Bloom Filter-based XML Packets Filtering for Millions of Path Queries

Xueqing Gong, Weining Qian, Ying Yan, and Aoying Zhou
Department of Computer Science and Engineering
Fudan University, Shanghai, P.R.China
{gongxq,wnqian,yingyan,ayzhou}@fudan.edu.cn

Abstract

The filtering of XML data is the basis of many complex applications. Lots of algorithms have been proposed to solve this problem[2, 3, 5, 6, 7, 8, 9, 11, 12, 13, 18]. One important challenge is that the number of path queries is huge. It is necessary to take an efficient data structure representing path queries. Another challenge is that these path queries usually vary with time. The maintenance of path queries determines the flexibility and capacity of a filtering system.

In this paper, we introduce a novel approximate method for XML data filtering, which uses Bloom filters representing path queries. In this method, millions of path queries can be stored efficiently. At the same time, it is easy to deal with the change of these path queries. To improve the filtering performance, we introduce a new data structure, Prefix Filters*, to decrease the number of candidate paths. Experiments show that our Bloom filter-based method takes less time to build routing table than automaton-based method. And our method has a good performance with acceptable false positive when filtering XML packets of relatively small depth with millions of path queries.*

1. Introduction

XML is quickly gaining dominance as a format for exchanging and storing semi-structured data. Plenty of information is represented in XML format and is delivered to end users or clients. For example, in a publish/subscribe system, users subscribe news or production information to their service providers. Service providers should filter all their information and deliver to users what they want. Such applications include stock and sports tickers, traffic information system, electronic personalized newspaper, and entertainment delivery. In such environments, new information is produced continually and the total amount of information is huge. Most of such information is represented in XML format and is processed as a stream. Furthermore, users' interest could vary with time. These applications must deal with the change of their users' interest efficiently. Therefore, the processing performance and queries maintenance are both critical for these applications.

Recently, lots of works have been proposed for XML filtering and stream processing[2, 3, 5, 6, 7, 8, 9, 11, 12, 13, 18]. All these works take a stream of XML documents as input, and compute path queries against the stream to identify the matches. In general, the documents in the stream are small packets, called as *XML Packets*. User's interests are represented as XML path queries in these works. In this paper, we focus on the scenario of XML data dissemination, where many XML path queries are preprocessed to build a routing table, and incoming XML packets are evaluated on the routing table for dissemination. There have been two types of works of solving this problem, automaton-based methods[2, 8, 12, 11, 9] and index-based method[5]. The former transforms XML path queries into an automaton, such as *NFA, DFA* or *Pushdown Machine*. Then, incoming XML packets are navigated through the automaton to identify the matched queries and relevant users. The latter combines XML path queries into a prefix tree and builds an element position index for the incoming XML packet. Then, the prefix tree is computed based on the index for the matched queries. In case the automaton or the index has been constructed, both of them can process incoming XML packets efficiently. However, in the scenarios where the number of XML path queries is huge or the set of XML path queries changes, the efficiency and the maintenance of routing table would be the bottleneck of systems.

This paper proposes a novel technology for XML packets filtering. We take an XML path query as a query string, and all query strings of one user are mapped into a Bloom filter by hash functions. The routing table is comprised of many Bloom filters. For each incoming XML packet, it is parsed to generate a set of candidate paths, while each candidate path is mapped to a bit-vector by the same hash functions to compare with the routing table. If a candidate path is determined existing in a user's Bloom filter, the related XML packet fragment is routed to the user. The main con-

Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)
1084-4627/05 $20.00 © 2005 **IEEE**

tributions of our work are the following:

- We present a novel technology of filtering XML packets, i.e. Bloom filter-based filtering. It can filter XML packets efficiently with relatively low false positive. In the scenario that the number of path queries is huge, our technology has obviously advantage in efficiency and routing table maintenance.

- To improve the performance of filtering, we introduce a new data structure, i.e. *Prefix Filters*. It can improve the filtering performance greatly by decreasing the number of candidate paths.

- We present an empirical study of Bloom filter-based and automaton-based filtering. Experiments show that the former has better performance when the number of path queries is huge and the depth of XML packets is relatively small.

1.1 Paper Organization

The rest of the paper is structured as follows. Section 2 is dedicated to some background knowledge and the problem statement. In Section 3 we present our basic approach of filtering XML packets using Bloom filters. Section 4 introduces prefix filters and the maintenance of routing table. Section 5 reports our experimental results. Section 6 presents related research works and Section 7 concludes the paper.

2. Background

In this section, we introduce XML packets and simple path queries that we process. An overview of Bloom filter is also presented followed by our problem statement.

2.1. XML Packets and Path Queries

An XML document can be represented as a rooted labelled tree, where each node corresponds to an element or a value. There is a root node for each XML document. For each element, there is a unique path from the root node to it which is called as its *Node Path*. The depth of a node path equals to the number of nodes in it. The maximal depth of all node paths is the depth of the XML document. In general, the depths of most XML documents transmitted in networks are not large. This paper focuses on the dissemination of XML packets, which are small depth XML documents .

Several query languages have been proposed for XML data processing, such as XQuery, XSLT and XPath. The basis of these query languages is path expressions. In this paper, we consider the simple path query which is described in definition 1.

Definition 1 (Simple Path Query): *A simple path query of length l has the form "$a_1 n_1 a_2 n_2 \ldots a_l n_l$", where each n_i is an element name or a wildcard symbol "$*$", and each a_i is either "$/$" or "$//$", which denote* parent-child *and* ancestor-descendant *axes respectively. l is the length of the path query.*

Given a simple path query, it may match many node paths with different lengths. For example, "$/A//*/D$" is a simple path query with length 3, it matches the following node paths: "$/A/B/D$", "$/A/B/C/D$", "$/A/C/D$", and so on. On the other hand, given a node path, it may also match many different simple path queries. For example, "$/A/B/C/D$" is a depth 4 node path. It matches the following simple path queries: "$//D$", "$/A//D$", "$//A/*//D$" and so on. To solve the problem of matching a node path and a simple path query, most traditional methods evaluate the simple path query on the node path node by node. We propose a different method to solve the problem. A simple path query can be seen as a query string, while many simple path queries can be produced from a node path. In other words, we can get a set of query strings from a node path. Then, the matching of a simple path query with a node path is equal to the problem of membership problem of the node path's query strings set. Thus, the string matching technology can be used.

2.2. Bloom Filters

A Bloom filter is a simple space-efficient data structure for probabilistic representation of a set that support membership queries. Bloom filters were introduced in the 1970's[4] and have been widely used in different applications, such as database applications[16, 17], web caching[10], intrusion detection and query filtering and routing.

A Bloom filter is essentially a bit-vector of length m used to efficiently represent a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements. All bits are set to 0 initially. Then, k independent hash functions, $h_1, h_2, \ldots, h_k$, are chosen. Each hash function has range from 1 to m. For each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. A particular bit may be set to 1 many times, but only the first operation has an effect. To check if an item y is in S, the bits at positions $h_1(y), h_2(y), \ldots, h_k(y)$ are checked. If any of them is 0, clearly y is not a member of S. If all $h_i(y)$ are set to 1, we assume that y is in S, although we are wrong with some probability. This is a *false positive*, where the Bloom filter suggests that an element y is in S even though it is not. It has been shown that the probability of a false positive is equal to $(1 - e^{-kn/m})^k$[4]. For many applications,

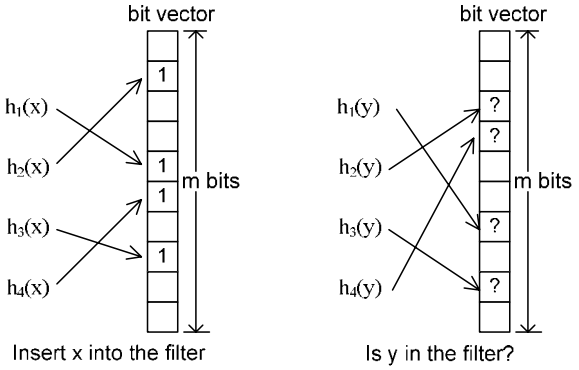


Figure 1. A Simple Bloom Filter with 4 Hash Functions

false positives may be acceptable as long as it is sufficiently small. Figure 1 provides an example of Bloom filter with 4 hash functions. The Bloom filter is a bit-vector of length 10. When we insert a value x into the Bloom filter, the bits of $h_1(x), h_2(x), h_3(x)$ and $h_4(x)$ are set to 1. If we want to know whether a value y is in the Bloom filter or not, the bits of $h_1(y), h_2(y), h_3(y)$ and $h_4(y)$ are checked.

One property of Bloom filters is that it is not possible to delete an element stored in the filter. Deleting a particular element requires that the corresponding positions determined by k hash functions in the bit vector be set to zero. This could disturb other elements stored in the filter which hash to any of these positions. In order to solve this problem, the idea of the *Counting Bloom Filters* was proposed in [10]. A Counting Bloom Filter maintains a counter for each bit in the Bloom filter. Whenever an element is added to or deleted from the filter, the counters corresponding to the k hash values are incremented or decremented, respectively. When a counter changes from zero to one, the corresponding bit in the bit vector is set to 1. When a counter changes from one to zero, the corresponding bit in the bit vector is set to 0. In fact, the counter maintains the number of elements that hashed to corresponding bit by any of the k hash functions.

2.3. Problem Statement

We define here the application scenario of XML filtering systems. There are some data producers in a network, which produce XML packets and send them to routing servers (Routers). Users subscribe their queries to routers. All routers are connected with each other, and every router subscribes all its user's queries to it's neighbors. Therefore, a router can be regarded as a user by its neighbors. A Router can consume incoming XML packets continuously, and determines which users the packets should be routed to.

An XML router may have lots of users, and every user has a unique identifier. In our scenario, all user queries are simple XPath queries. In other words, queries from a user compose a set of simple XPath queries. When a router receives an XML packet, it evaluates the query set on the XML packet for each user. In case there is a simple path query matched by the XML packet, the router forwards the corresponding XML packet or a fragment of the XML packet to the user. We formally define the XML filtering problem in Definition 2.

Definition 2 (XML Filtering Problem): *Given a router S and the user set of the router $U = \{u_1, u_2, \ldots, u_n\}$. For each user $u_i \in U (1 \leq i \leq n)$, it subscribes a set of simple path queries Q_i to the router. For each incoming XML packet p, the router S test every query set $Q_i (1 \leq i \leq n)$. If there is a path query q matching p in Q_i, user u_i's query set Q_i is matched by p. The XML packet p is routed to all matched users.*

One challenge here is that the number of users is huge and each user may have a large number of queries. There should be an efficient data structure in the router to represent all the queries. Another challenge is that user's interest will vary with time. The data structure should supports the update of queries. In this paper we use Bloom filters to represent users' queries, which are space-efficient data structures. In the scenario that users' queries are updated frequently, it is easy to maintain users' queries with counting Bloom filters.

3. Basic Approach

In this section, we introduce how a router maintains its routing table, and how to process incoming XML packets to produce candidate paths. After that, the method of routing XML packets is presented.

3.1. Routing Table

A router must maintain a routing table, which is used to determine how XML packets are to be forwarded. Each entry of the routing table corresponds to a user of the router. In this paper, each routing table entry is a Bloom filter. We use the Bloom filter to store the query set of a user. Every user's query is predigested to a simple path query and is regarded as a *Query String*. Then, all the query strings from the same user are mapped to a m-length Bloom filter by k hash functions. Figure 2 shows an example of how to build a routing table entry. The routing table entry is a Bloom filter of length 8. There are four queries, Q_1, Q_2, Q_3 and Q_4, which come from the same user. These queries are mapped into the Bloom filter by 2 hash functions, $h_1()$ and $h_2()$.

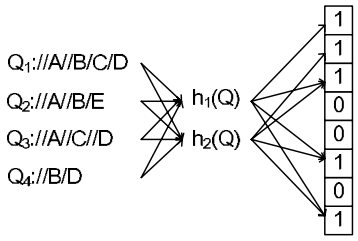


Figure 2. Building a Routing Table Entry

To support queries' update, we can use *Counting Bloom Filters* as the routing table entries. When a query is subscribed by a user, it is mapped to the user's routing table entry by k hash functions. For each position, the corresponding counter increments by 1. If the original value of the counter is zero, the bit of the routing table entry is set to 1. When a query is unsubscribed by a user, it is also mapped to the user's routing table entry by the same k hash functions. The difference is that the counters corresponding to the positions decrement by 1. A bit of the routing table entry is set to 0 when the corresponding counter changes to zero. The situation in which a user modifies a query can be treated as an unsubscription followed by a subscription.

Because Bloom filters are space-efficient data structures, one advantage of using Bloom filters as routing table entries is that the storage of routing table will be compact especially for large query sets. For example, given $10,000$ users and $10,000$ queries for each user, we use $100,000$ bits Bloom filters as routing table entries along with 8 hash functions. In such situation, the false positive is about 0.00846, which is acceptable for most applications. The size of the routing table is as much as 120 Megabytes. It is easy to be maintained in main memory nowadays.

3.2. Candidate Path

An XML packet can be viewed as a rooted label tree. There is a unique node path from root node to each element. However, a node path can match many simple path queries. We can generate candidate paths from a node path by reassembling its elements. Every incoming XML packet will produce lots of candidate paths. A candidate path is a simple path query expression, which is constructed based on a node path of the XML packet. The axes of a candidate path are child (/) or descendant-or-self (//). The element name of each candidate path step is an element name of the node path. The processing of incoming XML packets is straightforward. An input XML packet is parsed by a SAX parser to produce a stream of events. Each event is one of *Start-Document, Start-Element, End-Element, Text* and *End-Document*. [1] Our Bloom filter-based filtering system processes incoming XML packets by handling these events. During the process of parsing an XML packet, the current node path is stored in variable $CurNPath$. The system maintains a candidate path list for each level i, presented as L_i. L_i includes all candidate paths of length i belonging to current node path. Each candidate path in L_i has an attribute d, which is the depth of its corresponding element in the current node path. When a *Start-Document* event is met, it means a new XML packet appears, the system initializes environment. An *End-Document* event means the end of current XML packet, the system clears environment. *Start-Element* and *End-Element* events identify the beginning and the end of an element respectively. When a *Start-Element* event is met, the system generates a set of candidate paths corresponding to the new element. When an *End-Element* event is met, the system removes candidate paths related to the element from L_i. The algorithms corresponding to *Start-Element* and *End-Element* events are given in Algorithm 1 and 2 respectively. A *Text* event identifies that a text string is met. A text string can be processed as a special element. A *Text* event is treated as a *Start-Element* event followed immediately by an *End-Element* event. Here we ignore the algorithm of *Text* event processing.

Algorithm 1 StartElement()

Input: Element Name E, Current Depth d
Output: A list of candidate path O

1: **if** $d \equiv 1$ **then**
2: append $\{$"$/E$", "$//E$", "$/*$", "$//*$"$\}$ to L_1, O;
3: **else**
4: append $\{$"$//E$", "$//*$"$\}$ to L_1, O;{This "$//*$" has different depth with above.$\}$
5: **for** $i = 1$ **to** $d - 1$ **do**
6: **for all** candidate path $p \in L_i$ **do**
7: **if** $p.d \equiv (d-1)$ **then**
8: append $\{$"p/E", "$p//E$", "$p/*$", "$p//*$"$\}$ to L_{i+1}, O;
9: **end if**
10: **if** $p.d < (d-1)$ **then**
11: append $\{$"$p//E$", "$p//*$"$\}$ to L_{i+1}, O;
12: **end if**
13: **end for**
14: **end for**
15: **end if**
16: Output O;
17: return;

[1] There are other types of nodes in a XML packet, such as attributes. As a attribute node can be treated as a special element node, we do not distinguish among elements, attributes, and other types of nodes in this paper.

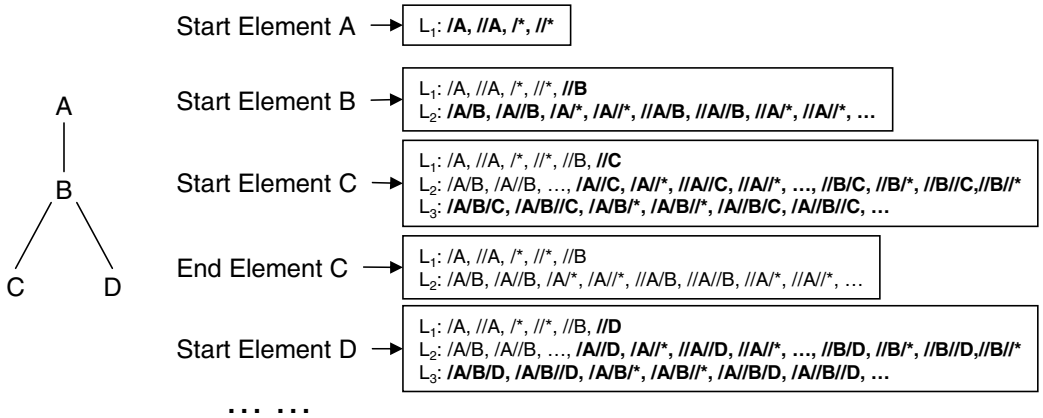


Figure 3. An Example of Candidate Paths Generating

Algorithm 2 EndElement()

Input: Element Name E, Current Depth d
Output:

1: **for** $i = 1$ **to** d **do**
2: **for all** candidate path $p \in L_i$ **do**
3: **if** $p.d \equiv d$ **then**
4: remove p from L_i;
5: **end if**
6: **end for**
7: **end for**

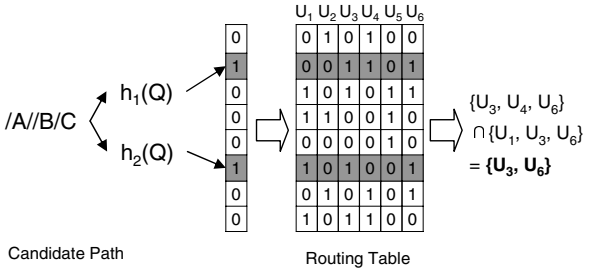


Figure 4. An Example of Comparison

Figure 3 gives an example of the process of candidate paths generating. Given an XML packet with four elements, A, B, C and D, A stream of events is produced as shown in Figure 3. The states of all candidate path lists are also shown with corresponding events. The candidate paths written in bold font are new, which are inserted during the event processing. The main problem of this method is that the number of candidate paths is huge. For each element E of depth $d(d > 1)$, it produces two candidate paths of length 1, $\{\text{“}//E\text{”}, \text{“}//*\text{”}\}$; and it can be combined with existent length 1 candidate paths to produce candidate paths of length 2, and so on. It is clearly that the number of candidate paths increases exponentially with the depth of current node path during the processing of XML packets. To solve this problem, we introduce a new data structure, *Prefix Filters*, to decrease the number of candidate paths in Section 4.

3.3. XML Packets Filtering

During parsing XML packet, we can get a set of new candidate paths when meeting a *Start-Element* event. For each candidate path of the set, it is mapped to a m-length bit vector by k hash functions. The parameters m and k are the same with parameters used to build routing table. Then, we compare the result bit vector with routing table to determine which users are interested in the XML packet. There are two types of filtering applications, one route the whole XML packet to users, another route a fragment of the XML packet to users. In the former system, all users' IDs who are interested in the packet are recorded, and the packet is forwarded to these users after it is parsed. In the latter, the current node path $CurNPath$ (described in Section 3.2) is forwarded to matched users during the XML packet is parsed.

Figure 4 gives an example of comparing a candidate path's bit-vector with routing table. The main process of the comparison are bit-wise operations. Therefore, it is efficient. Another benefit of this technology is that we can mark every user with the number of candidate paths matched with its queries. This number can be used as the weight of the user. The system can determine which user has high priority to get the packet based on users' weight.

This routing technology is approximate because of the existence of false positive. The probability of a false positive is equal to $(1 - e^{-kn/m})^k$[4] according to theoretical analysis, in which k is the number of hash functions, n is

the number of queries stored in a Bloom filter and m is the length of the bit-vector. As mentioned above, a system can route a whole XML packet or a fragment to users. In the former scenario, the system route a whole XML packet to a user mistakenly when all matched candidate paths are false positive. The probability of such conditions are met is low. In the latter scenario, the system can adjust the size of a Bloom filter and the number of hash functions to gain relatively low false positive, according to the number of queries. For most applications, such false positive's probability is acceptable.

4 Prefix Filters and System Architecture

In this section, we first introduce *Prefix Filters*, which are used to decrease the number of candidate paths. Then, a high-level architecture of an *Bloom Filter-based XML Filtering* system is presented.

4.1 Prefix Filters

When we generate candidate paths using the basic approach, every new incoming element must be combined with existent candidate paths. Thus, the number of candidate paths increases exponentially with the depth of current node path of the XML packet. It becomes the bottleneck of our system. It is obvious that there are many redundant candidate paths in all candidate paths. For each length n simple path query (query string), there are length $1, 2, \ldots, n$ prefix path query strings. If a length l candidate path does not match any length l prefix strings of all users' queries, all the candidate paths beginning with it will not match any user's query. According to it, we construct *Prefix filters* to decrease the number of redundant candidate paths.

Given the set of all users' query strings Q, d is the maximal length of query strings in the set. For each query string, there are different length query prefixes. For example, query string *"/A//B/*"* has length 2 prefix string *"/A//B"* and length 3 prefix string *"/A//B/*"*. There is a prefix filter $L_i (1 \leq i \leq d)$ for length i prefix strings. Prefix filter L_i is a Bloom filter storing length i prefix strings of all users' queries. During parsing an XML packet, every new produced candidate path of length l will be checked on prefix filter L_l. If it is not in the prefix filter, none of all user's queries will match it. The system will throw away this candidate path.

Figure 5 gives an example of prefix filters. there are four queries in Figure 5(a), i.e. Q_1, Q_2, Q_3 and Q_4. The maximal length of these queries is 4. Thus, four prefix filters L_1, L_2, L_3 and L_4 are constructed as shown in Figure 5(a), which are used to store different length query prefixes respectively. In Figure 5(b), an XML packet is parsed to generate candidate paths. When the *Start-Element* event of element A is met, the system generates four candidate paths of length 1, i.e. *"/A"*, *"//A"*, *"/*"* and *"//*"*. Among these candidate paths, only *"//A"* is in prefix filter L_1. So the system keeps the candidate path *"//A"* and throws away the others. When the *Start-Element* event of element B is parsed, it is just combined with *"//A"* to generate candidate paths of length 2. For the *Start-Element* events of elements C and D, the processing are similar.

4.2 System Architecture

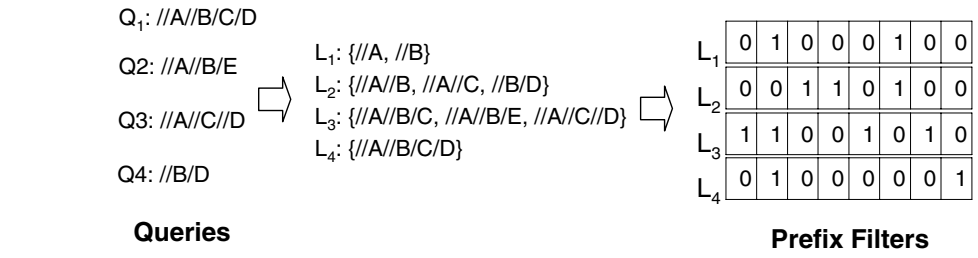
Our Bloom filter-based XML filtering system can work as an *XML Switch* in a content-based XML routing network described in [19]. XML packets are generated at certain source nodes in the network. Then, these packets are routed through the network's servers to end users. Thus, our system's user can be an end user or a neighbor router.

Figure 6 gives the architecture of our Bloom filter-based XML filtering system. The system maintains two data structures, *Routing Table* and *Prefix Filters*. Each user has a unique identifier and an entry in routing table, and each routing table entry is a Bloom filter (or counting Bloom filter). Prefix filters are also the set of Bloom filters corresponding to different level prefix strings of all queries. The length of a Bloom filter is determined by data it stores. Therefore, these Bloom filters can have different lengths. There are two main sets of inputs to the system: user queries and XML packets. An incoming XML packet is parsed by a SAX parser to generate a stream of events at first. When a *Start Element* event is met, the *Candidate Path Producer* keeps the current node path with the element and produces a set of candidate paths. These new candidate paths are stored and tested based on routing table by *Router*, and corresponding node path is forwarded to matched users. When an *End Element* event is met, the *Candidate Path Producer* throws away the candidate paths corresponding to the element.

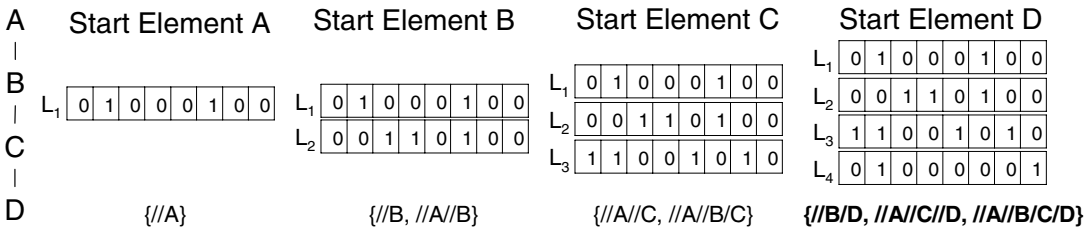
The aim of our system is to disseminate small size XML packets. For a large size XML document, it also can be processed if we divide it into many fragments.

5 Experiments

There have been many works on XML packets filtering now, and most of which are based on automaton technology. For the purpose of evaluating the performance of our Bloom filter-based XML filtering technology, we implement an automaton-based XML filtering system referencing *Y-Filter* described in [8, 5], which have the same functions as our system. We implemented both systems in C++, and shared as much code and data structures as possible for a fair comparison, such as the code of parsing XML packets. All of the experiments were performed on a PC machine



(a) Building Prefix Filters



(b) Using Prefix Filters

Figure 5. An Example of Prefix Filters

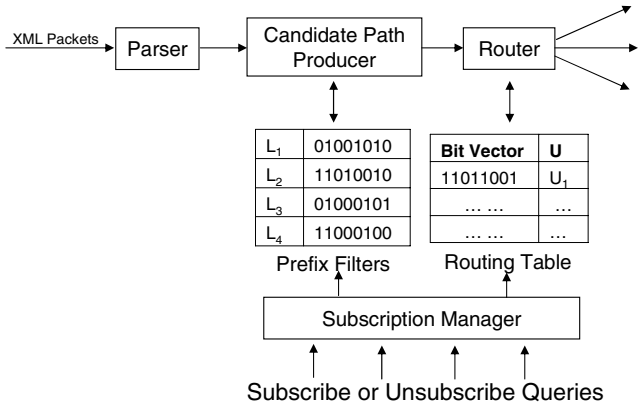


Figure 6. System Architecture

with Pentium IV 2.4 Ghz processor and 512 MB memory, running Windows XP professional operating system.

In the experiments, we used MD5[1] algorithm to get independent hash functions. MD5 is a cryptographic message digest algorithm, and usually is used in digital signature applications or as a way of verifying data integrity. It takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. To build k hash functions, we divide 128-bits MD5 signature of query strings into k groups, and each group is a hash function value with $128/k$ bits. We performed three types of experiments. The goal of the first set of experiments is to show the performance of Prefix filters. The second set of experiments focuses on the time of building routing table and the size of routing table. We compare Bloom filter-based routing table with automaton-based routing table for different sizes query sets. In the third set of experiments, we compare the routing times of Bloom filter-based and automaton-based filtering. Table 5 gives a summary of parameters used in the experiments.

5.1 XML Packets and Queries

In the experiments, we use DBLP[15] dataset as the source of XML packets. DBLP dataset is an XML document, including information about papers, thesis, books and authors. Each paper's information is represented as a

Table 1. Parameters of the experiments

Parameter	Range
# of hash functions	4-16
Size of Bloom filters	1000-1000000
# of queries per user	100-1000
# of users	100-1000
Depth of XML packet	2-5
Percentage of Matched Queries	10%-90%

fragment of the XML document. We take each such fragment as an XML packet. To get XML packets of different depth, several XML packets are combined into a single packet by replacing some citing papers with its actual XML information. In our experiments, an XML packet is the minimal unit of processing. We select part of DBLP dataset as our experimental data for convenience. The resulting XML packets set includes about 10,000 papers' information, and the range of depth of XML packets is from 2 to 5.

For the simple path queries, we developed a query generator, which takes element name sets and corresponding levels as input, and produces simple path queries of different users according to a set of workload parameters. Parameter *UserNum* determines the number of users, and each user has *QueryNum* queries. For a given XML packet, there is *PMQ (Percentage of Matched Queries)* percent queries matching the XML packet in each user's query set. The length of queries distribute uniformly between 1 to 6. There are not two same queries in any user's query set, and any two users do not have the same queries.

5.2 Performance of Prefix Filters

When prefix filters are used, they can decrease the number of candidate paths. We take two types of experiments to validate the performance of prefix filers.

- One experiment reports the relative performance of candidate paths produced *with prefix filters* with respect to *without prefix filters* (i.e., we divide number of candidate paths with prefix filters by that without prefix filters). Hence, the lower ratios means that the performance of prefix filters is better. We count the number of candidate paths for each incoming XML packets, and take the average number of $1,000$ XML packets as the experimental data.
- Another experiment compares the filtering performance of the system with and without prefix filters. similarly, we record the filtering time of $1,000$ XML packets and take the average as the experimental data.

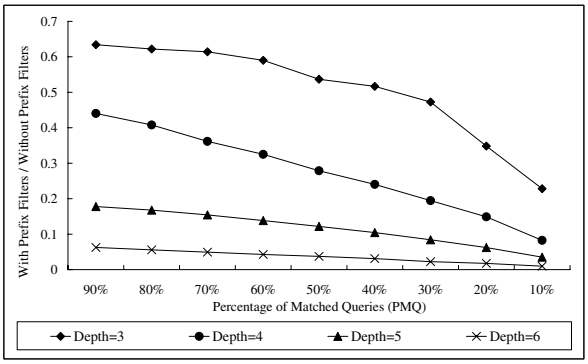
In the former experiment, we fix the number of queries per user to 200 and the number of users to 200. Therefore, the total number of queries is $40,000$. According to theoretical analysis, we set the number of hash functions and Bloom filters' size to 16 and $4,200$ respectively for relatively low false positive ($4.27e-05$). The parameter PMQ is the abbreviation of *Percentage of Matched Queries*, which means that there are PMQ% queries matched for each incoming XML packet. It varies from 90% to 10%. The main factor that determines the number of candidate paths is the depth of input XML packet. We do the experiment with XML packet's depth changing from 3 to 6. Figure 7(a) gives the result of this experiment. The ratio decreases along with the parameter PMQ. It is consistent with our analysis. When there are less matched queries, the number of candidate paths which pass the prefix filters is smaller. Therefore, the new element produces less new candidate paths by combining with existing candidate paths. When the depth of incoming XML packet increases, prefix filters are used at each level. Thus, the ratio will increase along with the depth of XML packet. The experimental result also proves it as shown in Figure 7(a). For example, when PMQ equals 50%, prefix filters throw away about half of candidate paths for depth 3 XML packets; however, only 3% candidate paths are kept for depth 6 XML packets.

For the latter experiment, the parameter PMQ is fixed as 50%. The number of users varies from 100 to $1,000$, and the number of queries for each user is 200. The number of hash functions and Bloom filters' size are set to 16 and $4,200$ respectively. From the experimental result shown in Figure 7(b), we know that prefix filters can improve the filter performance of the system greatly. The ratio of (Without Prefix Filters / With Prefix Filters) increases along with the number of users. When a candidate path is generated, it is mapped into a bit-vector. The bit-vector must be compared with each user's routing table entry. Therefore, prefix filters can save more comparing time when the number of users is larger.

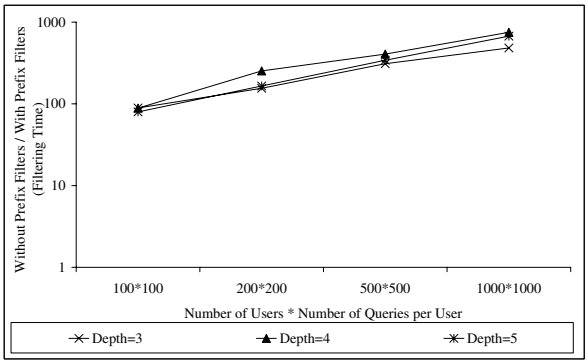
5.3 Building Routing Table

For a real application, the building time and size of routing table determine its capacity and ability of processing. In this set of experiments, on the one hand, we compare above measurements of our Bloom filter-based filter with and without prefix filters. On the other hand, we compare Bloom filter-based filter with automaton-based filter. The number of hash functions and size of Bloom filters vary with the number of queries per user to get a relatively low probability of false positive (about 0.5%).

We first present experimental results of Bloom filter-based filter comparing *with* and *without* prefix filters. Figure 8 shows the comparison of building time. The number of users varies from 100 to $1,000$, and each user's queries varies also from 100 to $1,000$. So the total number



(a) The Number of Candidate Paths



(b) Filtering Time

Figure 7. Performance of Prefix Filters

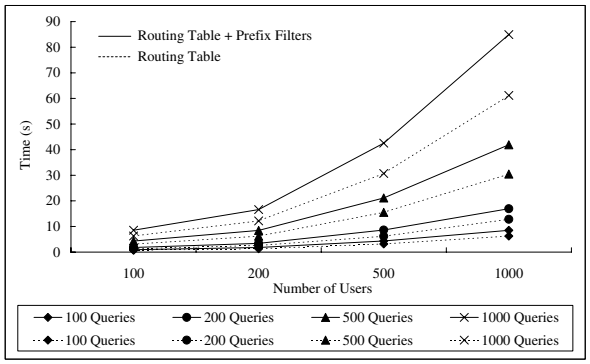
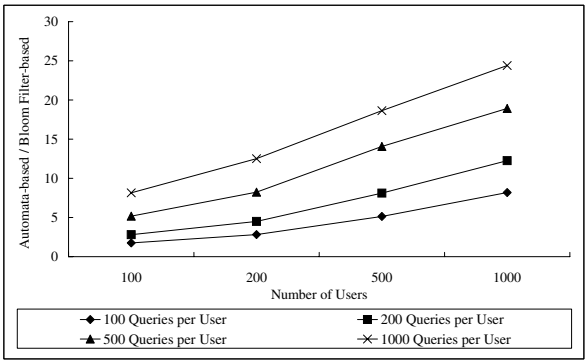


Figure 8. Building Time: With vs. Without Prefix Filters

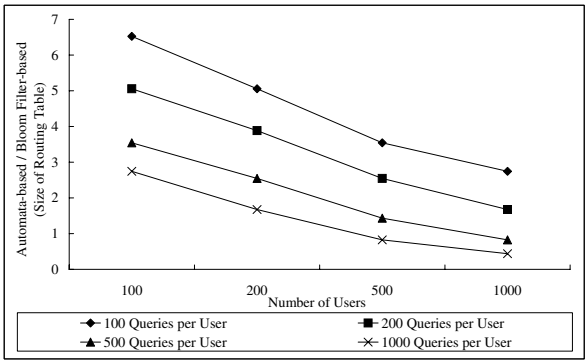
of queries varies from $10,000$ to $1,000,000$. In general, when the number of a user's queries is larger, the time of constructing routing table entry corresponding to the user is longer. More users means that the time of building the whole routing table is longer. As we can see in Figure 8, the building time has approximately linear relationship with the number of users. The building time of prefix filter increases along with the number of all queries.

We now report experimental results of comparing *Bloom filter-based* against *Automaton-based* filtering. Figure 9(a) and 9(b) give the results of building time and routing table's size respectively. Here, the building time of Bloom filter-based includes the time of building prefix filters. When the number of queries increases, Bloom filter-based technology is much more efficient than automaton-based method for building routing table. For example, the building time of automaton-based is almost 25 times as Bloom filter-based, when the number of all queries is $1,000,000$. The reasons for this situation are as follows. During the process of building routing table, automaton-based technology constructs a prefix tree[5] first. Then it transforms the prefix tree into an automaton. The latter step is time consuming. On the contrary, Bloom filter-based technology just hash each query in to a bit-vector. The routing table's size of Bloom filter-based system is determined by two factors, i.e., the number of users and the number of queries for each user. The former determines the number of entries in the routing table; the latter determines the size of each routing table entry. Given the two factors, we can compute the size of routing table. For example, the number of users is 500 and each user has $1,000$ queries, the routing table's size is about 610 KB ($1000*10*500 = 5,000,000$ bits. A Bloom filter's size is 10 times of the number of queries per user). The size of prefix filters is relatively stable. There are only several different levels prefix filter, and each prefix filter's size can be set to a large number for good performance. For example, we take each prefix filter's size as $10,000,000$. The routing table's size of automaton-based system is determined by the number of all queries and the number of common prefix shared by many queries. According to the method that we generate simple path queries, further queries will share common prefix when the number of queries increases. We get the routing table's size of automaton-based system by counting the number of state of the NFA. According to Figure 9(b), we know that the ratio of (automaton-based / Bloom filter-based) decreases when the number of queries increases. When the number of queries is large, there will be more queries share common prefix. On the contrary, a Bloom filter must increase its size to keep the probability of false positive stable.

Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)
1084-4627/05 $20.00 © 2005 **IEEE**



(a) Building Time



(b) Routing Table's Size

Figure 9. Bloom Filter-based Method vs. Automata-based Method

5.4 Filtering XML Packets

In this set of experiments, the filtering performances of Bloom filter-based and automaton-based are compared. The metric is the average time of filtering $1,000$ XML packets.

First, we vary the number of users from 100 to $1,000$ with 500 queries for each user, and select proper size of Bloom filters and number of hash functions to guarantee the relatively lower false positive. The filtering times of $1,000$ XML packets of depth 3 are shown in Figure 10(a). It is obviously that the automaton-based method consumes more time. It is also interesting that the time used by Bloom filter-based remains relative stable with the growing number of user. We can get the reason from the process of filtering XML packets. In automaton-based method, when the number of users gets larger with the same amount of queries each, the total number of queries becomes larger. Thus, the NFA gets larger accordingly. Each XML packet should test many states to determine whether it is matched or not. Thus the filtering time increases along with the number of users greatly. In Bloom filter-based method, as the increasing of users, the routing table get larger, but the bit-wise operations are very efficient. Therefore, the increasing size of routing table will not affect the time of Bloom filter-based much. We learn that our Bloom filter-based method run more stable as the increase number of user from this experiment.

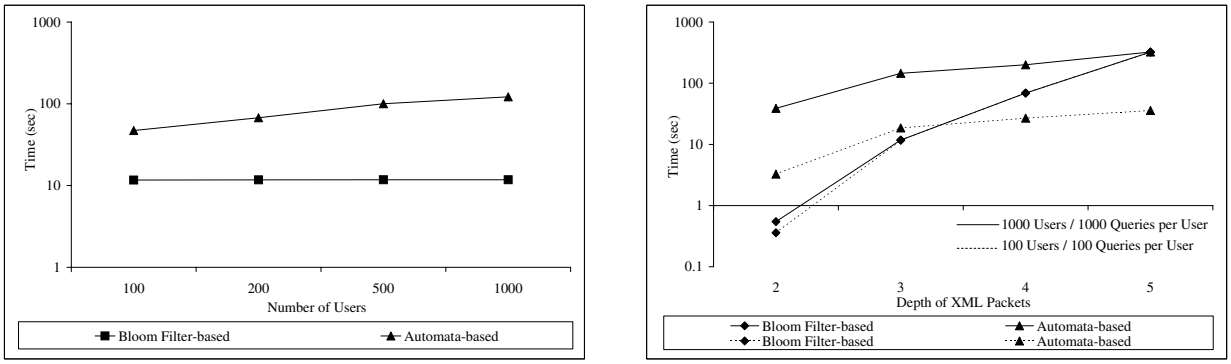
Then, we change the depth of XML packets and see the performance of the two methods for XML packets of different depths. The results are shown in Figure 10(b). The depth of XML packets varies from 2 to 5. Two sets of queries are 100 users with 100 queries each and $1,000$ users with $1,000$ queries each. It is obviously that when the depth increases the filtering times of two method increase accordingly. In each method the routing time grows with the increasing of number of queries. But the increasing rates of routing time of the two methods are different. Bloom filter-based method increases much more rapidly than automaton-based as the depth goes larger. When the depth of XML packets is 2 and there are $1,000$ users with $1,000$ queries each, the filtering time of our method is much smaller than automaton-based. But when the depth goes to 5, the two methods take almost the same time. Unluckily, when there are 100 users with 100 queries each, the routing time of our method exceeds the automaton-based method when the depth is bigger than 4. The reason is that when XML packets get deeper the number of candidate paths becomes much larger and consumes much filtering time. And the depth has relatively smaller effects on automaton-based method.

From the above experiments, we get the conclusion that our Bloom filter-based method use less building routing table time and less filtering time when processing millions of queries and XML packets with relative small depth.

6 Related Work

We compare briefly our work with related approaches regarding XML query processing, XML filtering and the use of Bloom filters.

In the context of XML data, XML filtering and XML stream processing have attracted much research attention. Lots of works have been proposed for XML filtering and stream processing[2, 6, 8, 11, 12, 3, 18, 9, 7, 13, 5]. Most of these works develop their own data structure to represent multiple path queries, then, input XML documents are navigated through the data structures for matching. For example, *XFilter*[2] builds a *Finite State Machine (FSM)* for each path query and links all the FSMs by a query index. When an XML document comes, all path queries can be processed simultaneously. *YFilter*[8] employs a single *Non-*



(a) 500 Queries per User

(b) 10, 000 Queries vs. 1, 000, 000 Queries

Figure 10. Filtering XML Packets: Bloom Filter-based vs. Automata-based

Deterministic Finite Automaton (NFA) to represent all path queries by sharing the common prefixes of the paths. Incoming XML documents drive the execution of the NFA to produce matched elements. *Xtrie*[6] devides each path query into sub-strings that only contain parent-child ("/") axis, and indexes all sub-strings by a trie-based data structure. Therefore, the processing of common sub-strings can be shared by queries. *Index-Filter*[5] presents all path queries as a prefix tree, and builds a position index of elements for each incoming XML document. Then, the prefix tree is evaluated on the index for matched elements. In [11], all path queries are combined into a single *DFA*. It is more efficient than *Y-Filter* when filtering input XML documents, however, the building time of *DFA* is more longer than other data structures. *XPush Machine*[12] is another similar system, which uses a single deterministic pushdown automaton to present all path queries. Most of above systems compute results by navigating input XML documents through query structures element by element. Our Bloom filter-based system takes a novel method differing from them. In our system, each path query is treated as a query string, and all query strings of one user are stored in a space-efficient data structure, a Bloom filter. Input XML documents are parsed to generate candidate paths. When a candidate path is approved existing in the Bloom filter, corresponding node path is routed to the user. Here, the problem of matching path queries with XML document is converted into a membership problem.

A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries, which is introduced in the 1970's[4]. It has been heavily used in various applications, such as spell check, summary cache, resource routing, logest prefix matching, multicast routing, etc. In the field of database applications, Bloom filters have been used to implement efficient join algorithms for distributed query processing[16, 17]. Recently, Bloom filters are used in a *Peer-to-Peer (P2P)* network to summarize the content of XML documents stored on a peer node[14]. Reference [14] aims to route path queries based on node's content. It proposes *Multi-level Bloom Filters* to maintain information about the structure of documents on a node. Its multi-level Bloom filters consists of two types of Bloom filters, *Breadth Bloom Filter (BBF)* and *Depth Bloom Filter (DBF)*. All the elements with depth i in an XML document are stored in a simple Bloom filter, denoted as BBF_i. DBF_i is also a simple Bloom filter, which stores all node paths of length i. Path queries are evaluated on BBFs and DBFs. Peer nodes in the network are organized hierarchically by clustering together nodes with similar content. Similarity between nodes is related to the similarity between their multi-level Bloom filters. The top level of the hierarchies are root nodes. When a path query is issued at a node, it is routed to the nodes whose documents can match the query. Our work is part of a routing system such as the system in [19]. The main difference between [19] and [14] is content being routed. In [19], each node is an XML router, and XML documents are routed through the network. Our Bloom filter-based filtering system can work as an XML switch described in [19]. In our work, Bloom filters are used to store path queries instead of XML documents. To gain better performance, we also use multi-level Bloom filters (Prefix Filters) to store different length path queries.

7 Conclusions and Future Work

In this paper we introduced a novel technology of filtering XML packets, Bloom filter-based filtering, which can filter input XML packets approximately with limited false

positive. It uses a Bloom filter to store all queries of a user, and uses Prefix Filters structure to decrease the number of candidate paths. The most important advantage of it is that the building and maintenance of routing table is faster than traditional technologies. Experiments show that automaton-based system takes more times of Bloom filter-based system when building the routing table. When both the numbers of users and queries are large, Bloom filter-based filtering is more efficient than automaton-based for low depth XML packets.

We plan to extend our work in several directions. First, because the performance of Bloom filter-based filter system is not good when the depth of input XML packets is large, we will devote ourselves to optimization technology for decreasing the number of candidate paths. Second, processing more complex path queries in a Bloom filter-based system is also a interesting topic. Last, XML packets routing in a P2P network is also a part of our future work.

Acknowledgements: Special thanks to Beng Chin Ooi of National University of Singapore for his advices to our work and some helpful discussions. This work was supported in part by the National Science Foundation of China(NSFC) under Grant No. 60496326 and 60228006, the National '863' High-Tech Program of China under Grant No. 2002AA413310, and the Shanghai Rising-Star Program under Grant No. 04QMX1404.

References

[1] The md5 message-digest algorithm. RFC1321.

[2] M. Altinel and M. J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 53–64. Morgan Kaufmann Publishers Inc., 2000.

[3] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and J. V. Streaming xpath processing with forward and backward axes. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, pages 139–150. IEEE Computer Society, 2003.

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

[5] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based xml multi-query processing. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, pages 139–150. IEEE Computer Society, 2003.

[6] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, page 235. IEEE Computer Society, 2002.

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

[8] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, page 341. IEEE Computer Society, 2002.

[9] S. C. F. Peng. Xpath queries on streaming data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 419–430. ACM Press, 2003.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 254–265. ACM Press, 1998.

[11] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata. In *Proceedings of the 9th International Conference on Database Theory*, pages 173–189. Springer-Verlag, 2002.

[12] A. K. Gupta and D. Suciu. Stream processing of xpath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 419–430. ACM Press, 2003.

[13] A. K. Gupta, D. Suciu, and A. Y. Halevy. The view selection problem for xml content based routing. In *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 68–77. ACM Press, 2003.

[14] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT 2004)*, pages 29–47. Springer, 2004.

[15] M. Ley. DBLP. Computer Science Bibliography. http://dblp.uni-trier.de/xml/.

[16] Z. Li and K. A. Ross. Perf join: an alternative to two-way semijoin and bloomjoin. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, pages 137–144. ACM Press, 1995.

[17] J. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558, 1990.

[18] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against xml streams. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, pages 139–150. IEEE Computer Society, 2003.

[19] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using xml. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 160–173. ACM Press, 2001.