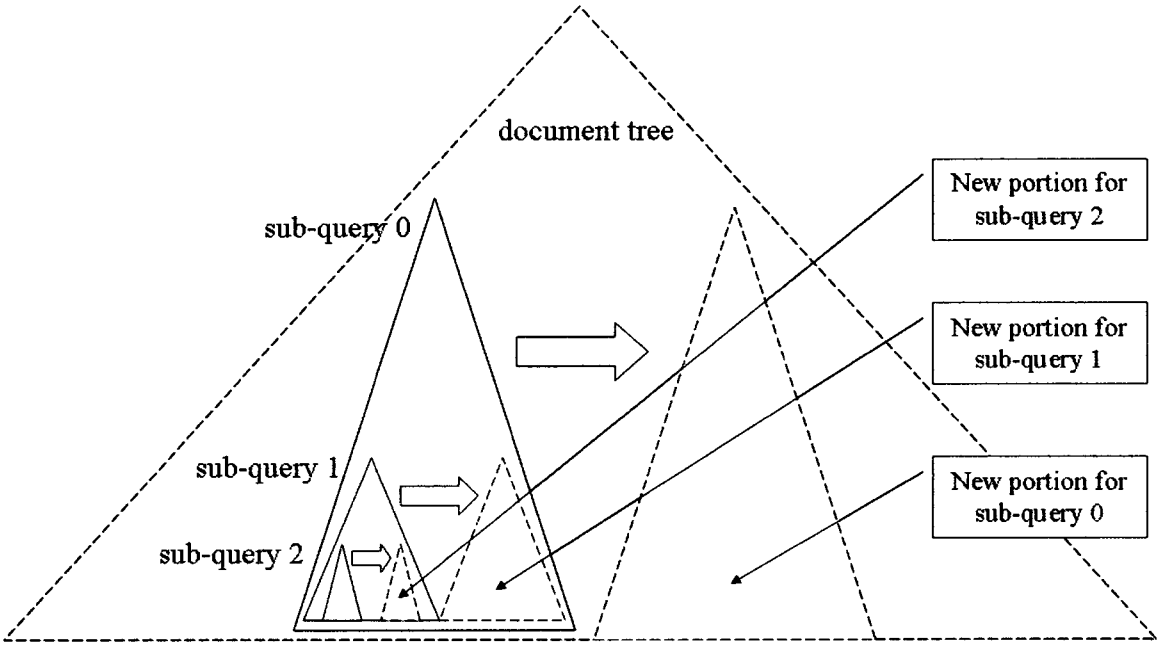


Figure 3.4 Example of Branch Point Matching

Figure 3.4 illustrates the branch point matching in Bfilter. As discussed earlier, Yfilter decomposes a complex query into simple queries and processes matching for these separated queries independently. Post-processing is needed to verify if the queries are matched in the same place in the document. The match candidates are retained until the document filtering is completed. In contrast to Yfilter, Bfilter cleans up all candidates of the current queries as well as those of their descendants when the corresponding branch point in the document is read over. When a new portion of the document arrives, if a corresponding query (the query that matches the root of the new portion) and its descendants are already matched, it does not do any thing. Otherwise, a new matching process will start for the query.

In Figure 3.4, the solid triangles represent a complex query composed of sub-query 0, 1 and 2. In the document tree, the candidate match portion for sub-query 0 contains the candidate portions for sub-query 1 and the candidate match portion for sub-query 1 contains the candidate portions for sub-query 2. If a new portion corresponding to sub-query 2 is read in (the root of the new portion matches the root of sub-query 2), sub-query 2 will do nothing if it is already matched. Otherwise, it will clean up all its match candidates and reprocess matching in the new portion. The same is true for sub-query 1 and sub-query 0, except that in these cases they will also need to clean up their descendants. In fact, Bfilter only keeps match candidates in the current portion (branch) of the XML document. This feature enables Bfilter to filter a document portion by portion.

Current filtering algorithms match user queries throughout the entire document base. For example, Yfilter has no easy way to obtain a matched portion for a complex query because it does not know where the separated queries match in the document until the end of the document is read and the post-processing is completed. It is difficult to save all relevant information if the document is very large. Bfilter, however, can provide matched portions of a document for a particular query based on different branch point levels. For instance, when sub-query 2 is matched, the corresponding portion can be saved. If its parent fails in matching, the portion will be deleted. At the end of the document, if sub-query 0 is matched, the matched portion for sub-query 2 is right there in the stack and ready to be used. Thus, if Bfilter is applied to the Publish/Subscribe systems, the upstream filters can deliver the correct portions of a document to the downstream filters.

3.2.2 System prototype

In the current implementation, Bfilter is built on top of Yfilter. There are two reasons for doing this. The first is to focus on concept demonstration and the algorithm itself without considering other aspects, such as XPath query parsing and query predicate matching; the second is that the implementation can reuse many existing classes from Yfilter due to the similarities between these two filters.

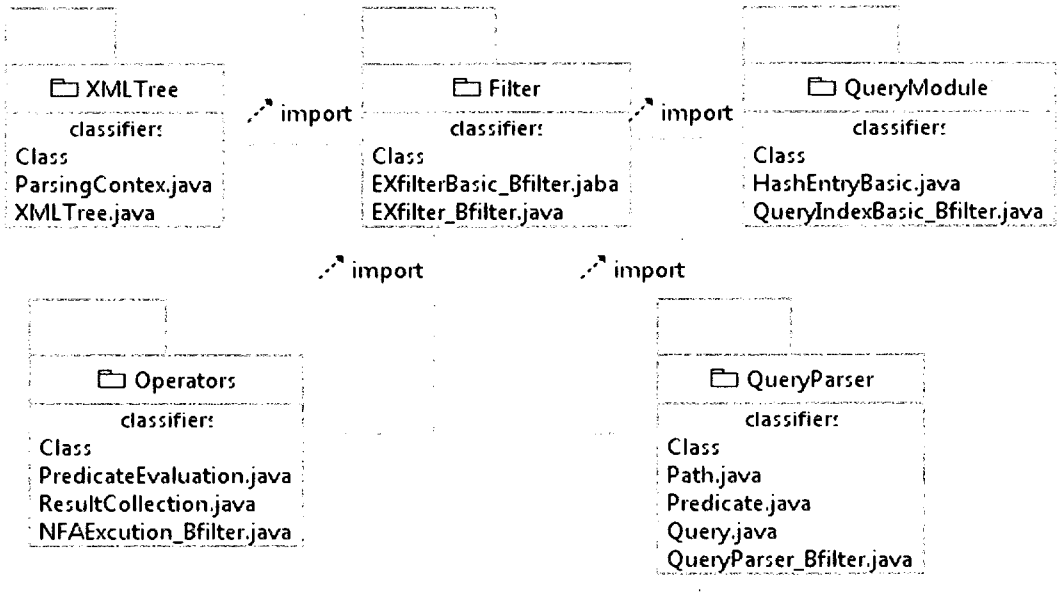


Figure 3.5 Packages of Bfilter

Figure 3.5 shows the packages of Bfilter that are derived from Yfilter. Each package of Bfilter is discussed in more detail.

QueryParser Package:

This package implements the XPath Parser component. The QueryParser_Bfilter class is the query parser that parses queries as required. This package is built on top of the Yfilter package in the same name. It changes path and query representation to meet the requirement of Bfilter. As discussed earlier, Bfilter treats a complex query as a whole

during the matching process. Figure 3.6 shows the representation of a query in Bfilter. Bfilter represents a complex query using sub-queries that are separated by branch points. Each sub-query is rooted at a branch point and has one or more paths. A sub-query is named as Q{m,n}, where *m* is the index of the complex query the sub-query belongs to, and *n* is the index of the sub-query which starts from 0. A sub-query has two pointers, one pointing to its parent and another to its child. If a sub-query is the first in a complex query (index equals zero), its parent is null; if a sub-query is the last one, its child is null.

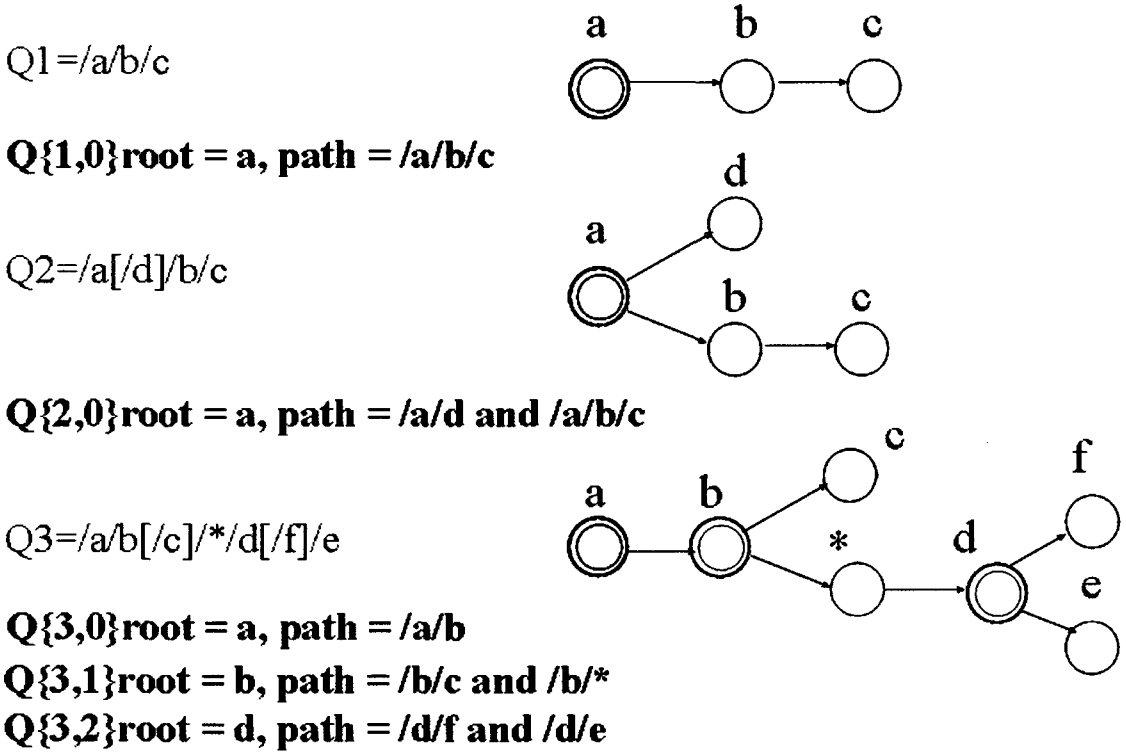


Figure 3.6 Example Query in Bfilter

In Figure 3.6, Q1 is a simple query; its root is *a* and it has a single path /a/b/c. Q2 is a complex query that has one branch point (the root), and it has root *a* and two paths /a/d and /a/b/c. Q3 is a complex query with three branch points, and is represented by three sub-queries, Q{3,0}, Q{3.1} and Q{3,2}.

QueryModule Package:

This package contains QueryIndexBasic_Bfilter class, which implements a hash table-based data structure to index queries. This data structure is called queryIndex Tree. In contrast to the QueryIndexBasic class of Yfilter, QueryIndexBasic_Bfilter is created with new structures to hold the queries together as a whole. It is capable of determining if the current active state is also a branch point for some other queries. The algorithm for building the queryIndex Tree is presented below:

Build QueryIndex Tree Algorithm

```
Input: XpathQuery q;
Init: Hashtable-Based QueryIndex Tree = Empty;

    Start State = root of QueryIndex Tree;
    While q is not null
        //index each sub-query one by one
        Indexing starts from Start State;
        Mark the state reached by the first element of q as branch point of q;
        For each tangling branch b in q
            Index b in QueryIndex Tree;
            Mark the state reached by the last element of b as accepting state of
            b;
        End for;
        If q is not last sub-query
            Index transit branch c in QueryIndex Tree;
            Mark the state reached by the last element of c as accepting state of
            c;
            Start State = the state reached by the last element of c;
        End if;
        q = NextSubQuery();
    End while;
End Algorithm;
```

This algorithm outlines the steps required to index queries. Based on the query representation discussed before, a query is composed of sub-queries. If a query is simple, it is represented as one sub-query.

In Figure 3.7, Query 1 is simple, and is therefore represented as one sub-query: Q{1, 0}. The indexing starts from its root element *a*. This element reaches state 2 in the tree. State 2 is then marked as a branch point of Query 1, in particular, the root of Q{1,0}. Q{1,0} has only one path. After the last element of the path reaches state 5, this state is marked as Q{1,0,p0}, which is the accepting state for path 0 of Q{1,0}.

Q1=/a/b/e
Q{1,0}root = a
p0 = /a/b/e
Q2=/a/b[/c]/e
Q{2,0}root = a
p0 = /a/b
Q{2,1}root = b
p0 = /b/e; p1 = /b/c
Q3=/a//c[/e]/d
Q{3,0}root = a
p0 = /a//c
Q{3,1}root = c
p0 = /c/d; p1 = /c/e
Q4=/a/*[/b]/c
Q{4,0}root = a
p0 = /a/*
Q{4,1}root = *
p0 = /*/c; p1 = /*/b

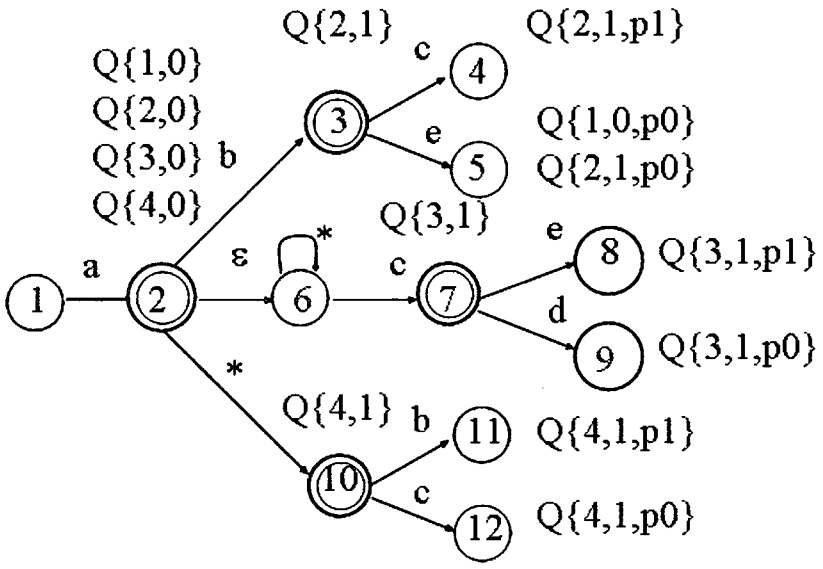


Figure 3.7 Example of Query Indexing in Bfilter

Similarly, Query 2 is represented as one sub-query Q{2,0} with two branches. State 2 is marked as the root of Q{2.0}. State 4 and state 5 are marked as accepting states for the two paths of Q{2,0} respectively. Query 3 is composed of two sub-queries, Q{3,0} and Q{3,1}. The indexing starts from Q{3,0} which is the parent of Q{3,1}. Because Q{3,0} has no tangling branch, its transit branch is indexed right away at the beginning. The tangling branch contains an ancestor/descendant relationship between

elements *a* and *b*, so state 6 with self-loop is created for this relationship. The symbol "ε" means that the transition needs no input to enter state 6. After the transit branch is indexed, state 7, which is the state of the last element of the transit branch *b*, becomes the start state for indexing Q{3,1}. When indexing for Q{3,1} begins, the start state is marked as branch point for Q{3,1}. Because Q{3,1} has two tangling branches, they are indexed one by one. State 8 and state 9 are then marked as accepting states for the two branches, respectively. Query 4 is indexed in the same way and has two branch points at state 2 and state 10.

This example shows that the query indexing treats a complex query as a whole. The branch points of queries can be found when corresponding nodes are reached in the QueryIndex tree.

Operator Package:

This package contains NFAexecution_Bfilter class, which conducts the main matching algorithm of Bfilter. Some supporting classes from Yfilter are reused with minor changes accordingly. They are RunstackElementBasic, PredictionEvaluation, ResultCollection and BottomStream.

XMLTree Package:

This package contains XMLTree class, which creates parsing events and passes them to the runtime system based on the SAX event-based XML parser. This package is derived directly from Yfilter.

Filter Package:

In this package, the EXFilterBasic_Bfilter class is created to detect branch points in documents and send parsing elements to the NFAexecution_Bfilter for match

processing. It contains all of the components described earlier and organizes the matching process. In this package, EXFilter_Bfilter class contains the Main method.

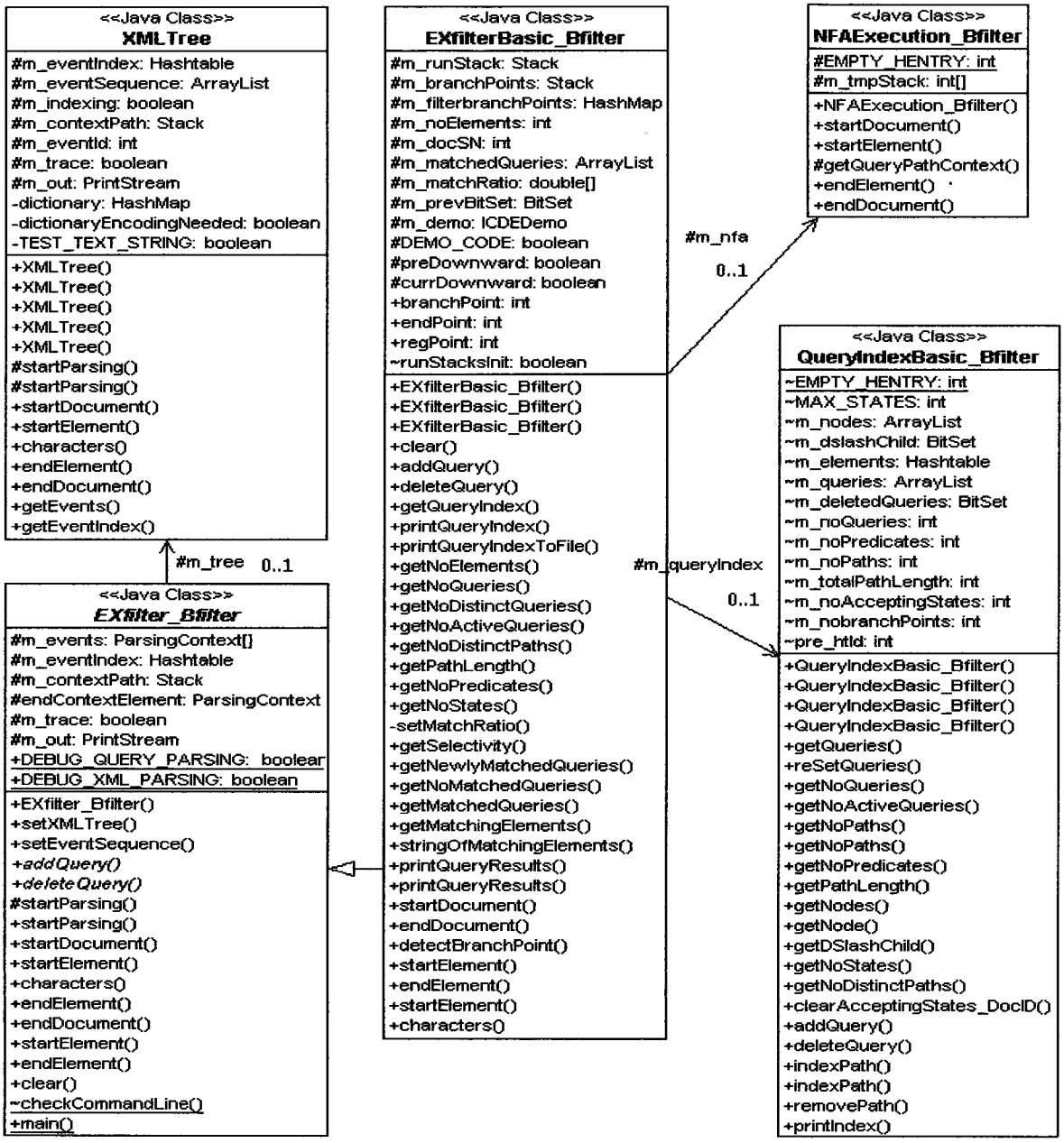


Figure 3.8 Diagram of the Major Classes in Bfilter

Figure 3.8 shows the details of the major classes of Bfilter in a class diagram. The EXfilterBasic_Bfilter inherits from the abstract class EXfilter_Bfilter and implements event handlers for the start document, end document, start element and end element event

from the XML Tree. XML Tree is a placeholder for the incoming XML document stream from the SAX XML parser, and passes events to EXfilterBasic_Bfilter.

The EXfilterBasic_Bfilter maintains a QueryIndex tree by using class QueryIndexBasic_Bfilter. The QueryIndexBasic_Bfilter uses hash table-based data structure to index the XPath queries. The EXfilterBasic_Bfilter can detect the branch point of the incoming XML document stream, and dispatches NFAExecution_Bfilter to carry out matching. In the class diagram, the XMLTree is derived from Yfilter. Appendix B presents the main sequence diagrams for Bfilter to illustrate how Bfilter operates.

3.2.3 Filtering Algorithms

As mentioned in the previous section, the filtering algorithm contains three main classes: EXfilterBasic_Bfilter, QueryIndexBasic_Bfilter and Execution_Bfilter. Figure 3.9 shows the procedure for handling the start tag event during filtering process. Figure 3.10 shows the procedure for matching branches when a match of branch point is found. Figure 3.11 shows the filtering algorithm.

The two stacks, documentBranches and queryBranches, are used to hold the branch points of documents and queries respectively. One stack, called runTimeBranch, is used to hold the current elements' branch of the document. When an element is read in, the element is pushed onto the runTimeBranch Stack. When the element's end tag arrives, the element is popped out from the runTimeBranch Stack. When an element sent to the QueryIndex Tree reaches an accepting state for a path of a sub-query, the portion in the current elements' branch, which is from the element that triggers the root of the sub-query to the current arriving element, will be saved to the sub-query as a candidate for a match of its path.

```
Procedure: HandlingStartTag(Document tag e)

        //get current state in gIndex for e
        Send e to gIndex for states lookup;
        For each current state c
                If c is branch point for query q in Q
                        push (e, q) onto queryBranches;
                If c is accepting state for query q in Q
                        If q has no predicates on this path
                                Make the path as matched;
                        Else
                                //delay the matching until
                                //the branch point is matched.
                                Save branch in the node;
                        End if;
                End if;
        End for;
End Procedure;
```

Figure 3.9 The Procedure for Handling Start Tag

```
Procedure MatchingBranches()

        For each matched query q, do
                If q's child is not matched
                        //no matching is needed
                        Set q as not matched;
                Else
                        Process matching on all branches of q;

                        If q is failed in matching
                                //reset states of all sub-queries
                                Set sub-queries as unmatched;
                        End if;
                End if;
        End for;
End Procedure;
```

Figure 3.10 The Procedure for Matching Branches

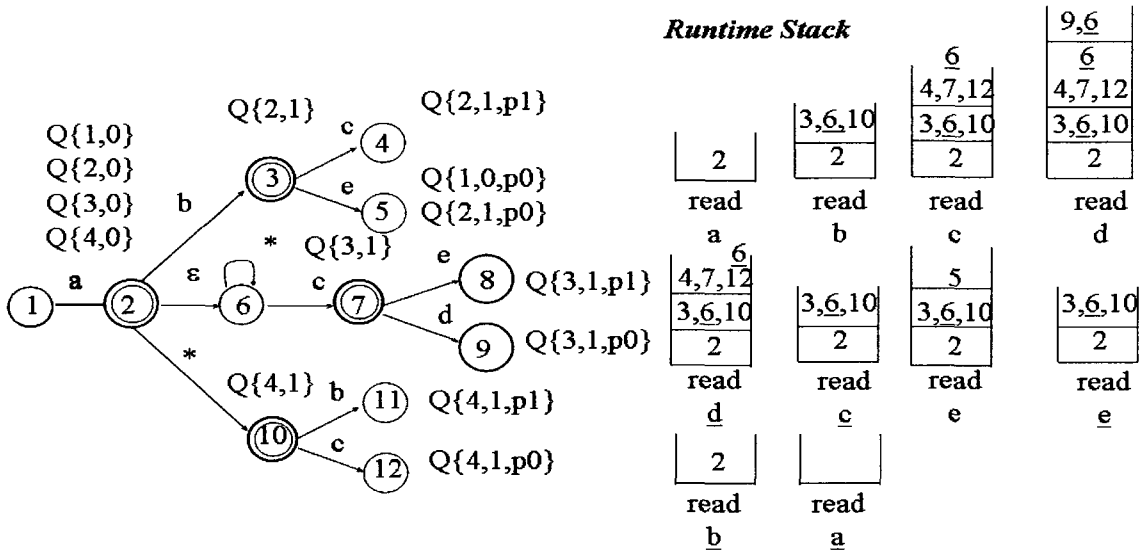
```
Filtering Algorithm
Input:  QueryIndex gIndex
        Stack runTimeBranch
        Stack documentBranches
        Stack queryBranches
        Incoming element e
        List CurrentQueries Q
Init:
        gIndex and Q is populated by user requests

        runTimeBranch = Empty;
        documentBranches = Empty;
        queryBranches = Empty;

        While incoming element e is not the end of document
                If e is start element then
                        Push e onto runTimeBranch;
                        If document branch point is detected
                                Push the top of runTimeBranch onto documentBranches;
                        End if;
                        Call Procedure HandlingStartTag(e);
                Else
                        Pop e from runTimeBranch;

                        //a branch point in queries matches a branch point in document
                        If e is a branch point in queryBranches and documentBranches
                                Call Procedure MatchingBranches();
                                Remove from documentBranches and queryBranches;
                        Else if e is a branch point in queryBranches
                                //no matched branch point in document, simply set
                                //unmatched
                                For each matched query q for this branch point, do
                                        Set q and its sub-queries as unmatched;
                                End for;
                                Remove from queryBranches;
                        Else if e is a branch point in documentBranches
                                // no matched branch point in queries, do nothing
                                Remove from documentBranches;
                        End if;
                End if;
        End while;
End Algorithm;
```

Figure 3.11 The Filtering Algorithm

When the end tag of an element is read in, the element is popped out from the runTimeBranch Stack. At the same time, both documentBranches and queryBranches are checked to see if a branch point that corresponds to this element exists in both stacks. If it does, the matching process starts for the sub-queries that have this branch point; otherwise the matching process is not performed. The matching process includes finding matches for all the sub-queries' branches and evaluating the predicates on the branches. Finally if the corresponding branch point is stored in documentBranches and queryBranches, it is popped out from these two stacks.

Figure 3.12 shows an example of executing the Bfilter. Figure 3.12 (a) depicts the execution in the query index tree. Figure 3.12 (b) shows the changes of the statuses of documentBranches and queryBranches. Notice that a dummy element should be pushed onto both stacks if no branch point corresponding to a start tag is found. This is to make the branch point matching in the two stacks easy by simply looking at the top elements in both stacks. The process of pushing dummy elements onto the stacks is not shown in Figure 3.12 (b) for brevity. In this example, the left hand side in Figure 3.12 (a) represents the query index as described in Figure 3.7. The incoming XML document stream is a/b/c/d/d/c/e/e/b/a, where underscored letters are end tags. The right hand side in Figure 3.12 (a) depicts the changes of the status of the Runtime Stack. This Runtime Stack is a part of the query index tree; it is used to hold active states during matching process.

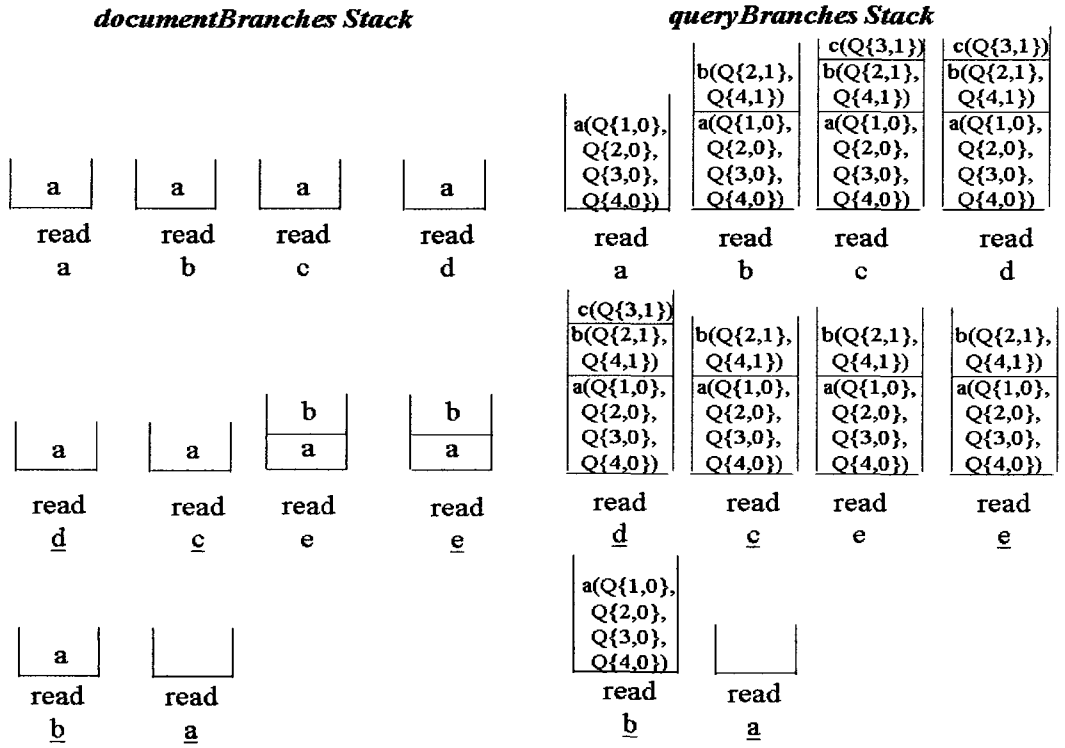
At the beginning the Runtime Stack is empty. When the first element, *a*, is read in, state 2 is reached in the QueryIndex Tree and is pushed onto the Runtime Stack. This state matches four branch points of the queries.



XML document stream: a/b/c/d/d/c/e/e/b/a document branch point: a, b

Execution in Query Index Tree

(a)



Execution in Branch Point Stacks

(b)

Figure 3.12 Example of Execution in Bfilter

(a) Execution in Query Index Tree (b) Execution in Branch Point Stacks

These four branch points are the first sub-queries of Q1, Q2, Q3 and Q4 and are represented as Q{1,0}, Q{2,0}, Q{3,0} and Q{4,0}, respectively. These branch points are wrapped as a unit which is associated with the element *a* so that they can be found when the end tag of element *a* is read. Because the first element is also the root of the XML document, a node associated with element *a* is pushed onto the documentBranches Stack.

When element *b* arrives, three states are active. First, there is a transition of *b* from state 2 to state 3, in which state 3 becomes active. Second, because state 6 is the next state of state 2 that requires no input, state 6 is also active. Finally, the transition of "*" from state 2 to state 10 matches any tag name so that state 10 is active too. Thus, state 3, state 6 and state 10 are pushed onto the Runtime Stack based on the QueryIndex Tree. Because states 3 and 10 are branch points of Q{2,1} and Q{4,1}, the two branch points are pushed onto the queryBranches Stack that are associated with element *b*. At this stage element *b* has not been detected; thus we do not know whether or not it is a branch point in the document. However, it will be detected before the end tag of b is read.

When element *c* is read, states 4, 7, 12 and 6 are reached from previous states 3, 10 and 6. These states are pushed onto the Runtime Stack. State 6 is automatically added because it needs no input. States 4 and 12 are accepting states of Q{2,1,p1} and Q{4,1,p0}, thus, the current branch in the runTimeBranch Stack will be saved in the query object for Q{2,1} and Q{4,1} as a candidate of match for the path 1 of Q{2,1} and path 0 for Q{4,1}, respectively. Because state 7 is a branch point of Q{3,1}, Q{3,1} is pushed onto the queryBranches Stack. After element *d* is read in, states 9 and 6 are pushed onto the Runtime Stack. The current branch is saved in Q{3,1}, because state 9 is the accepting state of Q{3,1,p0}.

When the end tag of *d* is read in, the top states (9 and 6) are simply popped out from the Runtime Stack. When the end tag of *c* arrives, top states 4,7,12 and 6 are popped out from the Runtime Stack. Because element *c* has a corresponding branch point in queryBranches Stack but not in documentBranches Stack, the corresponding branch point is popped out from queryBranches and Q{3,1} is marked as unmatched. No matching will be processed for the sub-query Q{3,1} at this point because element *c* is not a branch point in the document.

When the element *e* is read in, state 5 is reached from state 3. The current branch is saved as a match candidate for corresponding paths Q{1,0} and Q{2,1} respectively. Because element *e* follows a sequence of pop operations (due to end tags), the element *b*, which is on the top of the runTimeBranch Stack before the element *e* is pushed, it is detected as a branch point of the document according to the branch point detection algorithm. Thus *b* is pushed onto the documentBranches Stack. After the end tag of *e* is read in, the top of the Runtime Stack is popped out.

When the end tag of *b* is read in, the top states on the Runtime Stack are popped out first. Because element *b* has associated branch points in both the documentBranches Stack and queryBranches Stack, Q{2,1} and Q{4,1} find matches in the document. Thus the matching process starts for Q{2,1} and Q{4,1} after the branch points are popped out from both stacks. Because the two paths of Q{2,1} have match candidates, Q{2,1} is marked as matched. Notice that this example does not provide predicates on path. If it does, the matching process has to evaluate all predicates; only the path match is not enough. Because the Q{2,1} is matched, the current branch from the root of Q{2,0} to the root of Q{2,1} is saved in Q{2,0} as a match candidate of the transit branch of

Q{2,0}. Suppose Q{2,1} is not matched, the algorithm does not save the candidate for Q{2,0}, because a parent query cannot match if its child query is not matched. Q{4,1} is different from Q{2,1}, because its path 1 has no candidate. So Q{4,1} is simply marked as unmatched at this point.

Finally, when the end tag of *a* is read in, Bfilter finds a match of branch points in the documentBranches Stack and queryBranches Stack. The four branch points, namely Q{1,0}, Q{2.0}, Q{3,0} and Q{4,0}, match branch point *a* in the document. Because the child query Q{4,0} is unmatched, Q{4,0} is simply marked as unmatched without undergoing the matching process. Q{1,0} is simple because it has only one path and is marked as matched. Because the child query of Q{2,0} is matched, Q{2,0} is marked as matched after the match of its transit branch succeeds. Similarly the Q{3,0} is marked as unmatched because its child is unmatched. At the end of this process, Bfilter checks whether a query is matched or not by looking at its root in the sub-query representation. Thus Q1 and Q2 are matched in this example.

Note that the matching process only starts when a root of a sub-query is popped out from the queryBranches Stack. Whenever a match is determined for a sub-query, no matter whether the result is matched or not, all match candidates of the sub-query are deleted. If the result is unmatched, all the descendants of the sub-query are marked as unmatched.

This example demonstrates the difference between Yfilter and Bfilter. Bfilter matches queries backwards and the matching process only starts when branch points match in both the document and the queries. For queries Q1, Q2, Q3 and Q4 in the example, Yfilter will execute the matching process at each accepting state for the

corresponding queries that are decomposed from the four queries. In particular, Q3 will be decomposed into two simple queries Q3.1 (/a//c/d) and Q3.2 (/a//c/e); Q4 will be decomposed into two simple queries Q4.1 (/a/*/c) and Q4.2 (/a/*/b). From the queryIndex Tree we can see that states 9 and 12 will be the accepting states for Q3.1 and Q4.1 respectively. Thus the matching process occurs when the two states are reached in Yfilter, but no matching is processed for Q3 and Q4 in Bfilter. In the case of a document having more content after the end tag of element *b* is read, and the remaining part matches Q3.2 and Q4.2, Yfilter will conduct post-processing to verify these matched branches and will eventually discover that the decomposed queries are not matched in the same place in the document.

Unlike Yfilter, Bfilter will clean up all candidates for the current sub-queries as well as their descendants when the corresponding branch point in the document is read over. Bfilter only keeps match candidates for the current portion of the XML document. Thus, if a sub-query and its descendants are matched, it needs not do anything when the new portion of the document arrives. Otherwise the match of branch point restarts from the root of the last sub-queries in the new portion. From this example we can see that Bfilter is more efficient to deal with complex queries than Yfilter.

3.2.4 Comparison of Space and Time Complexity

This thesis does not provide the analysis of space size and complexity for all of XML filters discussed in this paper. This is because it is complex and difficult to build a general model for these filters that use different techniques and there is lack of similar work conducted on these filters and other related materials. A formal and thorough

analysis can be investigated in the future research. This section provides a brief comparison of space and time complexity between Yfilter and Bfilter.

Bfilter reuses the QueryIndexBasic class of Yfilter to implement a hash table-based data structure to index queries - queryIndex Tree. The new structure is capable of holding the queries together as a whole and determining if the current active state is also a branch point for some other queries. The Runtime Stack is used to hold the active states during the filtering process, which is depicted in Figure 3.12 (Section 3.2.3). The ancestor/descendant relationship "//" and wildcard "*" that appear in the queries are two important factors for the total number of active states in the Runtime Stack. The ancestor/descendant relationship "//" creates a self-loop state in NFA and can increase the number of active states exponentially [29]. The wildcard "*" matches any element from a XML document being filtered and thus the number of active states can also increase. For example, when the b tag is read in Figure 3.12, state 10 is also pushed onto the Runtime Stack together with others. Given a set of queries Q $\{q_i : 0 \leq i \leq N$, where N is the total number of queries$\}$, in the worst case where all of the queries' element names are "*" and there is no "//" in all of the queries, the maximum number of active states that contributed by "*" itself is Σn_i, where n_i is the number of elements in q_i.

Given a XML document D and a set of queries Qs, The queryIndex Trees constructed by Qs have the same number of nodes in both Yfilter and Bfilter. Let Ts be the sequence of start tags and Te be the sequence of end tags in D. Ts and Te have the same set of tags but in a reversed order. The active states are triggered by Ts in both Yfilter and Bfilter. The Runtime Stacks have the same number of active states in both filters, too. However, Bfilter uses the additional queryBranches to hold the branch points

of queries, which saves all active states that are the queries' branch points in the current implementation. In the best case for Bfilter, there are no active states corresponding to the branch points that have even been reached, the space size of the queryBranches Stack is hence zero. So the space size of Bfilter is the same as Yfilter. In the worst case for Bfilter, all of the active states can be branch points. For example, if the queryIndex tree has only one query and the query has only one element, the element is the root of the query and thus it is also a branch point. When the state corresponding to this element is reached, only one active state will be in the Runtime Stack, and it will also be pushed onto the queryBranches Stack. The space size of the active states for this worst case in Bfilter will be twice of that in Yfilter. However, because the number of the branch points in a tree is at most half of the total number of nodes in the tree, so the number of active states corresponding to the branch points will be approximately half of the total number of the active states. Therefore, the total space size in Bfilter will be 1.5 times of that in Yfilter in general. In the future work, a light weight data structure such as index array (BitSet) may be used to hold the queries' branch points to reduce the space size.

In order to compare the time complexity between Yfilter and Bfilter, we can briefly categorize the filtering cost into two parts. The first part is the cost for matching the incoming element names from a XML document with the element names of queries. The second part is the cost of matching the predicates of the queries against those in incoming document. The fist part happens during the lookup stage in the queryIndex Trees. The cost for the lookup stage is the same for both filters. The costs for the second part, however, are different for these two filters. In the worst case for Bfilter, all of the queries are matched so that all of the predicates in these queries are evaluated eventually.

The cost for Yfilter, in the worst case, is identical to the cost for Bfilter. In the best case for Bfilter, the branch points matching fails on all of the queries at the beginning. Because the branch point matching process starts from the last branch point of a query and no predicates having been evaluated at the point; the second part could cost almost nothing for Bfilter. However, Yfilter will evaluate predicates when the accepting states are reached for these queries. Let T-name and T-predicates be the costs for the first part and the second part, respectively. Let Tb and Ty be the filtering costs for Bfilter and Yfilter, respectively. In the best case for Bfilter, Tb/Ty = T-name/(T-name + T-predicates). In the worst case for Bfilter, Tb = Ty. Thus, Bfilter can achieve higher efficiency than Yfilter in general.

There are numerous possible scenarios, e.g., the number of nested paths, "//" and "*", that could complicate the analysis of Bfilter and Yfilter and any other XML message filters. Thorough complexity analysis is extremely complicated and difficult, and may not reveal cost for realistic or desired scenarios. Therefore, the next chapter will compare Bfilter and Yfilter through a number of experiments that consider various scenarios and different parameters.

Chapter 4: Performance Results

This chapter provides the performance measurement results for the current implementation. As discussed earlier, Bfilter is built on top of Yfilter to reuse existing components. This implementation is suitable for comparing the filtering algorithms. The performance analysis is divided into two parts. The first part focuses on the overall results of the performance testing. The second part focuses on comparing the effect of different query attributes on the performance of Yfilter and Bfilter.

4.1 Experimental Environment

The BFilter solution was implemented using Java 1.6, and run on a PC with an AMD 1.6 GHz processor and 1.0 Gb of main memory running the Windows Vista operating system. The Java virtual machine memory size was set to 256Mb.

BFilter was compared with the release version of YFilter. The data type used in the experiments was the same as in Yfilter: News Industry Text Format (NITF) [17]. ToXGene [3] was used to generate the XML document. A total of ten documents were used in this experiment. These ten documents were used in the evaluation of Yfilter presented in [17]. The average document size was 10 Kbytes. Running experiments with a larger set of documents to enable statistical analysis of the performance result forms an important direction for future research.

The query generator in the YFilter test suite [17] was used to generate the queries. Parameters, such as query depth, number of branches, number of predicates and probability of "//" and wildcard "*" were varied in each of the experiments. To measure

system performance, the java API is used to get the time interval in milliseconds between the start point and end point of each filtering process. Each result in the experiment reflects an average of ten runs.

4.2 Measurement Parameters

This section provides the definitions of all parameters used in the performance measurement.

- **Indexing Cost** is the total time needed to read the user requests from plain text file, parse them into query type that can be recognized by Bfilter and insert them into a QueryIndex Tree.
- **Filtering Cost** is the total time needed to filter the input documents. It is the time period that starts from reading in the start tag of the root of the first XML document from the SAX event-based XML parser, and ends when all documents are filtered and the matching results of the user requests are obtained. In the experiments filtering cost was used to measure the performance as the focus was on the comparison of the filtering algorithms only.
- **Query Depth** is a sequence of element names separated by "/" and "//" (which stand for parent/child relationship and ancestor/ descendent relationship respectively). The number of element names determines the query depth. For a complex query, query depth is the depth of its longest path.
- **Number of Predicates** is the total number of predicates in a query. An element name in a user request may contain predicates. When matching such a query, both the element name and its predicates have to find a match in the XML document.

- **Number of Nested Paths** refers to the number of nested paths in a complex query. A simple query has no nested path.
- **Probability of "//"** is the probability that the ancestor/descendant relationship appears in a query. The ancestor/descendant relationship creates a self-loop state in NFA and can increase the number of active state during matching. This parameter is used to measure how sensitive an algorithm is to the increase of "//".
- **Probability of "*"** is the probability that "*" appears in a query as an element name. The wildcard "*" matches any element from a XML document being filtered and thus the number of active states can increase. This parameter is used to measure how sensitive an algorithm is to the increase of "*".

4.3 Performance Comparisons

In this section, three general test cases are described. Each case contains a scenario description and analysis.

4.3.1 Test case 1 - branch point mismatching

This case demonstrates a scenario that a complex query cannot find branch point matching in a XML document. The query and XML document are shown in Figure 4.1.

This query is complex. Its last branch point is *w*, which has two branches "/b" and "/c". However, all the three nodes *w* in the XML document are not branch points. After the start tags *nitf*, *head*, *a*, *b*, *w* and *c* is read in sequentially, the end tags *c* and *w* follows. At this point the node *w* (id = "9") is detected. Because it is not a branch point in the document, the matching for the last branch point *w* in the query fails. Similarly, the other two nodes (id = "99" and id = "9") in the document also fail to match the query's branch

point. Thus, the matching of the higher part of the query has never been processed in Bfilter. When Yfiter processes this filtering, it decomposes the query into three simple queries:

/nitf[@change.time=1]/head[@id=9]/a/c
/nitf[@change.time=1]/head[@id=9]/a/b/w/b
/nitf[@change.time=1]/head[@id=9]/a/b/w/c

Query: /nitf[@change.time=1]/head[@id=9]/a[/c]/b/w[/b]/c
Document:

```
<nitf change.date="1" change.time="1" id="5" uno="11" version="1">
    <head id="9">
        <a id="9">
            <b id="9">
                <w id="9">
                    <c id="9"></c>
                </w>
                <w id="99">
                    <b id="19"></b>
                </w>
            </b>
            <c id="9"></c>
            <b id="9">
                <w id="9">
                    <c id="9"></c>
                </w>
            </b>
        </a>
    </head>
</nitf>
```

Figure 4.1 The Sample Query and XML Document

Because each simple query can find a match in the document, Yfilter spends time on matching each of them and finally discovers they are not matched in the same place in the document. Figure 4.2 shows the comparison of the filtering cost for the two filters. Because the filtering cost for this test case is small, we perform the test case 40 times in a loop and measure the total filtering cost. The results indicate that Bfilter is more efficient than Yfilter in the case of branch point mismatching.

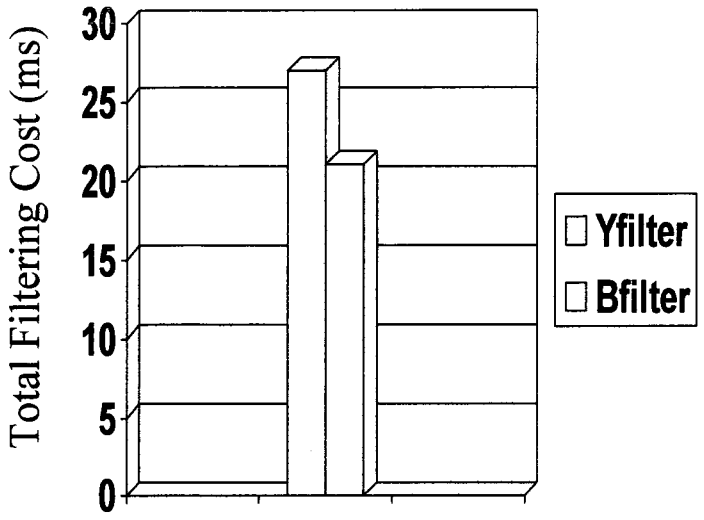


Figure 4.2 Comparison of Yfilter and Bfilter for Test Case 1

4.3.2 Test case 2 - test for queries without branch points

This case is to demonstrate a scenario that the current implementation of Bfilter is slower than yfilter while handling simple queries. This is because the current implementation of Bfilter reuses the components in Yfilter and reconstructs the classes on top of them to meet its needs. It has no performance gain to offset the overhead in the case of simple queries.

Figure 4.3 shows a comparison of the filtering cost corresponding to the number of queries for matching. In this test case the criteria used to generate queries are shown in the figure. The probability of "*" and "//" are 20% and 0 respectively. The number of predicates is 6 and the query depth is 6. The number of queries generated varies from 500 to 2000. None of the queries has branches.

Although the result in Figure 4.3 is for a particular set of criteria, the variation of the parameters in the criteria does not change the result in the case of simple queries.

Considering the fact that Bfilter is more efficient than Yfilter in dealing with complex query, we believe the current implementation has room for improvement.

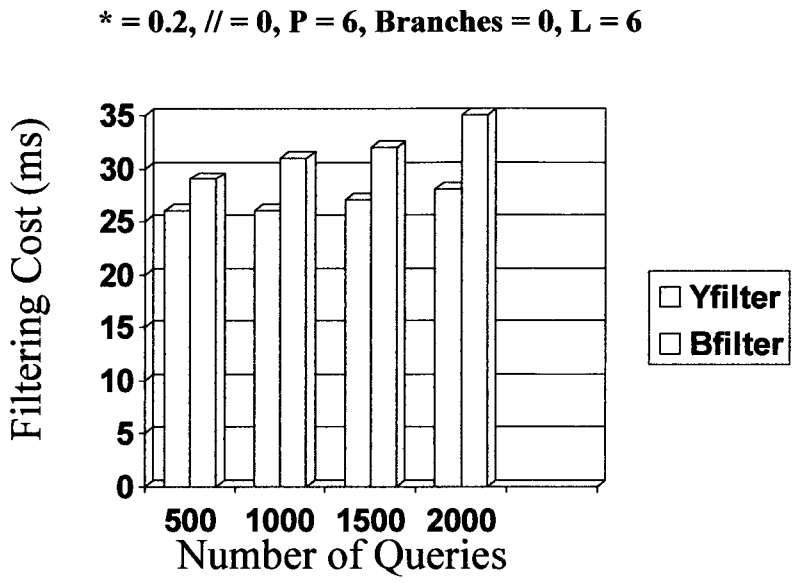


Figure 4.3 Comparison of Yfilter and Bfilter for Test Case 2

4.3.3 Test case 3 - test for queries generated with 2 nested paths

This case demonstrates the performance of Bfilter and Yfilter while dealing with complex queries.

In this test case the criteria used to generate queries is the same as the one employed in test case 2, except that there are two nested paths in this experiment. The results shown in Figure 4.4 indicate that Bfilter is faster than Yfilter while the number of queries is less than or equal to 1000, but slower when the number of queries increases. In the current implementation of Bfilter, the query indexing and prediction matching processes are built on top of the Yfilter process by wrapping the existing methods. Query indexing reunites the separated nested paths to make a complex query after splitting Yfilter's indexing. Prediction matching takes the whole context path of a branch and

feeds it into the Yfilter's prediction matching method reused in Bfilter after cutting off the part beyond the current branch point. When the number of queries increases, the overhead becomes too large and offsets the advantages of Bfilter's filtering algorithm.

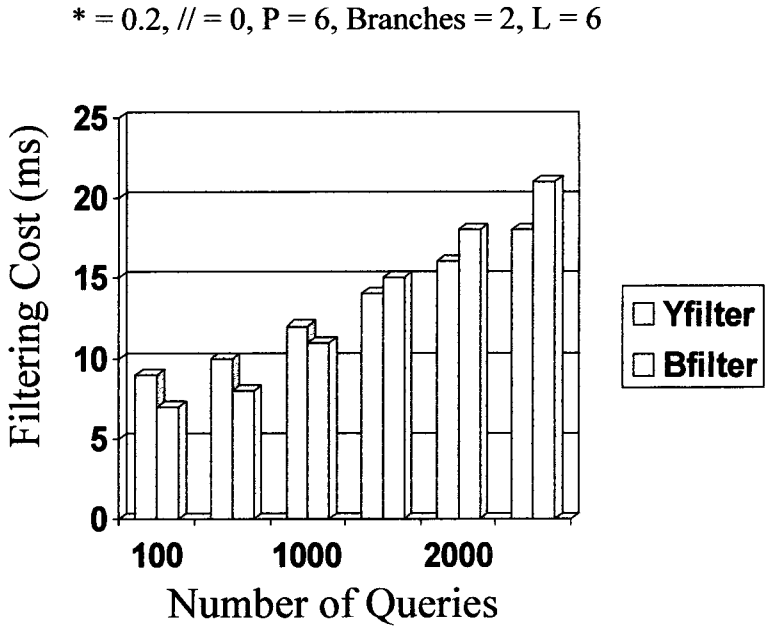


Figure 4.4 Comparison of Yfilter and Bfilter for Test Case 3

4.4 Measurement Results for Different Query Attributes

This section presents the effect of five different parameters on the performance of Bfilter and Yfilter. The selected values of each parameter are outlined below:

Query depth (L): 3, 6.
Number of nested paths (No.ofNP): 0, 1, 2, 3, 4, 5.
Probability of wildcard (*): 0, 0.1, 0.2, 0.4.
Probability of ancestor/descendant relation (//): 0, 0.1, 0.2, 0.4,
Number of predicates (P): 0, 2, 5, 10.

As mentioned in Section 5.1, a total of ten XML documents were generated and used in this experiment. The default level of these documents is 6. A query cannot find a match if its depth is greater than 6, so the selected values for the query depth are 3 and 6.

Since the number of nested paths cannot be greater than the maximum value of the query depth in a query, the number of nested paths is varied from 0 to 5. The probabilities of wildcard and ancestor/descendant relation range from 0 to 0.4. The number of predicates varies from 0 to 10. The ranges of parameters used in the experiments are apt for analyzing the relative performance of Bfilter and yfilter. Because the total number of permutations of the five variables is very large, one parameter is varied at a time in each of the experiments. The number of queries generated by each group of values from the five parameters is set as 100.

4.4.1 The effect of the query depth

The purpose of this experiment is to determine the effect of the query depth on the filtering cost. In this experiment, the number of nested paths is varied from 0 to 5, and two parameter sets are used:

Set 1: the probability of "*" and "//" are 0, the number of predicates (P) is 2, and the query depth (L) is 6.

Set 2: the probability of "*" and "//" are 0, the number of predicates (P) is 2, and the query depth (L) is 3.

Figure 4.5 has two graphs. Figure 4.5 (a) uses the parameter set 1, and shows the result when the query depth equals 6. Figure 4.5 (b) uses the parameter set 2, and shows the result when the query depth equals 3.

Figure 4.5 (b) illustrates that in the cases where query depth L is 3, the number of nested paths has no significant effect on the filtering cost after it reaches 3. This is because the query generator is a reused component from Yfilter which can create at most one branch at each level of a query. Yfilter does not deal with recursive nested paths (a

branch containing another branch). If the query depth is 3, the query generator still creates 3 nested paths for a query even though the number of nested paths is set to be greater than 3. So the query depth will be set as 6 for the remaining experiments.

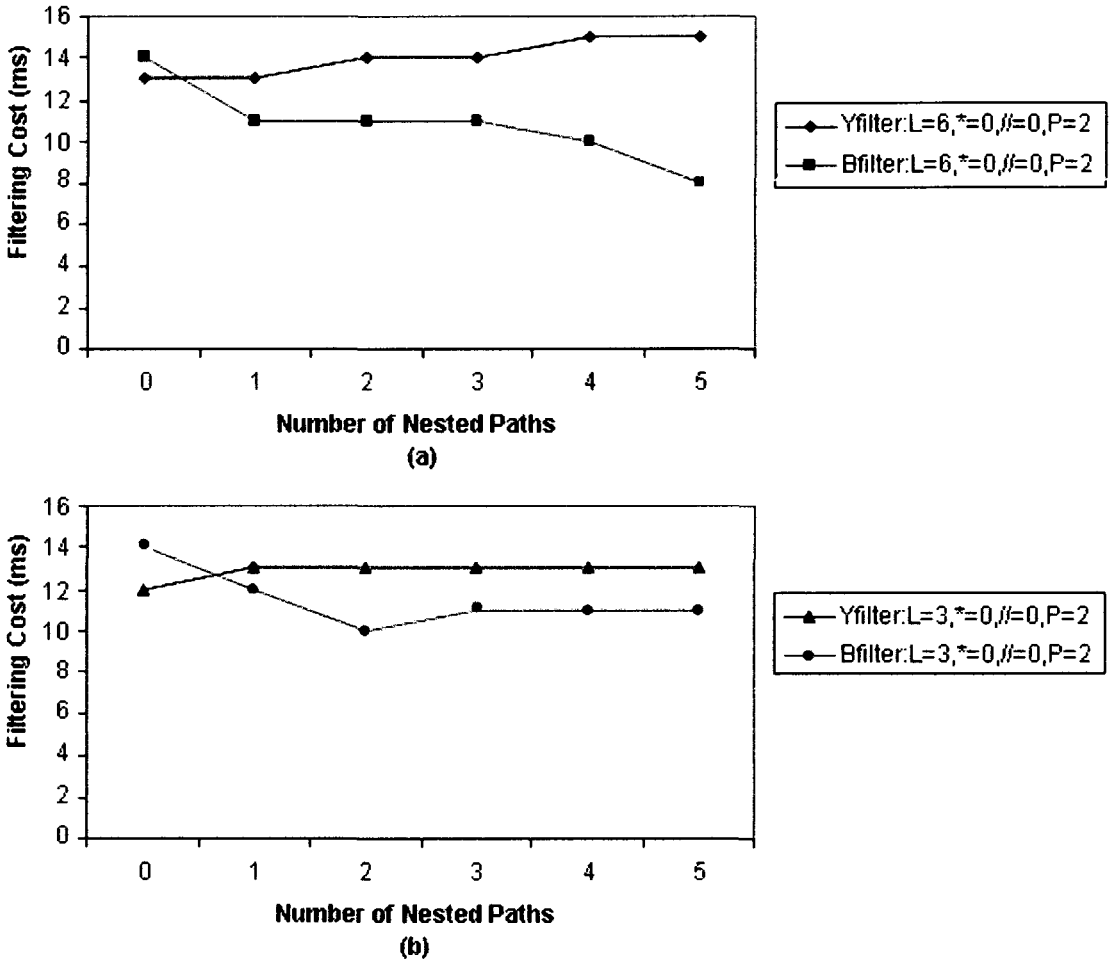


Figure 4.5 The Effect of the Query Depth

(a) Set 1 (b) Set 2

4.4.2 The effect of the number of nested paths

In this experiment, the number of nested paths is varied from 0 to 5. There are four groups of comparisons that use different sets of parameters:

Set 1: the probability of "*" and "//" are 0, the number of predicates (P) is 2, and the query depth (L) is 6.

Set 2: the probability of "*" and "//" are 0.1, the number of predicates (P) is 2, and the query depth (L) is 6.

Set 3: the probability of "*" and "//" are 0.2, the number of predicates (P) is 2, and the query depth (L) is 6.

Set 4: the probability of "*" and "//" are 0.4, the number of predicates (P) is 2, and the query depth (L) is 6.

The purpose of this experiment is to determine how sensitive the two filtering algorithms are with respect to the number of nested paths in complex queries.

Figure 4.6 contains two graphs. Each graph shows two groups of comparisons. Figure 4.6 (a) presents the results for set 1 and set2. Figure 4.6 (b) presents the results for set 3 and set 4. In every group of comparison (in the same set of parameters), Bfilter beats Yfilter except when the number of nested paths equals zero, in other words, when the queries are simple. As explained in Section 4.3, this is because the current implementation of Bfilter reuses the components in Yfilter and reconstructs the classes on top of Yfilter components. It has no performance gain to offset the overhead in the case of a simple query.

In Yfilter, when the number of nested paths increases the number of simple queries decomposed from complex queries also increases. This is because Yfilter decomposes a complex query by creating a simple query for each nested path. Therefore, Yfilter's filtering cost tends to increase because it needs to do more work during post-processing to verify the separated query as a whole. In the case of Bfilter, because the

number of nested paths increases, the probability of branch point matching will decrease. Thus, the processing for matching will be reduced in Bfilter, and Bfilter's filtering cost will decrease.

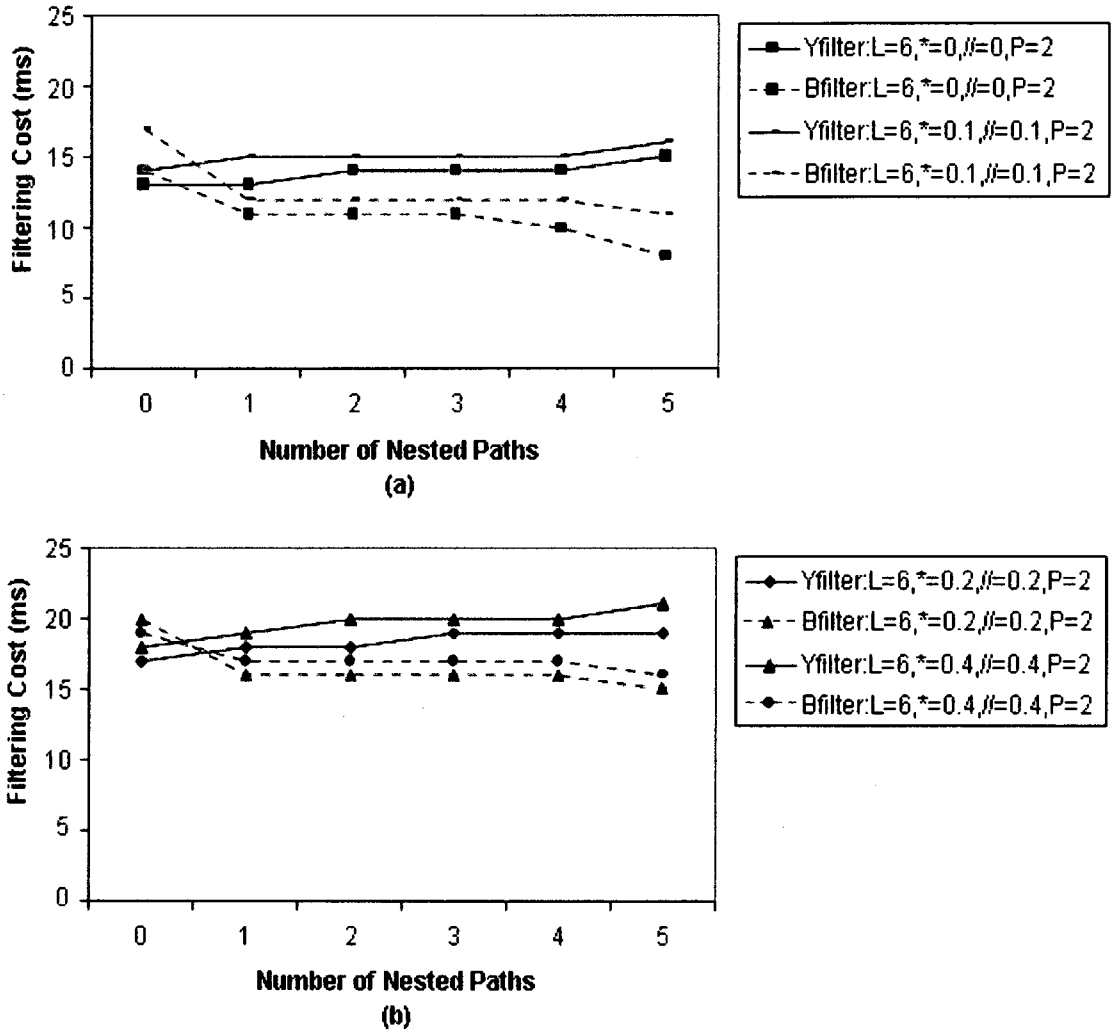


Figure 4.6 The Effect of Different Numbers of Nested Paths

(a) Set 1 and Set 2 (b) Set 3 and Set 4

4.4.3 The effect of probability of "//"

In this experiment, the probability of "//" that appears in the queries is varied from 0 to 40%. There are four groups of comparisons that use different sets of parameters:

Set 1: the number of nested paths (No.ofNP) is 0, the probability of "*" is 0, the number of predicates (P) is 0, and the query depth (L) is 6.

Set 2: the number of nested paths (No.ofNP) is 1, the probability of "*" is 0.1, the number of predicates (P) is 2, and the query depth (L) is 6.

Set 3: the number of nested paths (No.ofNP) is 2, the probability of "*" is 0.2, the number of predicates (P) is 5, and the query depth (L) is 6.

Set 4: the number of nested paths (No.ofNP) is 4, the probability of "*" is 0.4, the number of predicates (P) is 10, and the query depth (L) is 6.

The purpose of this experiment is to analyze how sensitive the two filtering algorithms are to the probability of "//". As described in Chapter 3, Bfilter uses NFA to implement query indexing as Yfilter does. The increase of the number of "//" causes an exponential increase of active states in NFA. If this is an issue for Yfilter, it is an issue for Bfilter too. However, the method that Bfilter uses to perform matching from the underlying NFA is different from Yfilter. The following experiment is intended to uncover the difference.

There are two graphs in Figure 4.7. Each graph shows two groups of comparisons. Figure 4.7 (a) presents the results for set 1 and set2. Figure 4.7 (b) presents the results for set 3 and set 4. The results show that Bfilter is faster than Yfilter in all cases when the number of nested paths is not zero.

As mentioned above, Bfilter reuses the query index tree of Yfilter. It suffers the same overhead as explained in Yfilter due to the large number of active states produced by '//'. This can be seen from the figure: the filtering cost of both Yfilter and Bfilter increases when the probability of ancestor/descendant relationship '//' increases.

However, Bfilter gains in the matching process but not in the query indexing. When the technique of backward matching branch point is used in Bfilter, although the number of active states is large in Runtine Stack in the NFA, these active states may not actually cause the performing of matching, they will simply be popped out from the stack due to the failure of branch point matching.

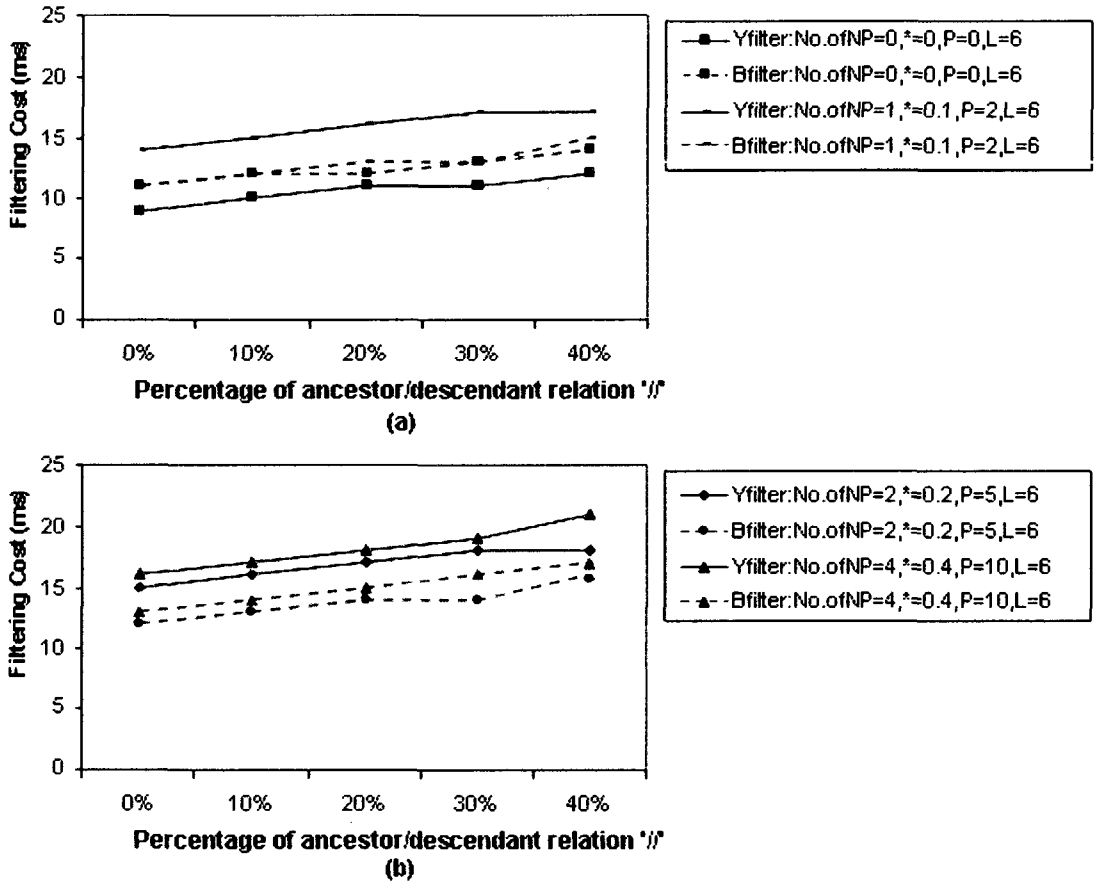


Figure 4.7 The Effect of Different Percentages of '//'

(a) Set 1 and Set 2 (b) Set 3 and Set 4

4.4.4 The effect of probability of "*"

In this experiment, the probability of "*" that appears in the queries is varied from 0 to 40%. There are four groups of comparisons that use different sets of parameters:

Set 1: the number of nested paths (No.ofNP) is 0, the probability of "//" is 0, the number of predicates (P) is 0, and the query depth (L) is 6.

Set 2: the number of nested paths (No.ofNP) is 1, the probability of "//" is 0.1, the number of predicates (P) is 2, and the query depth (L) is 6.

Set 3: the number of nested paths (No.ofNP) is 2, the probability of "//" is 0.2, the number of predicates (P) is 5, and the query depth (L) is 6.

Set 4: the number of nested paths (No.ofNP) is 4, the probability of "//" is 0.4, the number of predicates (P) is 10, and the query depth (L) is 6.

The purpose of this experiment is to uncover how sensitive the two filtering algorithms are to the percentage of "*" queries.

The effect of "*" on the NFA is similar to that of "//". It also causes the number of active states to increase because it matches any tag from the document. Each tag creates an active state on the Runtime Stack as described in Chapter 3. It does not lead to an exponential increase of active states in NFA.

There are two graphs in Figure 4.8. Each graph shows two groups of comparisons. Figure 4.8 (a) presents the results for set 1 and set2. Figure 4.8 (b) presents the results for set 3 and set 4.

The results show that Bfilter is faster than Yfilter in all cases except when the number of nested paths is zero.

The results also demonstrate that when the percentage of wildcard '*' increases, the filtering cost of both Yfilter and Bfilter increases. This is because the wildcard matches any incoming tag from a XML document so that both Yfilter and Bfilter need to do more work due to the increase of active states. This slows down the matching process.

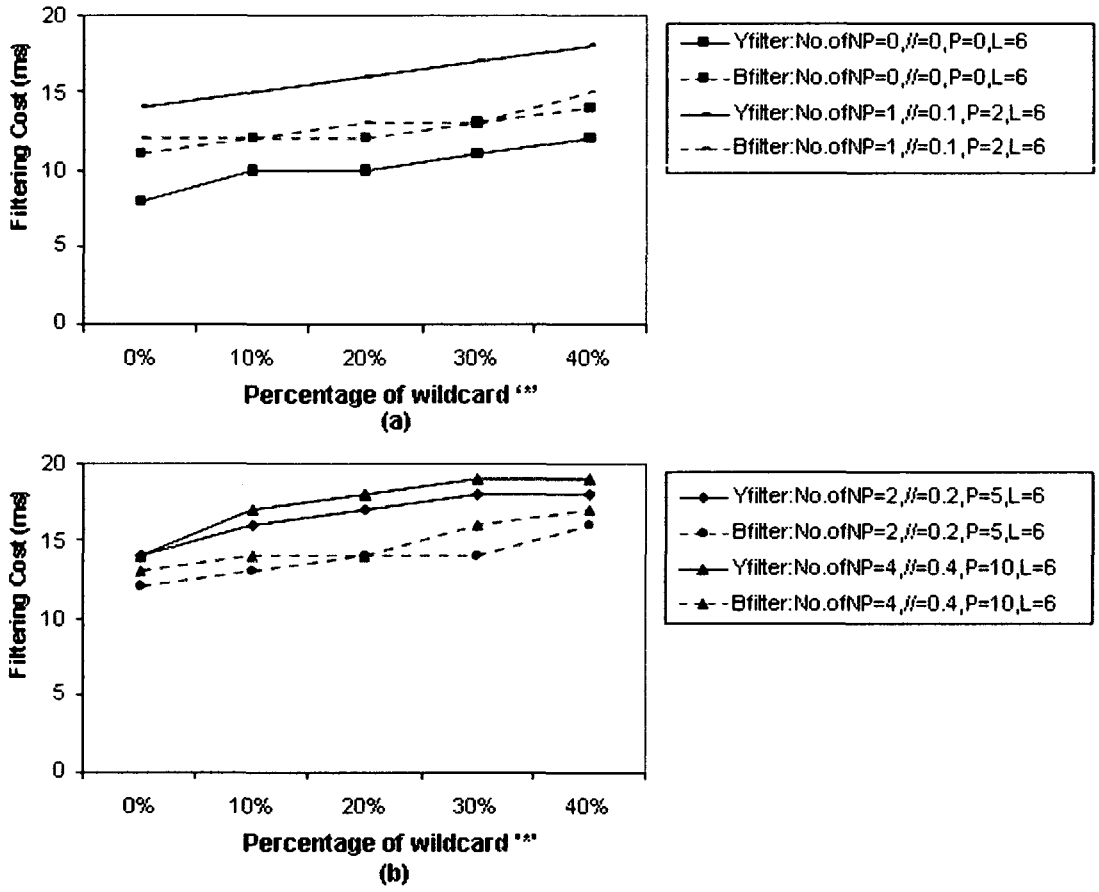


Figure 4.8 The Effect of Different Percentages of '*'

(a) Set 1 and Set 2 (b) Set 3 and Set 4

4.4.5 The effect of different number of predicates

In this experiment, the number of predicates that appears in a query is varied from 0 to 10. There are four groups of comparisons that use different sets of parameters:

Set 1: the number of nested paths (No.ofNP) is 0, the probability of "*" and "//" are 0, and the query depth (L) is 6.

Set 2: the number of nested paths (No.ofNP) is 1, the probability of "*" and "//" are 0.1, and the query depth (L) is 6.

Set 3: the number of nested paths (No.ofNP) is 2, the probability of "*" and "//" are 0.2, and the query depth (L) is 6.

Set 4: the number of nested paths (No.ofNP) is 4, the probability of "*" and "//" are 0.4, and the query depth (L) is 6.

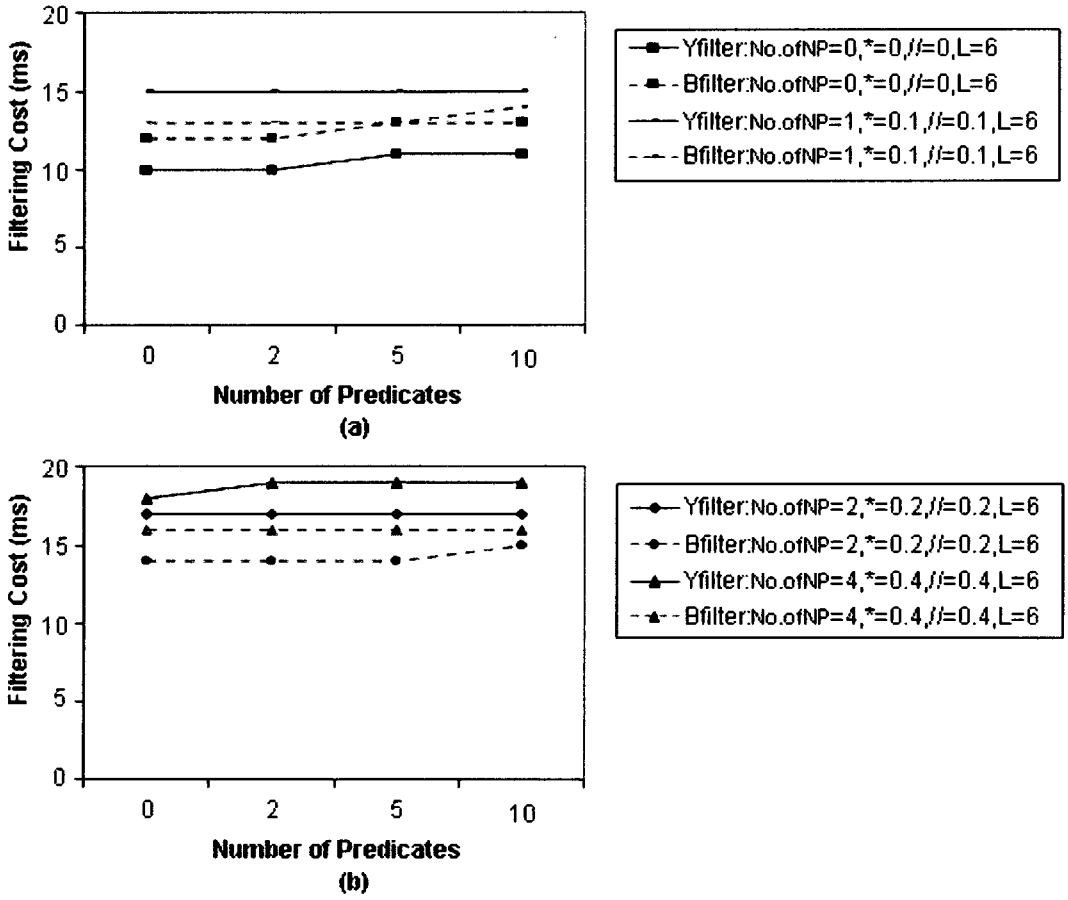


Figure 4.9 The Effect of Different Numbers of Predicates

(a) Set 1 and Set 2 (b) Set 3 and Set 4

The purpose of this experiment is to determine how sensitive the two filtering algorithms are to the number of predicates. This parameter is different than the others. It

has no effect on the NFA-like number of nested paths and probability of "//" and "*". It can only affect the time needed to perform matching. The matching process for a path needs to match all element names on the path as well as all predicates on the path.

There are two graphs in Figure 4.9. Each graph shows two groups of comparisons. Figure 4.9 (a) presents the results for set 1 and set2. Figure 4.9 (b) presents the results for set 3 and set 4.

The results show that Bfilter is faster than Yfilter in all cases except when the number of nested paths is zero. The results also demonstrate that the number of predicates has little effect on the filtering cost of both Yfilter and Bfilter. When the number of predicates increases, the time to verify if a XML tag matches a query node containing a predicate increases. However the chance of failure also increases that causes the matching process to terminate. Thus the overall outcome makes little difference in filtering for both Yfilter and Bfilter.

4.5 Discussion

From the measurement results, we see that Bfilter's filtering algorithm performs better at handling nested path queries than yfilter in the case of mismatching. The current implementation of Bfilter has overheads from wrapping the existing components of Yfilter. There is room for Bfilter to improve its performance by building its own infrastructural component to reduce these overheads.

For a set of queries containing both simple queries and complex queries, a hybrid approach can be used. The simple queries and complex queries may be separated and

handled by Yfilter and Bfilter, respectively. By performing theses filtering operations in parallel, on a multi-core computer for example, the overall performance can be improved.

Chapter 5: Conclusions

BFilter performs XML message filtering and matching by leveraging branch points in both XML documents and user profiles. It uses the technique that matches branch points backwards to defer further matching processes until branch points match in both the XML document and user profile. Unlike Yfilter, Bfilter matches requests with branches without having to decompose them. It treats a complex query as a whole, and no post-processing is needed for a complex query. In comparison to Afilter and Gfilter, Bfilter not only processes matching backward but also utilizes branch point matching as a precondition for further steps. The matching of query branches is delayed until the branch point they attach to is matched. In this way, Bfilter has a high probability of detecting mismatching early in the matching process. XML message filtering in Bfilter can be performed more efficiently in comparison to the three other algorithms.

The measurement results from the current implementation show that Bfilter performs better than Yfilter while handling complex queries. This is observed when the values of the five parameters (Query Depth, Number of branches, Probability of wildcard, Probability of ancestor/descendant relation and Number of predicates) used in the experiments are varied. The filtering costs of both Bfilter and Yfilter increase when the values of these five parameters increase. Because Bfilter is built on the top of Yfilter, it should have similar behaviour as Yfilter does. The gain of Bfilter is not from the implementation but from the filtering algorithm – backward branch point matching. Among the five parameters, the number of branches is an important parameter for both Yfiler and Bfilter. If the number of branches equals zero, all the queries generated are

simple. In this case Bfilter is slower than Yfilter, which reveals that the current implementation of Bfilter gives rise to overheads by wrapping the existing components of Yfilter.

While processing matching, Bfilter only keeps match candidates for the current portion of the XML document. It filters a document portion by portion. Thus Bfilter can specify the matched portions of a document for a particular query. This adds an option for upstream filters to deliver the appropriate parts of a document to downstream filters in Publish/Subscribe Systems.

Some issues in the current implementation need further discussion:

- In the current implementation, a query that has nested paths is represented as a list of sub-queries using a linked list. When indexing the query, a node in the QueryIndex tree will be marked as a branch point of the query if the node is the root of one of its sub-queries. In the filtering process, Bfilter uses the queryBranches Stack to hold the active states that correspond to the queries' branch points. This makes the space requirement for Bfilter higher than that for Yfilter. A light weight data structure such as an index array (BitSet) may be used to reduce the space size of Bfilter.
- In each operation in an accepted state, the current context path is created and saved for each corresponding query branch as a candidate. The current context path is discarded from a query if the query's branch point at which the context path is attached is removed. To create a context path for a query branch, the node that corresponds to the root of the query must be found in runTimeBranch Stack. The context path for the query is from the node to the top node in the runTimeBranch Stack. All of these operations increase the filtering cost. Moreover, since Bfilter

reuses the predicate structure of Yfilter, in which the predicates of a query branch is held in an array corresponding to the query branch, the predicates of a query branch need to be reconstructed accordingly whenever a context path is created for the corresponding query branch. In Yfilter, all context paths that are candidates for the queries are held in a large array until post-processing occurs at the end of document reading. It is evident that Bfilter does extra work in order to reuse the existing components of Yfilter. A Bfilter-specific implementation can further improve the efficiency.

Future Work

Building on the findings and conclusions of this thesis regarding opportunities around Bfilter implementation, there are two proposed plans of action. The first is to build an infrastructural component for Bfilter to improve performance. This involves implementing the XPath query parser, and predicate matching components and the data structure for the query index. The second is to investigate a hybrid approach that utilizes the strengths of both Yfilter and Bfilter. For example, it may be possible to devise a hybrid technique that uses Bfilter to handle complex queries and Yfilter to handle simple queries. Performing these filtering operations concurrently on a multi-core system, for example, can improve overall system performance.

Appendix A: A Case Study for Yfilter, Afilter and Bfilter

A.1 Description of the Case Study

This appendix compares the matching processes executed in Yfilter, Afilter and Bfilter. In this example, a user makes a request as follows:

Q = a//b[/c/d]/e[/g]/f

which is decomposed in Yfilter and Afilter as:

Q1 = a//b/e/f
Q2 = a//b/c/d
Q3 = a//b/e/g

The XML document stream is: a/b/e/f/f/c/c/e/c/d/d/c/b/c/b/e/g/g/e/b/c/a

which is represented in the tree structure as follows:

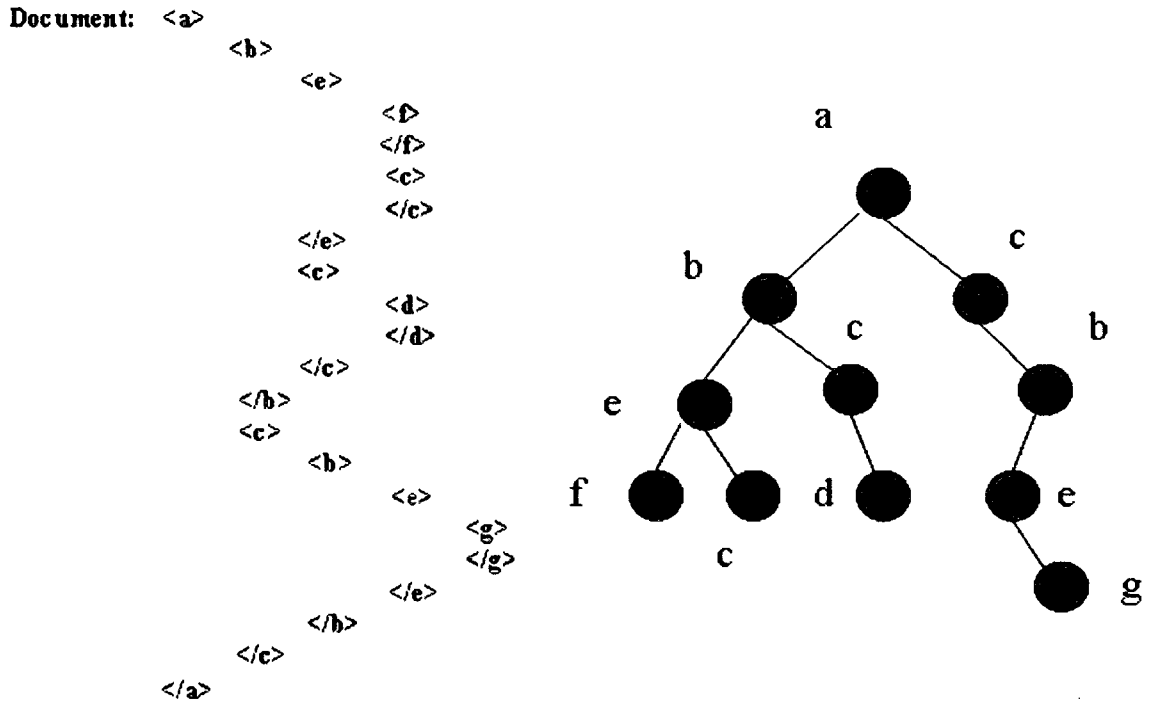


Figure A.1 Document Tree

In this case study the document will not match Q.

A.2 Operations of Yfilter

This case study shows the matching operations of Yfilter. Figure A.2 represents the query indexing tree for Q. Figure A.3 shows the status of the Run Time Stack while performing match. Please refer Section 2.3.2 for details.

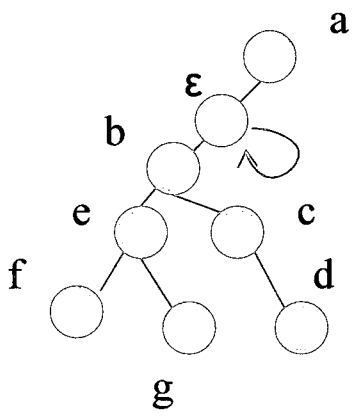


Figure A.2 Query Index Tree of Q

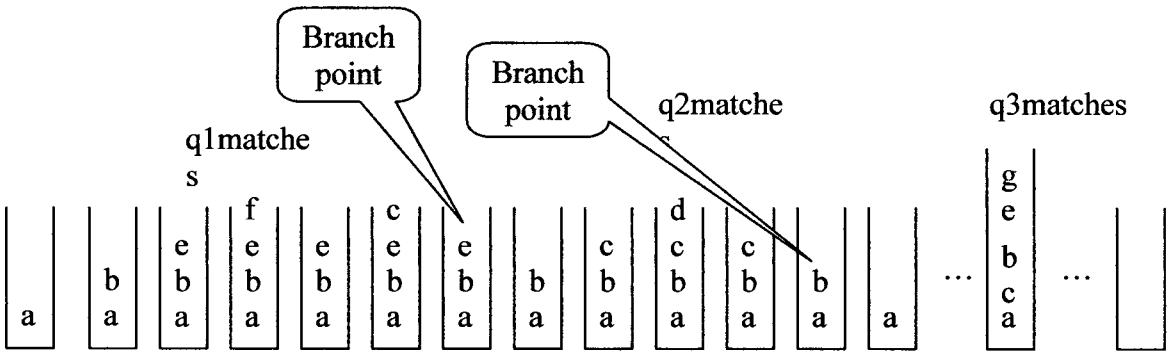


Figure A.3 Yfilter Matching Process

The following list presents the main features of matching performed by Yfilter that distinguish it from other filters:

- As each tag is read in, it is pushed into (start tag) or popped from (end tag) the stack. When each start tag is read in, it searches the possible active states from the Yfilter tree and pushes them onto the stack.
- After matching the three requests, Yfilter carries out post-processing to verify if they match at the same place in the document.
- The verification result is false, because "//" operator is not a simple waiting state with self transition. It introduces one more state for others to share.
- In this case, we can see that after each branch point (e.g., e and b) is cancelled by its end tag, the matching for these requests in the sub-tree (rooted at the branch point) is finished (q3 is supposed to be cancelled).

A.3 Operations of Afilter

In Afilter matching starts from the end of the request. Afilter uses a directed graph and index to keep requests as described in Section 2.3.3. Figure A.4 is the AxisView for the three simple queries.

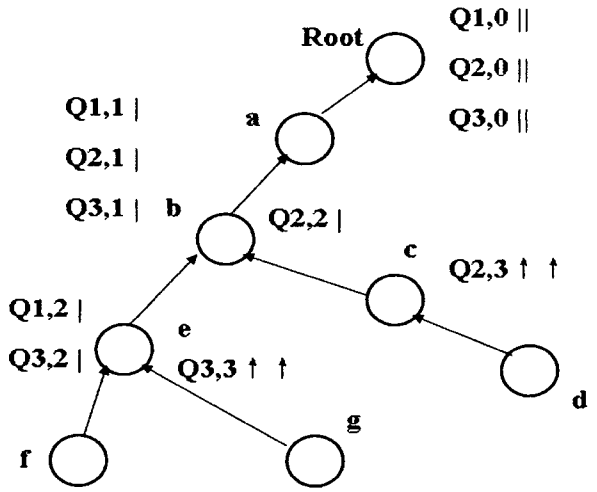


Figure A.4 AxisView of Afilter

The following list presents the main features of matching performed by Afilter that distinguish it from other filters:

- The example illustrates the matching of request from the end to the front performed by Afilter.
- As each tag is read in, it is pushed into (start tag) or popped from (end tag) the corresponding stack that has the same name as the tag. Note that Afilter uses a set of named stacks to store matched states. When each start tag is read in, it looks up its parent state in the directed graph and creates a pointer in the stacks to the corresponding state object.
- Looking at the stacks for the Afilter to accommodate incoming tags from document steam, it is evident they are the same as the Yfilter stack that is laid down and separated by using tag name instead by level.
- Afilter does not mention handling complex query, so it needs a technique similar to that used in Yfilter to verify if the three simple queries from Q match in the same place in the document.

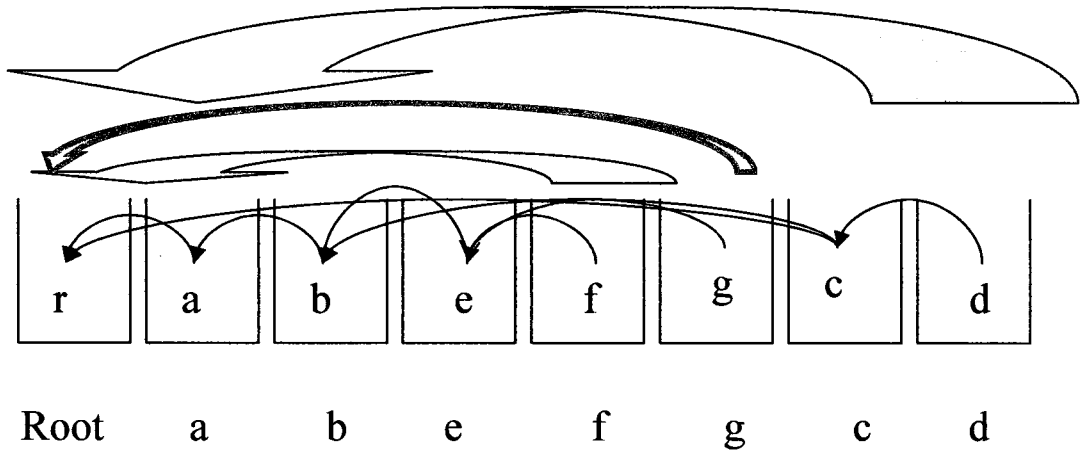
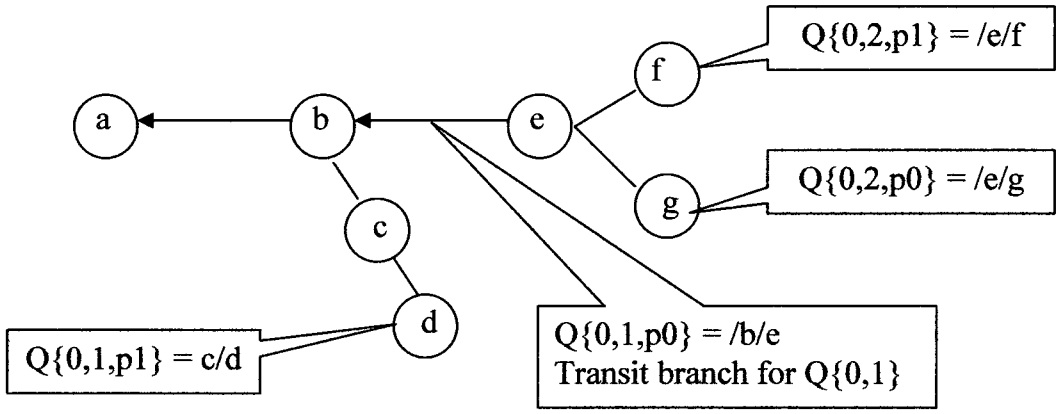


Figure A.5 Afilter Matching Process

A.4 Operations of Bfilter

In Bfilter, the query Q is represented by using sub-queries and is treated as a whole during matching. The matching starts when the last branch point of Q finds a match in the XML document. Figure A.6 shows the Query Index tree for Q. In Figure A.6, Q is represented by three sub-queries that are rooted at a, b and e respectively. The three sub-queries are represented as Q(0,0), Q(0,1) and Q(0,2), respectively. The first number in the brackets is the index of the complex query Q and the second number is the index of the sub-query. Please refer to Section 3.2 for details.



Q = a//b[/c/d]/e[/g]/f
Q{0,0} root = a; Q{0,1} root = b; Q{0, 2} root = e
Note: Q{0,0} is not presented. Since there is only one complex query in this case study, Q's index is 0

Figure A.6 Bfilter Matching Process

- Q = a//b[/c/d]/e[/g]/f has branch point *b* and *e*. Bfilter matches this query by first finding the match of the last branch point *e*.
    - Branch point detecting: a sequence of end tags follows a start tag that has a higher level than the one in the runTimeBranch Stack.

- As each tag is read in, it is pushed into (start tag) or popped from (end tag) the runTimeBranch Stack. When e is popped out from the stack, Bfilter starts matching e and finds out that Q{0,2,p0} has no candidates for matching, so the sub-query Q{0,2} is marked as unmatched. In this case, the document has no e branch point, and no further matching will be performed.

Appendix B: Sequence Diagrams for Bfilter

Appendix B presents the main sequence diagrams for Bfilter to illustrate how Bfilter operates.

B.1 Sequence Diagram — main Method

Figure B.1 shows the sequence diagram for the main method in the class EXfilter_Bfilter. The class EXfilterBasic_Bfilter inherits from the class EXfilter_Bfilter and implements the methods called from the main method in the class EXfilter_Bfilter. The class Profiler4 is used to save and report the filtering results.

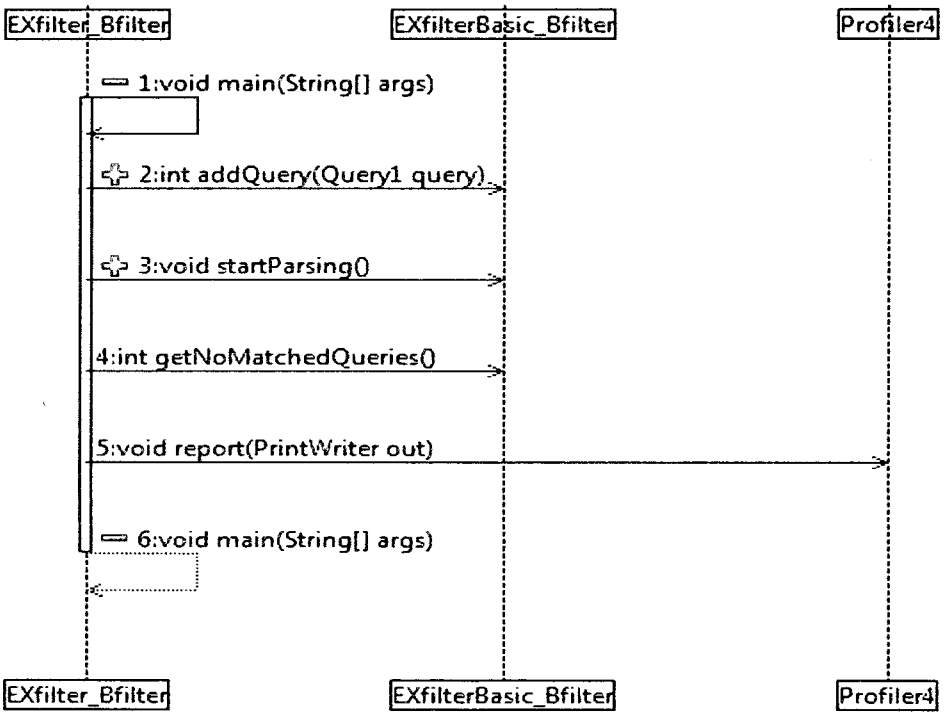


Figure B.1 Sequence Diagram for the Main Method

The main method is the entry point to execute the filtering program. When the program starts, the user queries are read in from plain text and the query objects are

created from them. These queries are indexed in the queryIndex Tree by calling the addQuery method. Then the startParsing method is called to perform filtering. Finally the matching results are collected and saved by calling the getNoMatchedQuery method and report method.

B.2 Sequence Diagram — addQuery Method

Figure B.2 shows the sequence diagram for the addQuery method in Figure B.1. Class QueryIndexBasic_Bfilter implements a queryIndex Tree, a hash table-based data structure, to index queries. Class HashEntryBasic2 is used to build the data structure. The addQuery method adds each sub-query of a complex query one by on into the queryIndex Tree and saves branch points of the query in corresponding nodes in the tree.

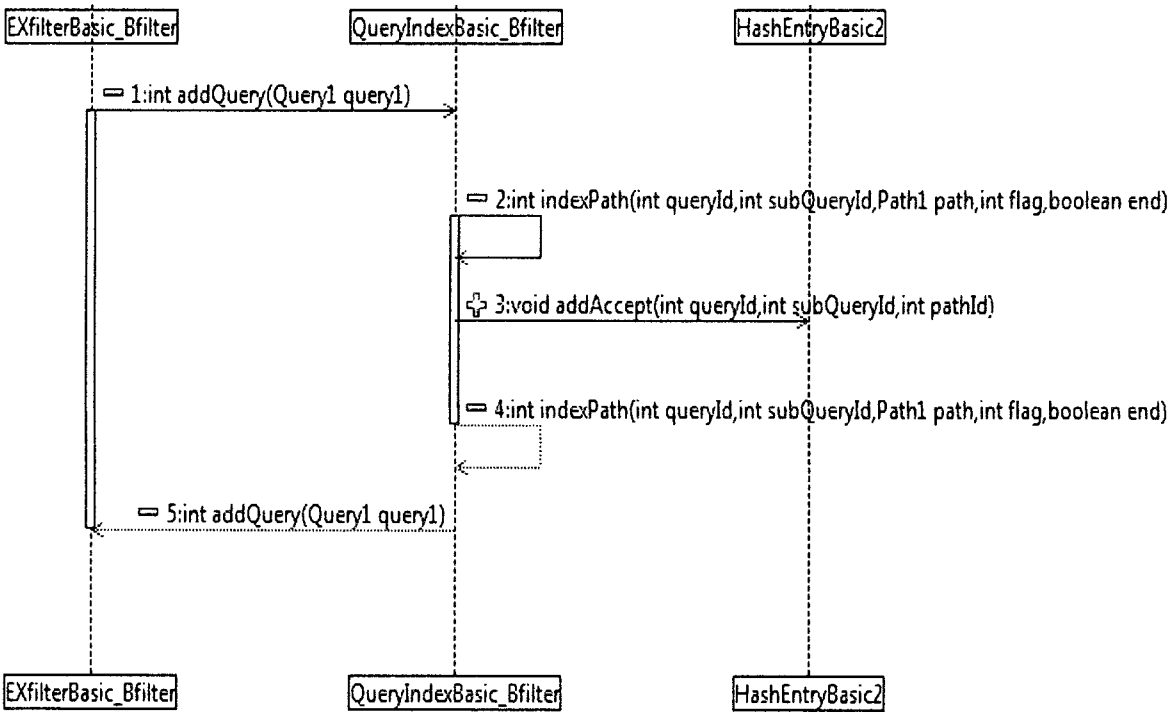


Figure B.2 Sequence Diagram for the addQuery Method

B.3 Sequence Diagram — startParsing Method

Figure B.3 shows the sequence diagram for the startParsing method in Figure B.1. The startParsing method executes the corresponding event handlers based on the incoming event from the SAX event-based XML parser. In the beginning the start tag of an incoming document generates an event; the event triggers the startDocument event handler in Bfilter. When an event of start tag of an element is generated, the startElement handler is called; when an event of end tag of an element is generated, the endElement handler is called. When completed, the endDocument handler is called.

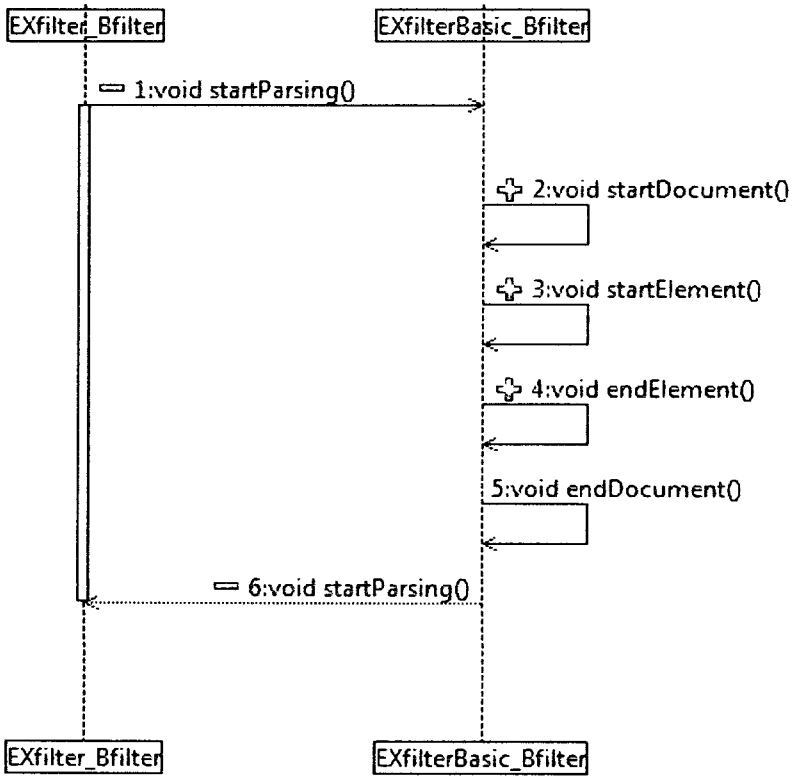


Figure B.3 Sequence Diagram for the startParsing Method

References

[1] Altinel, M. and Franklin, M. "Efficient Filtering of XML Documents for Selective Dissemination of Information," in *Proceedings of the International Conference on Very Large Data Base (VLDB)*, Cairo, Egypt, September 2000. Page(s): 53-64.

[2] Banerjee, S., Bhattacharjee, B. and Kommareddy, C., "Scalable Application Layer Multicast," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Pennsylvania, USA, August 2002, Page(s): 205–217.

[3] Barbosa, D., Mendelzon, A., Keenleyside, J. and Lyons, K. "ToXgene: A Template-Based Data Generator for XML," in *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, USA, June 6-7, 2002, Pages(s): 49-54.

[4] Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M. and Josifovski, V. "Streaming XPath Processing with Forward and Backward Axes," in *Proceedings of 19th International Conference on Data Engineering*, Bangalore, India, March 5-8, 2003, Page(s): 455-466.

[5] Bhargava, A., Kothapalli, K., Riley, C., Scheideler, C. and Thober, M., "Pagoda: a Dynamic Overlay Network for Routing, Data Management, and Multicasting," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, Barcelona, Spain, June 2004, Page(s): 170–179.

[6] Bloom, B. H. "Space/time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, Vol. 13, Issue 7, July 1970, Page(s): 422–426.

[7] Boone, P. "A Hybrid XML Filtering Engine for Publish/Subscribe Content-Based Routing", Technical Report, The School of Computer Science, Carleton University, January, 2007.

[8] Burstein, P. and Rizvi, S. "Peer-to-Peer Result Dissemination in High-Volume Data Filtering," University of California, Berkeley, http://www.cs.berkeley.edu/~kubitron/courses/cs294-4-F03/projects/rizvi_burst.pdf, last accessed on August 20, 2009.

[9] Cao, F. and Singh, J.P. "MEDYM: Match-early and Dynamic Multicast for Content-based Publish-subscribe Service Networks," in *Proceedings of the Sixth International Conference on Middleware (Middleware'05)*, Grenoble, France, November 2005, Page(s): 292–313.

[10] Chand, R. and Felber, P. "A Scalable Protocol for Content-Based Routing in Overlay Networks," in *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'03)*, Cambridge, MA, USA, April 2003, Page(s): 123-130.

[11] Chan, C.-Y., Felber, P., Garofalakis, M. and Rastogi, R. "Efficient Filtering of XML Documents with XPath Expressions," in *Proceedings of the 18th International Conference on Data Engineering*, San Jose, USA, February 26 - March 1, 2002, Page(s): 235–244.

[12] Chand, R. and Felber, P. "XNET: a Reliable Content-based Publish/Subscribe System Reliable Distributed Systems," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed System,* Florianpolis, Brazil. October 2004, Page(s): 264–273.

[13] Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D. and Candan, K. S. "Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents", in *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, Seoul, Korea, September 2006, Page(s): 283-294.

[14] Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D. and Candan, K.S. "Scalable Filtering of Multiple Generalized-Tree-Pattern Queries over XML Streams," *IEEE Transactions on Knowledge and Data Engineering,* Vol. 20, Issue 12, December 2008, Page(s): 1627–1640.

[15] Chen, Z., Jagadish, H.V., Lakshmanan, L.V.S. and Paparizos, S. "From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery," in *Proceedings of the 29th International Conference on Very Large Databases (VLDB)*, Berlin, Germany, September 2003, Page(s): 237–248.

[16] Chu, Y., Rao, S. and Zhang, H. "A Case for End Systems Multicast," *IEEE Journal on Selected Areas in Communication: Networking Support for Multicast/Special Issue,* Vol. 20, Issue 8, October 2002, Page(s): 1456–1471.

[17] Diao, Y., Altinel, M., Franklin, M., Zhang, H. and Fischer, P. M. "Path Sharing and Predicate Evaluation for High-Performance XML Filtering," *ACM Trans. Database Systems,* Vol. 28, No. 4, December 2003, Page(s): 467–516.

[18] Diao, Y., Rizvi, S. and Franklin, M. "Towards an Internet-scale XML dissemination service," in *Proceedings of the Thirtieth International Conference on Very Large Databases (VLDB)*, Toronto, Canada, August 2004, Page(s): 612–623.

[19] Fenner, W., Rabinovich, M., Ramakrishnan, K., Srivastava, D. and Zhang, Y. "XTreeNet: Scalable Overlay Networks for XML Content Dissemination and Querying

(Synopsis)," In *Proceedings of the International Workshop on Web Content Caching and Distribution (WCW'05),* French Riviera, France, September 2005, Page(s): 41–46.

[20] Kwon, J., Rao, P., Moon, B. and Lee, S. "FiST: Scalable XML Document Filtering by Sequencing Twig Patterns," in *Proceedings of the 31st International Conference on Very Large Databases (VLDB)*, Trondheim, Norway, August 2005, Page(s): 217–228.

[21] Langerman, S., Lodha, S. and Shah R. "Algorithms for Efficient Filtering in Content-Based Multicast," in *Proceedings of the 9th Annual European Symposium on Algorithms*, Lecture Notes in Computer Science, Vol. 2161, August 2001, Page(s): 428–439.

[22] Lee, S.-J., Gerla, M., and C.-C., Chiang, "On-Demand Multicast Routing Protocol," in *Proceedings of the IEEE International Conference on Wireless Communications and Networks (WCNC'99)*, New Orleans, LA, USA, September 1999, Page(s): 1298-1302.

[23] Moustafa, H. and Labiod, H. "A Multicast On-demand Mesh-based Routing Protocol in Multihop Mobile Wireless Networks," in *Proceedings of the 58th Vehicular Technology Conference*, Florida, USA, October 2003, Vol. 4, Page(s): 2192- 2196.

[24] Opyrchal, L., Astley, M., Auerbach, J., Banavar, G., Strom, R. and Sturman, D. "Exploiting IP Multicast in Content-Based Publish-Subscribe Systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, New York, USA, April 2000, Page(s): 185–207.

[25] Paris, G., Arrufat, M., Lopez, P. G. and Sanchez-Artigas, M. "An Application Layer Multicast for Collaborative Scenarios: The OMCAST Protocol," in *Proceedings of the Seventh International Conference on Networking*, Cancun, Mexico, April 2008, Page(s): 99–104.

[26] Perng, G., Wang, C. and Reiter, M. K. "Providing Content-Based Services in a Peer-to-Peer Environment," Carnegie Mellon University, Pittsburgh, PA, USA, 2004, http://www-serl.cs.colorado.edu/~carzanig/debs04/debs04perng.pdf, last accessed on August 20, 2009.

[27] Sax Project Organization. 2001. SAX: Simple API for XML. http://www.saxproject.org, last accessed on May 18, 2008.

[28] Segall, Bill and Arnold, David. "Elvin Has Left the Building: a Publish/subscribe Notification Service with Quenching," In *Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUC'97),* Brisbane, Australia, September 1997, Page(s): 243-255.

[29] Selcuk Candan, K., Hsiung W.-P., Chen, S., Tatemura, J. and Agrawal, D. "AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering", in *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, Seoul, Korea, September 2006, Page(s): 559 – 570.

[30] Transier, M., Fubler, H., Widmer, J., Mauve, M., Effelsberg, W. and Widmer, J. "Scalable Position-Based Multicast for Mobile Ad-hoc Networks," in *Proceedings of the 1st International Workshop on Broadband Wireless Multimedia: Algorithms, Architectures and Applications (BroadWim),* San Jose, USA, October 2004, Page(s): 173–180.

[31] Yan, T.W. and Garcia-Molina, H. "The SIFT Information Dissemination System," ACM Transactions on Database Systems, Vol. 24, Issue 4, December 1999, Page(s): 529–565.

[32] Yoneki, E. and Bacon, J. "Content-based Routing with On-demand Multicast," in *Proceedings of the 24th International Conference on Distributed Computing Systems,* Tokyo, Japan, March 2004, Page(s): 788–793.

[33] Yoneki, E. and Bacon, J. "Distributed Multicast Grouping for Publish/Subscribe over Mobile Ad Hoc Networks," in *Proceedings of the IEEE International Conference on Wireless Communications and Networking*, New Orleans, USA, March 2005, Vol. 4, Page(s): 2293– 2299.

[34] Zheng, Q.-H., Jiang, S., Zhang, F., Peng, T. and Chen, C. "TapMulti: A Scalable and Low-Delay Application-Layer Multicast Protocol on Tapestry Overlay Network," Information Technology Journal, Vol. 7, Issue 5, 2008, Page(s): 728–736.