Distributed Structural and Value XML Filtering

Iris Miliaraki*
Dept. of Informatics and Telecommunications
National and Kapodistrian University of Athens
Athens, Greece
iris@di.uoa.gr

Manolis Koubarakis
Dept. of Informatics and Telecommunications
National and Kapodistrian University of Athens
Athens, Greece
koubarak@di.uoa.gr

ABSTRACT

Many XML filtering systems have emerged in recent years identifying XML data that structurally match XPath queries in an efficient way. However, apart from structural matching, it is considered equally important to deal with value-based predicates. In this paper, we propose methods to combine both structural and value XML filtering in a distributed environment based on distributed hash tables. Structural matching is performed using automata, while we study different methods for evaluating value-based predicates. As a result, our algorithms scale in both the size of the query set and the number of the predicates per query. We perform an experimental evaluation and demonstrate the strengths and weaknesses of the proposed methods in both a controlled environment of a cluster and on a real testbed provided by the PlanetLab network.

1. INTRODUCTION

As the Web is growing continuously, a great amount of data is available to users, making it more difficult for them to discover interesting information by searching. For this reason, publish/subscribe systems, also referred to as information filtering systems, have emerged in recent years as a promising paradigm. In a publish/subscribe system, users express their interests by submitting a *continuous query* or *subscription* and wait to be notified whenever an event of interest occurs or some interesting piece of information becomes available. Applications of such systems include popular notification services such as news monitoring, blog monitoring and alerting services for digital libraries. Since XML is widely used for data exchange on the Web, a lot of research has focused on designing efficient and scalable XML filtering systems.

In XML filtering systems, subscribers submit continuous queries expressed in XPath/XQuery asking to be notified whenever their queries are satisfied by incoming XML documents. In recent years, many approaches have been presented for providing efficient filtering of XML data against large sets of queries. Centralized approaches include works like XTrie [8], YFilter [14], FiST [23] and others [28, 7, 32, 21]. However, in order to offer XML filtering functionality on Internet-scale and avoid the typical problems of centralized solutions, such a service should be deployed in a distributed environment. Therefore, many works study distributed content-based routing protocols to disseminate XML data over a network of XML brokers or routers [31, 11, 17, 15, 18, 34, 9, 27, 10, 25]. XML brokers are organized in mesh [31] or tree-based overlay networks, and disseminate XML data using information stored in their routing tables. A major weakness of these proposals is that a tree-based overlay can result in load imbalances between the brokers. In addition, some kind of centralized control is frequently required for assigning queries or data sources to the network brokers. With this in mind, we propose an alternative architecture that exploits the power of distributed hash tables (DHTs) to overcome these weaknesses [26].

With respect to the strategy employed for XML filtering, many works use automata or tree-based structures [14, 17, 26], Tian et al. [32] employ relational database techniques, while Chan and Ni [9] and Vagena et al. [35] focus on aggregation techniques for reducing the number of subscriptions. While these strategies have been used with success for representing a set of queries and identifying XML documents that *structurally match* XPath queries, little attention has been paid to *value matching* (i.e., evaluation of value-based predicates). This is an important problem since typical queries, apart from defining a structural path (e.g., $/bib/article/author$), also contain value-based predicates (e.g., $/bib/article[@year > 2007]\ /author[text() =$ "*John Smith*"]). Depending on the selectivity of these predicates, the number of queries which are only structurally matched (i.e., false positives), might be large. For this reason, the benefit of using a filtering engine for structural matching, can be diminished. In other words, XML filtering systems should scale with respect to both the number of the queries indexed and the predicates included in the queries.

Works that deal with the evaluation of value-based predicates in a centralized setting include Gupta and Suciu [19], Diao et. al [14] and Kwon et al. [24]. Gupta and Suciu propose to perform predicate evaluation directly with automata by treating predicates as elements. For this purpose, they compute a lazy deterministic finite automaton (DFA) with the hope that only a small fraction of the DFA states will

*Iris Miliaraki is supported by Microsoft Research through its European PhD Scholarship Programme.

DEBS'10, July 12–15, 2010, Cambridge, UK.

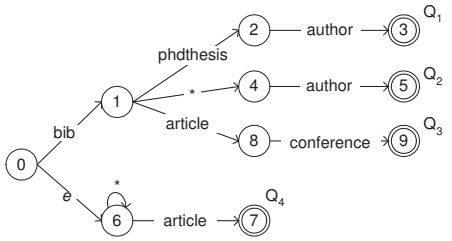
be constructed during runtime. Diao et al. [14] propose two methods to deal with predicate evaluation in the centralized engine of YFilter, where structural matching is performed using a nondeterministic finite automaton (NFA). Method Inline processes predicates as soon as the relevant state is reached during NFA execution while Selection-Postponed delays predicate evaluation until after the whole structure of a query is matched. Kwon et al. [24] extend FiST, a centralized filtering engine [23], where XML filtering takes place by converting queries and data to Prüfer sequences. They propose pFist (predicate enabled FiST) which evaluates queries in a bottom-up fashion, meaning that value-based predicates are checked before the structural matching. To the best of our knowledge, the only approach that deals explicitly with the evaluation of value-based predicates in a distributed environment is the recent work by Chand and Felber [10]. The authors describe an XML content-based network, called XNet, which performs filtering using the XTrie algorithm [8]. Queries are organized in a tree structure (called *factorization tree*), while aggregation techniques are applied to minimize the size of the routing tables kept by XML routers. Value-based predicates are handled in XNet by associating each tree node with a set of predicates.

In this paper, we study how we can combine structural and value XML filtering in an efficient way in a distributed environment based on DHTs. We are interested in XML filtering systems that will run on large collections of loosely maintained, heterogeneous, unreliable machines spread throughout the Internet, thus focus in P2P techniques and especially DHTs. One can imagine that collections of such machines may be used in the future to run non-commercial XML-based public filtering/alerting services for community systems like CiteSeer, blogs etc. Our focus in this work is on how different methods perform when dealing with value-based predicates in such an environment. The main contributions of this paper are the following:

- We fully implement a system offering structural and value XML filtering functionality on top of Pastry DHT [30] (Section 4). For this purpose, we adopt and extend our previous work [26] which only deals with structural matching. (Section 3).
- We describe and compare different methods for combining structural and value XML filtering in our system (Section 4). Firstly, we consider a baseline method which performs filtering in a *bottom-up* way, evaluating first the value-based predicates and then proceeding with the structural matching. The second method filters the document in a *top-down* fashion, performing structural matching before predicate evaluation. Finally, we propose a method that performs *step-by-step* evaluation, checking at each step of the evaluation both the structural and the value-based predicates. To the best of our knowledge, no other work exists that provides an extensive comparison of methods for combined structural and value XML filtering in a distributed environment.
- We propose an optimization for the top-down evaluation method that prunes execution to decrease redundant structural matching. While the former methods are able to deal with all types of predicates, the latter method employs Bloom filters to optimize its performance and is only applicable to equality predicates. To make the comparison of our methods meaningful, we focus only on the evaluation of equality predicates in the rest of this paper.
- We use sampling methods for collecting statistics of XML data filtered by the system and estimate predicate selectivities. These statistics are utilized by one of our methods for improving its performance during predicate evaluation (Section 5).
- We perform an experimental evaluation in both a controlled environment provided by a local cluster and a real-world network infrastructure provided by Planetlab. We demonstrate that our approach scales with respect to the size of the query set and the number of predicates per query for various query workloads (Section 6).

2. BACKGROUND

In this section, we give a very short introduction to the XML data model, the subset of XPath we allow, nondeterministic finite automata, the structure of Bloom filters which will be used for summarizing predicate information and DHTs.



Q_1: /bib/phdthesis[@published=2005]/author[@nationality=greek]
Q_2: /bib/*/author[text()="John Smith"]
Q_3: /bib/article/conference[text()="WWW 2010"]
Q_4: //article[@year=2009]

Figure 1: An example NFA constructed from a set of XPath queries

2.1 XML and XPath

An XML document can be represented using a rooted, ordered, labeled tree where each node represents an element or a value and each edge represents relationships between nodes such as an element - subelement relationship. Element nodes may contain attributes which describe their additional properties or textual data.

XPath [12] is a language for navigating through the tree structure of an XML document. XPath treats an XML document as a tree and offers a way to select paths of this tree. Each XPath expression consists of a sequence of *location steps*. We consider location steps of the following form:

$$axis\ nodetest\ [predicate_1] \ldots [predicate_n]$$

where *axis* is a child (/) or a descendant (//) axis, *nodetest* is the name of the node or the wildcard character "*", and $predicate_i$ is a predicate in a list of one or more predicates used to refine the selection of the node. Each predicate is either an *attribute predicate* of the form [*attr op value*] where

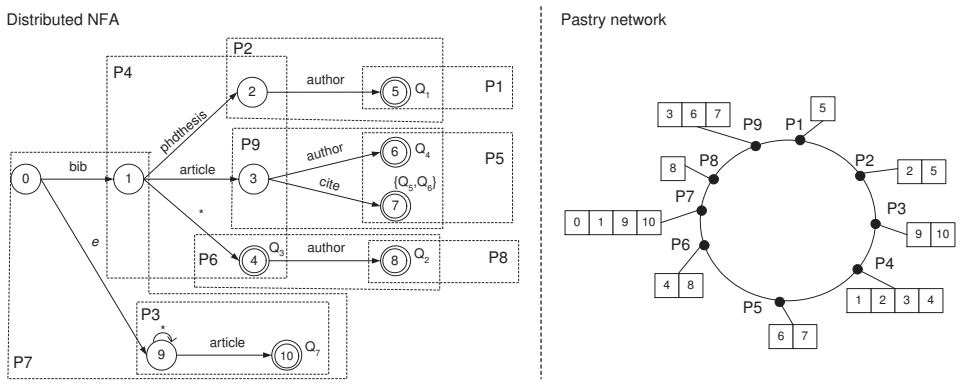


Figure 2: Distributing the NFA on top of a DHT network (l=1)

attr is an attribute name, *value* is an attribute value and *op* is one of the basic logical comparison operators $\{=, >, >=, <, <=, <>\}$ or a *textual predicate* of the form [*text*() *op value*] where *value* is a string value and *op* is a string operator [12].

A *linear path query* q is an expression of the form $l_1 l_2 \ldots l_n$, where each l_i is a location step. In this paper queries are written using this subset of XPath, and we will refer to such queries as *path queries* or *XPath queries* interchangeably. Queries containing branches can be managed by our algorithms by splitting them into a set of linear path queries. Example path queries for a bibliographic database are:

Q_1: `/bib/phdthesis[@published=2007]`
which selects PhD theses published in year 2007.

Q_2: `/bib/*/author[text()="John Smith"]`
which selects any publication of author John Smith.

2.2 Nondeterministic finite automata

A *nondeterministic finite automaton* (NFA) is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of input symbols, $q_0 \in Q$ is the *start state*, $F \subseteq Q$ is the set of *accepting states* and δ, the *transition function*, is a function that takes as arguments a state in Q and a member of $\Sigma \cup \{\epsilon\}$ and returns a subset of Q [20]. The *language* $L(A)$ of an NFA $A = (Q, \Sigma, \delta, q_0, F)$ is $L(A) = \{w \mid \hat{\delta}(w, q_0) \cap F \neq 0\}$. $L(A)$ is the set of strings w in $\Sigma \cup \{\epsilon\}$ such that $\hat{\delta}(q_0, w)$ contains at least one accepting state, where $\hat{\delta}$ is the extended transition function constructed from δ. Function $\hat{\delta}$ takes a state q and a string of input symbols w, and returns the set of states that the NFA is in, if it starts in state q and processes the string w.

Any path query can be transformed into a regular expression and consequently there exists an NFA that accepts the language described by this query. Following Diao et al. [14], for a given set of path queries, we will construct an NFA $A = (Q, \Sigma, \delta, q_0, F)$ where Σ contains element names and the *wildcard* (*) character, and each path query is associated with an accepting state $q \in F$. An example of this construction is depicted in Figure 1.

2.3 Bloom filters

A commonly used data structure for probabilistic representation of a set to support membership queries is the structure of Bloom filters [6]. A Bloom filter is a bit-vector of length m used to represent a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements. Initially all bits are set to 0. Then, using k independent hash functions $h_1, h_2, \ldots, h_k$ with range 1 to m, each element $x \in S$ sets to 1 the bits of positions $h_i(x)$ for $1 \leq i \leq k$. Each bit can be set to 1 many times, but only the first operation has an effect. Then, to check whether an item y is in S, the bits at positions $h_1(y), h_2(y), \ldots, h_k(y)$ are checked. In case any of them is 0 then y is not a member of S, else we assume y is in S. There is however a probability that this is a false positive and it has been shown that this probability is equal to $(1 - e^{kn/m})^k$ [6].

2.4 Distributed hash tables

DHTs like Pastry [30] have emerged as a promising way of providing a highly efficient, scalable, robust and fault-tolerant infrastructure for the development of distributed applications. DHTs are structured P2P systems which try to solve the following *lookup* problem: given a data item x stored in a network of peers, find x. In Pastry [30], each peer and each data item is assigned a unique m-bit identifier. The identifier of a peer can be computed by hashing its IP address and is used to indicate its position in a circular identifier space ranging from 0 to $2^m - 1$. For data items, we first have to compute a *key* and then hash this key to obtain the identifier. Pastry routes messages to the peer whose identifier is numerically closest to the given key using a technique called prefix routing. Such requests can be done in $O(logn)$ steps, where n is the number of nodes in the network.

In the rest of the paper we use Pastry as the underlying DHT. However, our techniques are DHT-agnostic; they can be implemented on any DHT that offers the standard lookup operation.

3. STRUCTURAL MATCHING

Before describing the methods we propose for dealing with value-based predicates, we briefly discuss in this section how structural matching is performed. We have chosen to use the methods described in our previous work [26], where we use an nondeterministic automaton (NFA) for indexing XPath queries and execute this NFA for filtering incoming XML data. The NFA representing the queries is distributed among the network peers and peers collaborate with each other for filtering incoming XML data distributing the processing load. In the following, we describe how

the distributed NFA corresponding to a set of XPath queries is constructed, maintained and executed. A more detailed description can be found in our previous work [26].

3.1 NFA-based distributed index

We use an NFA-based model, similar to the one used in the system YFilter [14], for indexing queries in our system. The NFA is constructed from a set of XPath queries and is used as a matching engine that scans incoming XML documents and discovers matching queries. We distribute the NFA on top of a Pastry network and for this reason we use the term *distributed NFA* to refer to it. Each state is uniquely identified by a key and this key is used for determining the peer that will be responsible for this state. The *responsible peer* for state with key k is the peer whose identifier is numerically closest to $Hash(k)$, where $Hash()$ is the DHT hash function. The states of the NFA are distributed by assigning each state q along with every other state included in $\hat{\delta}(q, w)$, where w is a string of length l included in $\Sigma \cup \{\epsilon\}$, to the responsible peer for q. Note that l determines how much of the NFA is the responsibility of each peer and this results in having peers responsible for overlapping fragments of the NFA, the size of which is characterized by parameter l. The key of an automaton state is formed by the concatenation of the labels of the transitions included in the path leading to the state. For example, the key of state 2 in Figure 1 is the string "*start*"+"*bib*"+"*phdthesis*" and the key of state 6 is "*start*"+"$"[1], since ϵ-transitions are represented using character $. An example of how an NFA is distributed on top of a Pastry network for $l = 1$ is depicted in Figure 2.

3.2 Constructing the NFA

To achieve the above distribution of the NFA, the automaton is incrementally constructed as queries arrive. A location step can be represented by an NFA fragment [14]. The NFA for a path query can be constructed by concatenating the NFA fragments of the location steps it consists of, where the last state of the NFA is the accepting state of the path query. Inserting a new query into an existing NFA requires to *combine* the NFA of the query with the already existing one. To insert a new query represented by an NFA S to an existing NFA R, we start from the common start state shared by R and S, and we traverse R until either the accepting state of S is reached or we reach a state for which there is no transition that matches the corresponding transition of S. If the latter happens, a new transition is added to that state in R. Since the states of the NFA are distributed among the network peers, we traverse the distributed NFA by visiting the relevant peers. Depending on the value of parameter l a peer may need to store additional states locally before forwarding the indexing request.

3.3 Executing the NFA

NFA execution proceeds in an event-driven fashion. The XML document is parsed using a SAX parser and the produced events are fed, one event at a time, to the NFA. The parser produces events of the following types: `StartOfElement (SOE)`, `EndOfElement (EOE)`, `StartOfDocument (SOD)`, `EndOfDocument (EOD)` and `Text`. The nesting of elements in an XML document requires that when an `EOE` event is raised, the NFA execution should backtrack to the states it was in when the corresponding `SOE` was raised. For achieving this, YFilter maintains a stack, called the *run-time stack*, while executing the NFA. Since many states can be active at the same time in an NFA, the stack is used for tracking multiple active paths. The states placed on the top of the stack represent the *active states* while the states found during each step of execution after following the transitions caused by the input event, are called the *target states*. Execution is initiated when a `SOD` event occurs and the start state of the NFA is pushed into the stack as the only active state. Then, each time a `SOE` event occurs for element e, all active states are checked for transitions labeled with e, *wildcard* and ϵ-transitions. In case of an ϵ-transition, the target state is recursively checked one more time. All active states containing a *self-loop* are also added to the target states. The target states are pushed into the run-time stack and become the active states for the next execution step. If a `EOE` event occurs, the top of the run-time stack is popped and backtracking takes place. Execution proceeds in this way until the document has been completely parsed.

Similarly to YFilter, we maintain a stack in order to be able to backtrack during the execution of the NFA. For each active state, we want to retrieve all target states reached by a certain parsing event. We propose two methods for executing the NFA [26]: the first proceeds in an iterative way while the other executes the NFA in a recursive fashion. In the *iterative method*, the publisher peer is responsible for parsing the document, maintaining the run-time stack and forwarding the parsing events to the responsible peers. In this case, the execution of the NFA proceeds in a similar way as in YFilter, with the exception that target states cannot be retrieved locally but need to be retrieved from other peers. In the case of the *recursive method*, the publisher peer forwards the XML document to the peer responsible for the start state to initiate the execution of the NFA. The execution continues recursively, with each peer responsible for an active state continuing the execution. Notice that the run-time stack is not explicitly maintained in this case, but it implicitly exists in the recursive executions of these paths. The execution of the NFA is parallelized in two cases. The first case is when the input event processed has siblings with respect to the position of the element in the tree structure of the XML document. In this case, a different execution path is created for each sibling event. The second case is when more than one target states result from expanding a state. Then, a different path is created for each target state, and a different peer continues the execution for each such path. We experimented with both methods [26] and demonstrated that the recursive method outperforms the iterative one with respect to both the required messages and total filtering time. In the rest of the paper, we only use the recursive method for filtering incoming XML data.

4. VALUE MATCHING

In the previous section, we described how structural matching is performed using the methods described in our previous work [26]. In this work, we focus on the evaluation of value-based predicates. Consider for example query q: $/bib/article[@conf = DEBS]/author[text() =$ "*John Smith*"], which selects the articles of author "John Smith" published at the DEBS conference. Filtering incoming XML data

[1]Operator + is used to denote the concatenation of strings.

against this query requires to check whether the data structurally match the query and also whether the value-based predicates of the query are satisfied. In this section, we describe different techniques for dealing with the evaluation of value-based predicates together with distributed structural matching approaches as the ones supported by our system.

Following a widely used strategy from relational query optimization, where selections are applied as early as possible, we can check the value-based predicates before proceeding with the structural matching following a *bottom-up* approach. Even though the heuristic of pushing selections early works well in the case of relational query processing, in our case peers may put a lot of effort evaluating predicates for queries whose structure may not be matched. Thus, we can alternatively check the predicates after the structural matching operating in a *top-down* fashion. In this case, value-based predicate evaluation takes place after the structural matching of queries. Furthermore, considering that XPath queries consist of distinct steps and each step may be associated with one or more value-based predicates, we can perform structural matching along with predicate evaluation at each step. We consider that the latter approach performs XML filtering in a *step-by-step* fashion.

Finally, since in our case the XPath queries are indexed using an NFA, we could perform predicate evaluation directly with the automaton by adding extra transitions for the predicates. An expected drawback of such a method comes from the fact that the elements in a set of XPath queries represent a rather small set since they are constrained by the schema, while the values of the predicates may form a large set. This could result in a huge increase of the NFA states and also destroy the sharing of path expressions for which the NFA was selected to begin with. For this reason, we do not study this method any further.

In the following, we describe how we implement the above methods for offering XML filtering functionality on top of the Pastry DHT. We design our methods assuming that queries are indexed using the distributed NFA described in our previous work [26]. In the case of bottom-up evaluation method, we use a different indexing algorithm based on the query predicates similar to the ones used in the area of information filtering (IF) by Tryfonopoulos et al. [33] and Aekaterinidis and Triantafillou [4], where queries are expressed using a simple attribute-value data model. Lastly, we propose certain optimizations for achieving better performance when dealing with equality predicates by utilizing Bloom filters. While the former methods are able to deal with all types of predicates, the latter method which employs Bloom filters to optimize its performance is only applicable to equality predicates. To make the comparison of these methods meaningful, we study only the evaluation of equality predicates in the rest of this paper.

4.1 Prerequisites

4.1.1 Data representation

As in our previous work [26], we use the same representation for the XML documents that arrive for filtering. In particular, we parse the XML document using a SAX parser and produce events of the following types: `SOE, EOE, SOD, EOD` and `Text` (also described in Section 3). We enrich parsing events `SOE` and `EOE` with the position of the corresponding element using a pair (L:R,D), where L and R are generated by counting tags from the beginning of the document until the start tag and the end tag of this element, and D is its nesting depth. This representation was introduced by Consens and Milo [13] and is used to efficiently check structural relationships between two elements.

We also generate a set of *candidate predicates* from the XML document. Each candidate predicate is an equality predicate constructed using either an element name, an attribute and its value (attribute predicates) or the element name and its text value (textual predicates), as found in the XML data fragment. We consider these predicates as candidates because they correspond to the query predicates that can be satisfied by this document.

The publisher peer is responsible for enriching the parsing events and producing the set of the candidate predicates. The enriched parsing events along with the candidate predicate set are forwarded with each filtering request. Note that we need to parse the document for generating the above structures before the execution begins. However, we do not consider this additional parsing operation to cause significant load to the system since it is typical for XML dissemination systems to deal with relatively small documents. As mentioned in the study of Barbosa et al. [5], the average size of an XML document in the Web is only 4 Kb, while the maximum size can reach 500Kb.

4.1.2 Terminology

In the following we give some definitions that will be used throughout the rest of the paper.

Definition 1. An *NFA path* is a sequence of NFA states $st_0, st_1, \ldots, st_n$ such that for every pair states st_i,st_{i+1} there exists an input symbol w where $\delta(w, st_i) = st_{i+1}$ (i.e., there exists a transition from st_i to st_{i+1}).

Definition 2. An *NFA accepting path* of a query q is an NFA path $st_0, st_1, \ldots, st_n$ where st_0 is the start state of the NFA and st_n is the accepting state of q.

4.2 Bottom-up evaluation

Based on the heuristic of pushing selections as early as possible in the case of relational query processing, we describe a method that operates in a bottom-up fashion. This method indexes queries in the network using their predicates and performs filtering by first checking the value-based predicates and then proceeding with the structural matching. We will refer to this method as *bottom-up evaluation* and we consider this to be our baseline method. Even though we only describe how this method operates for equality predicates, the algorithms can be extended to support other complex predicates, like range predicates, adopting techniques like the ones described by Aekaterinidis and Triantafillou [4].

4.2.1 Indexing queries

In contrast to the other methods, where the indexing of the queries is done using the distributed NFA, a different indexing algorithm is used in the bottom-up evaluation. Since we first want to discover queries that contain specific predicates and then structurally match them against the incoming XML data, indexing is based on the predicates included in the query. For each distinct predicate included in a query set, we consider a certain peer which will be responsible for it and hence also responsible for the set of queries that contain this predicate. Each predicate is uniquely identified by

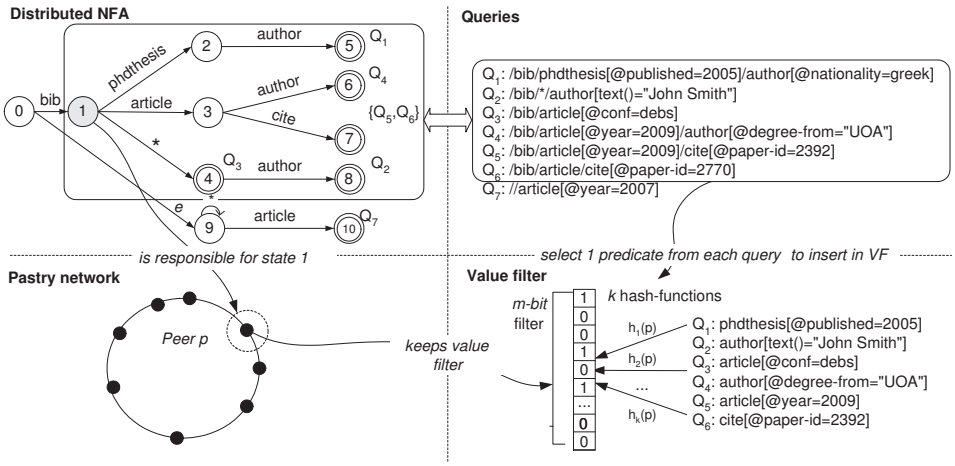


Figure 3: Top-down evaluation with pruning: Constructing value filters

a key formed by its string representation and this key is used for determining the peer that will be responsible for it. Given that we assume the Pastry DHT as the underlying network, the *responsible peer* for predicate with key k is the peer whose identifier is numerically closest to $Hash(k)$, where $Hash()$ is the DHT hash function. Each peer keeps a local index mapping predicates to the list of queries that contains them. This indexing algorithm resembles works presented for information filtering (IF) on top of DHTs including work of Tryfonopoulos et al. [33], where queries are expressed using a simple attribute-value data model and attribute values are used to map queries to peer identifiers.

4.2.2 Filtering data

Whenever XML data arrives for filtering, we construct a set of candidate predicates as usual. For each candidate predicate, we send a filtering request to the peer responsible for this predicate. The peer checks its local hash index, retrieves the queries that contain the specific predicate and then performs *locally* structural matching for these queries. If a query is also structurally matched with the incoming XML data, a notification is sent to its subscriber.

4.3 Top-down evaluation

In contrast to the previous method, this approach operates in a top-down fashion, evaluating predicates after performing structural matching. In this method, we use the distributed NFA [26] to identify the subset of queries that structurally match incoming XML documents, and then we evaluate the predicates of these queries. In other words, this method evaluates predicates after the execution of the NFA. Since structural matching is performed in parallel by multiple peers, each of these peers identifies a different subset of structurally-matched queries. So, whenever a peer identifies such a set, it is also responsible for the predicate evaluation. We will refer to this method as *top-down evaluation*.

4.3.1 Indexing queries

Query indexing is performed as described in Section 3. However, when an accepting state is reached, we need to perform predicate evaluation for the queries associated with that state. In order to avoid evaluating each one of the predicates in the queries separately, we utilize an index structure. Since we only deal with equality predicates at the current state of our work, it is sufficient to construct a *hash index* mapping predicates to the list of queries which contain them. For each accepting state, we include in the hash index all the predicates of the corresponding queries. We can easily extend our method to support range predicates if we keep B^+-tree indexes instead of hash indexes.

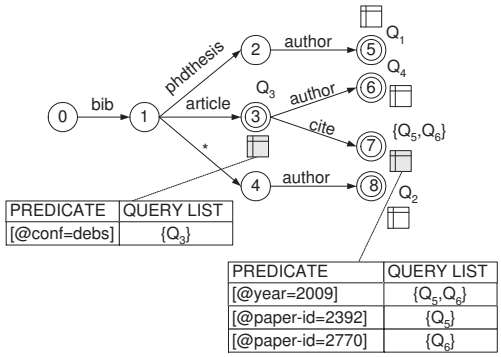


Figure 4: Hash indexes for top-down evaluation

We illustrate the above using an example shown in Figure 4. Using the same example NFA, in the top-down evaluation we keep hash indexes for the accepting states (i.e., states $3, 5, 6, 7$ and 8). The hash index of state 3 contains only one entry for Q_3 and the index of state 7 contains three entries, one for each distinct predicate contained in Q_5 and Q_6.

4.3.2 Filtering data

Filtering is performed by executing the distributed NFA until we reach an accepting state. When a peer reaches an accepting state, it needs to further evaluate the queries associated with that state with respect to their value-based predicates. Instead of sequentially checking all queries, we use the candidate predicates to probe the hash index. If a query is matched by incoming data then it will be returned as a match by the index. If a query has all its predicates satisfied, then a notification is sent to its subscriber.

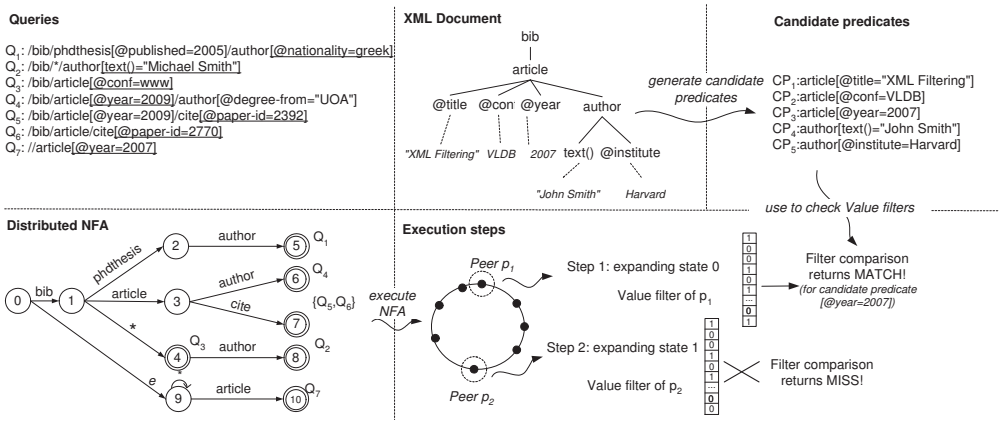


Figure 5: Top-down evaluation with pruning: Querying value filters

4.4 Top-down evaluation with pruning

To overcome possible shortcomings of top-down evaluation caused by spending too much effort with structurally matching queries containing predicates that are not satisfied by incoming XML data, we propose the following optimization when dealing with equality predicates. We use a compact summary of predicate information to stop the execution of the NFA (i.e., *prune* this execution path) whenever we can deduce that no match can be found if the execution continues. We will refer to this method as *top-down evaluation with pruning*.

Recall that the NFA is a tree structure distributed among the network peers and during structural matching we traverse the distributed NFA. Each peer is responsible for storing many fragments of this NFA. At each step of the execution, we can consider that a part of the NFA has been *revealed* while the rest part is not. So, we can use Bloom filters to represent these NFA fragments with respect to the predicates they contain. Then, we can decide whether or not we will continue execution by consulting these filters. The main idea of this method is the use of Bloom filters to summarize the query predicates indexed in a specific NFA fragment. For this reason, each peer keeps one Bloom filter, called *value filter* (VF), which summarizes a set of value-based predicates.

Consider a peer p and a state st for which p is responsible for. Since we assume only conjunctions of predicates in queries, if at least one of the query predicates is not satisfied, then the query cannot be matched. Thus, for each query q whose NFA accepting path contains st, we insert one predicate of q in the VF of p. In other words, only one predicate of each query is required in the value filter. Each attribute predicate of the form $element[@attr = value]$ is inserted as a whole in the VF using its string representation $element{+}attr{+}value$ concatenated with the state identifier. Likewise, textual predicates of the form $element[text() = value]$ are inserted as a whole in the VF using their string representation $element + text() + value$ together with the state identifier. We insert predicates in the filters during query indexing. Since we need to traverse the NFA accepting path of each query in order to index it, it is guaranteed that all relevant VFs will be updated. Since each query may contain more than one predicates, we need to select which one will be inserted in the value filter. Section 5 studies how we can select this predicate.

We illustrate the above using an example shown in Figure 3. We consider queries $Q_1, Q_2, Q_3, Q_4, Q_5, Q_6$ and Q_7 and the corresponding NFA. This NFA is distributed among the network peers. In our example, peer p is responsible for state 1. Since state 1 belongs to the NFA accepting paths of queries Q_1, Q_2, Q_3, Q_4, Q_5 and Q_6, one predicate of each query is inserted in the VF of p. Note that we do not insert a predicate of Q_7 since Q_7 is not indexed in the corresponding NFA fragment i.e., its NFA accepting path does not include state 1.

4.4.1 Filtering data

In this method, each peer that participates in the execution process performs an additional step before expanding an NFA state. During this step, it checks whether its VF matches any of the candidate predicates. In case no candidate predicate can be matched (*miss*), execution is pruned instantly, while if at least one of the candidate predicates is found in the filter (*match*), the peer proceeds with the execution. In the worst case where no execution path is pruned, this method works exactly like the top-down evaluation. As described previously, when we reach an accepting state each peer keeps a hash index for performing predicate evaluation.

Note that we do not explicitly check query predicates, but at each step of the execution, VF only give us enough information regarding whether to continue the execution or not. We have no further information regarding which queries contain any of the candidate predicates.

We demonstrate how we use value filters to prune NFA execution using an example shown in Figure 5. Again, we consider the same set of queries and the corresponding NFA distributed among the network peers. In our example, peer p_1 is responsible for state 0, while peer p_2 is responsible for state 1. The value filters at peers p_1 and p_2 contain the query predicates that are underlined in the figure. Suppose a peer publishes the XML document depicted in the figure. This peer is responsible for generating candidate predicates from the document and initiating filtering. Peer p_1 which is responsible for the start state receives the filtering request and initiates filtering. Before proceeding with the expansion of state 0, it checks whether any of the candidate predicates

is included in its local value filter. In this case, VF returns a match for predicate $article[@year = 2007]$ which is included in query Q_7. So, state 0 is expanded causing states 1 and 9 to become active. Peer p_2 continues execution from state 1. In this case, when p_2 checks its local value filter, it founds no match and this execution path is pruned and state 1 is not expanded. Due to space limitations, the execution path from state 9 is not depicted in the figure.

4.5 Step-by-step evaluation

Our last method is based on the concept of performing value matching simultaneously with the structural matching. Considering that XPath queries consist of distinct steps and each step may be associated with one or more value-based predicates, we can perform structural matching along with predicate evaluation at each step. Therefore, in this case predicates are evaluated during NFA execution.

4.5.1 Indexing queries

As before, queries are indexed using the distributed NFA. During query indexing, each peer organizes all the predicates included in its local queries using an index. Each predicate is associated with the relevant NFA state it refers to. This local index maps predicates to the list of queries which contain them. Since a query may not contain predicates, we also map a generic `true` predicate to queries with no predicates at this step.

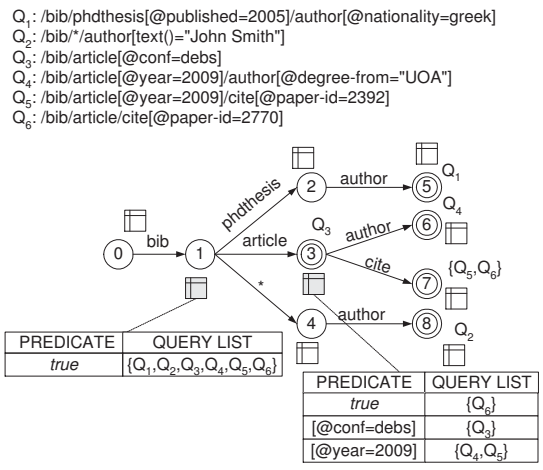


Figure 6: Hash indexes for step-by-step evaluation

4.5.2 Filtering data

Recall that each filtering request carries a document representation and the candidate predicates produced by the document. In addition, each filtering request also includes the queries which have been partially matched with respect to both their structure and predicates until the current execution step. To filter incoming XML data, each peer participating in the filtering process uses the candidate predicates to probe its local index.

Let us explain how this works in detail. After peer p expands a state st during execution, it uses the candidate predicates to probe its hash index. Let $CurrQ$ and $PrevQ$ be the set of queries returned by the hash index and the set of previously satisfied queries respectively. We will denote as $NextQ$, the set of satisfied queries after expanding st, i.e., $NextQ = PrevQ \cap CurrQ$. In case state st is an accepting state for a query $q \in NextQ$ (both structure and predicates have been satisfied), then p notifies the subscriber of q and removes q from $NextQ$. Then, execution continues and p forwards a new filtering request that includes $NextQ$. At the current state of our work, we only support equality predicates and for this reason we use a *hash index*. We plan to support range predicates using appropriate indexes like B^+-trees.

We illustrate the above using an example shown in Figure 6. We construct an NFA from a set of six queries. For the sake of simplicity, we do not show how this NFA is distributed among network peers. For each NFA state, we keep a different hash index. In Figure 6, we show the contents of the hash indexes associated with states 1 and 3 respectively. The index at state 1 contains only one entry mapping the *true* predicate to $\{Q_1, Q_2, Q_3, Q_4, Q_5, Q_6\}$, since no query contains a predicate at that step. While, the index at state 3 contains three entries, one for the *true* predicate and two for the predicates contained in queries Q_3,Q_4 and Q_5. Q_4 and Q_5 contain the same predicate, so one entry is added to the hash index. Execution ends when $NextQ$ becomes empty.

5. ONLINE SELECTIVITY ESTIMATION

We previously described the top-down evaluation with pruning which uses Bloom filters to summarize predicate information. Recall that we want to select and insert only one of the predicates for each query in the respective filters. This selection can be made *randomly* or ideally we can select the predicate which can lead to better pruning. For example, consider query Q:
`/dblp/article[@year=2009]/author[text()="John Smith"]`,
which selects all articles by John Smith published in 2009. Since one would expect that there are many more articles published in 2009 than articles written by John Smith in general, we would like to insert the predicate on the author in VF. Thus, it is reasonable to prefer inserting the *most selective predicate* of each query in the value filter.

In this section, we define the selectivity of predicates in our context, describe our techniques for keeping statistics to estimate predicate selectivities in our system, and describe how these statistics are used by the top-down evaluation method.

5.1 Definitions

As described in Section 2.1, we deal with attribute and textual predicates on XML elements. We define the *selectivity of a textual predicate* $[text() = v]$ on element e for an XML document collection D as the fraction of elements e, reachable by any path, with value v in D. Similarly, we define the *selectivity of an attribute predicate* $[@attr = v]$ for attribute $attr$ on element e for an XML document collection D as the fraction of elements e, reachable by any path, that satisfy the predicate in D. We also consider that any predicate on an element explicitly defined is more selective than a predicate on a wildcard element. The following formulas give the formal definitions:

$$\text{sel}(e[text() = v]) = \frac{\text{total occurrences of } e \text{ equals } v \text{ in } D}{\text{total occurrences of } e \text{ in } D} \quad (1)$$

$$\text{sel}(e[@attr = v]) = \frac{\text{total occurrences of } attr \text{ equals } v \text{ in } D}{\text{total occurrences of } e \text{ in } D} \quad (2)$$

5.2 Distributed sampling

In an XML dissemination system like the one described in this work, a large volume of XML data is expected to arrive for filtering. As a result, it is not feasible to store the entire set of XML data that have been processed to compute the exact predicate selectivities. For this reason, we use sampling and keep information only about a fixed-size random subset of the data collection.

Formally, we want to obtain a random sample of size n from a set of size N, where N is not known beforehand. A well-known and broadly used algorithm for random sampling is the *reservoir sampling algorithm* proposed by Vitter et al. [36]. For obtaining a random sample of size n, the algorithm works as follows. The first n items are inserted in the reservoir. Then, a random number of items is skipped and the next item replaces a random item from the reservoir. Again, a random number of items is skipped and so on. This way the reservoir always contains a random sample of n items.

Let us recall at this point what kind of selectivities we want to extract from the document sample. We keep the occurrences of each element along with the occurrences of its attributes and their values. Each peer keeps its own independent reservoir of n items, in our case XML documents. This reservoir is created from the documents that arrive at this peer. This random sample is kept at the document-level, so each item in the reservoir is actually an XML document that has previously arrived for filtering. Whenever a document arrives at peer, it runs the reservoir sampling algorithm to decide whether or not to add this document to its local reservoir.

5.3 Using statistics in predicate evaluation

We have described what kind of selectivity statistics we keep and how we compute them. Let us now describe how these statistics are exploited by the top-down evaluation method with pruning to select which predicates will be inserted in the VFs.

Each time a query arrives at a peer p, before starting indexing the query, p computes the selectivities of its predicates by consulting its local reservoir. Then, p marks the most selective predicate of the query and initiates indexing. This process is performed only by the subscriber peer. Recall that we update value filters during query indexing, so the most selective predicate is chosen using the current selectivity statistics. Alternatively, we could also update filters at a later time because XML data may follow a different distribution. However, this would require to be able to also remove entries from the Bloom filters. In this case, one could use Counting Bloom filters [16].

6. EXPERIMENTS

In this section, we present an experimental evaluation of our system. We have implemented our methods for combined structural and value matching in Java on top of FreePastry [1]. For our experiments, we used as testbeds both PlanetLab network (`http://www.planet-lab.org/`) as well as a more controlled environment provided by a local shared cluster (`http://www.grid.tuc.gr/`). We focus on the performance of the methods we described previously for evaluating value-based predicates in our system. Demonstrating the performance of structural matching is out of the scope of this paper and the interested reader can refer to our previous work [26].

6.1 Experimental setup

In the case of the top-down evaluation method which uses pruning, we run two versions of this method based on how the predicate of each query included in the value filters is chosen. In the first case, we use selectivity statistics to choose the most selective predicate (denoted as *top-down with pruning (most-sel)*) and in the second case, we choose the predicate randomly (denoted as *top-down with pruning (random)*). We are interested in measuring the time at which the notifications arrive at the subscribers during the XML filtering process referred to as *notification arrival times*. We also refer to the arrival time of the last notification as the *total filtering time* or *filtering latency*. In addition we also measure the total number of messages sent through the network either during indexing queries or during the filtering of XML data. All measurements presented below are averaged over 10 runs for each experiment. In the following, we describe our experimental setup.

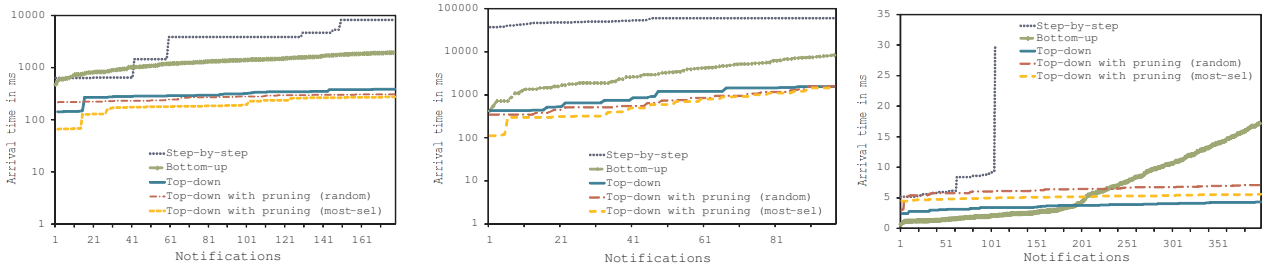
Network setup. When using the Planetlab testbed, our network consists of 253 peers across four continents that were available and lightly loaded at the time of the experiments. In the cluster, which consists of 34 machines, we run up to 136 peers, i.e., 4 peers in each machine. In each experiment, firstly, queries are indexed in the system by a set of randomly selected available peers and then, XML data is published by random peers. In the case of Planetlab, we used 100 peers of the 253 available peers for indexing the query set. The same number of peers was also used for the experiments conducted in the cluster.

Data sets. We generate a synthetic data set using the NITF (News Industry Text Format) DTD, which has also been used in other works [26, 8, 14, 21]. The NITF DTD represents an interesting case where a large fraction of elements are allowed to be recursive. We use the IBM XML generator [2] to synthetically generate 100 documents. The same DTD is used to generate sets of 1,000,000 path queries using the XPath generator available in the YFilter release [3]. Each set is generated using the following parameters: the maximum number of steps d per query fixed at 15 and the number of predicates P per query ranging from 1 to 6.

Bloom filter size. We use Bloom filters with a fixed size of 100,000 bits in our experiments resulting in a false probability rate of about $0,8\%$ while using 7 hash functions. We have chosen the above parameters based on our experimental setting. However, the false positive rate depends on the total number of items (i.e. in our case predicates) expected to be inserted in the Bloom filter. This may not be known beforehand in a realistic scenario and different assumptions should be made for selecting the values for the above parameters. We consider as future work to further explore this.

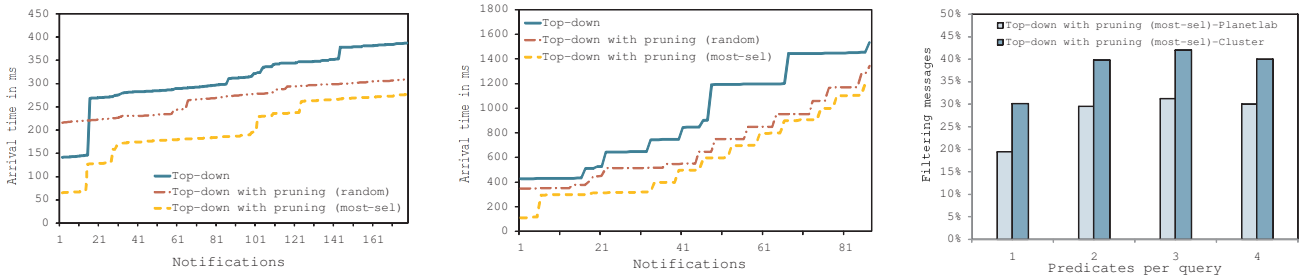
6.2 Filtering data

In this experiment, we study the performance of our methods during XML filtering. Firstly, we show results from experiments conducted using the cluster machines after indexing 100,000 queries (Figures 7(a) and 7(b)). Figures 7(a) and 7(b) show the arrival times for queries involving 2 and 4 predicates respectively plotted on a logarithmic scale. In general, methods which evaluate predicates in a top-down fashion outperform the others by a wide margin.



(a) Cluster (2 predicates per query) (b) Cluster (4 predicates per query) (c) Planetlab (2 predicates per query)

Figure 7: Notification arrivals



(a) Cluster (2 predicates per query) (b) Cluster (4 predicates per query) (c) Performance improvement

Figure 8: Benefit of using value filters

The step-by-step method exhibits the worst performance in both cases, deteriorating as the number of predicates per query is increased from 2 to 4. This is due to the fact that peers spend a lot of effort evaluating predicates for queries whose structure is not matched by the XML data. Likewise, bottom-up evaluation method which indexes queries based on their value-based predicates also demonstrates a poor performance, even though it is 10 times faster compared to step-by-step. This is explained because each peer responsible for a candidate predicate (as this is generated by incoming XML data) performs structural matching for all queries containing this predicate.

In Figure 7(c) we show the notification arrival times when using the Planetlab network. Since Planetlab peers are located in four continents, network delays in such a setting are considerably higher compared to the ones observed in the cluster. As before, the step-by-step evaluation method exhibits the worst performance suffering even more from the network delays of Planetlab peers. It is interesting though that the bottom-up evaluation method generates the first notifications faster than top-down evaluation methods. An explanation for this behavior is that less peers participate in the filtering process decreasing network delays, while top-down evaluation methods exhibit a more stable performance due to the fact that processing load is distributed among a larger number of peers. For example, bottom-up evaluation generates the 260th notification about 3 seconds later compared to the top-down evaluation methods.

In Figure 9, we show the total number of messages that travel through the network during filtering of XML data while varying the total number of indexed queries. Note that the notifications generated during the filtering are not included since all methods generate the same amount of notification messages. Bottom-up method generates a constant number of messages since this is analogous to the number of candidate predicates generated by the XML data and independent of the total indexed queries. Moreover, top-down evaluation method and step-by-step generate almost the same number of messages in each case. This is due to the fact that both methods traverse almost the same NFA fragment during XML filtering. Finally, when pruning is used less messages are sent since, as expected, execution is stopped at some cases.

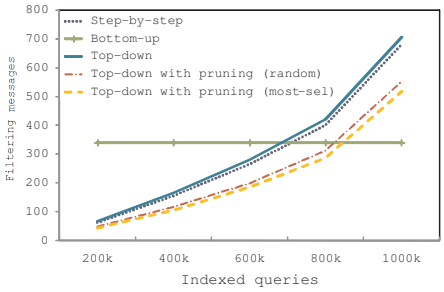


Figure 9: Filtering messages for 200k to 1M queries

6.3 Benefit of using value filters

In this experiment, we study the benefit of using Bloom filters to summarize predicate information aiming at stopping NFA execution if we determine that no query can be satisfied by the incoming XML data. We compare our methods after having indexed 100,000 queries and again show notification arrival times after publishing the XML docu-

ments. The number of predicates per query ranges from 2 to 4. Figures 8(a) and 8(b) show the arrival times for queries involving 2 and 4 predicates respectively. In general, we observe that top-down evaluation demonstrates a better performance when pruning is used, either by exploiting selectivity statistics or not. As queries involve a larger number of predicates, the gain of using Bloom filters increases, resulting in faster arrival times. This is depicted in Figure 8(b) where queries involve 4 predicates and the use of value filters results in a better performance. In addition, the use of selectivity statistics when constructing the value filters results in better arrival times in most cases. We expect that we could further improve its performance with more sophisticated techniques for selectivity estimation but we consider this to be out of the scope of the paper.

Figure 8(c) summarizes how the use of selectivity statistics can result in better performance as we increase the number of predicates per query. We have conducted this experiment in both Planetlab and the cluster. We measure the performance improvement using the formula $\frac{t_{base}-t_{method}}{t_{base}}$, where t_{method} is the filtering time for the method under observation and t_{base} is the filtering time of the top-down evaluation method without any pruning. As shown in the figure, for queries with one predicate, the use of pruning results in a 20% improvement in Planetlab, while in the cluster, where network delays are minimized, we achieve a 30% improvement. As the number of predicates per query grows, the performance improvement due to the use of pruning over the simple top-down evaluation method increases significantly, reaching a 39% improvement for queries involving 4 predicates. We have experimented with a higher number of predicates using other datasets and the trend observed was similar. However, in most datasets, more predicates result in significantly less notifications, minimizing the potential improvement in performance.

6.4 Indexing queries

We also compared our methods with respect to the time required for query indexing. Let us recall at this point that top-down evaluation which uses pruning needs to maintain and update the value filters, step-by-step method keeps predicate information for each NFA state, while in simple top-down evaluation peers keep predicate indexes associated with the accepting NFA states. In the case where 100,000 queries were indexed in the Planetlab network we can report the following. All the methods exhibit similar performance indexing the set of queries in about 20 seconds. Even when the number of predicates increases, total indexing time remains almost the same due to the fact that many peers participate and share the load. We omit the relevant graph due to space limitations.

7. RELATED WORK

Many approaches exist for XML filtering in a centralized setting like YFilter [14], XTrie [8], XPush [19], Index-Filter [7] and others. Also, many systems have been proposed that deal with XML dissemination in distributed environments [11, 17, 15, 18, 34, 9, 26]. As mentioned previously, systems like XTrie [8] which use automata or other tree-structures for indexing XML path queries, can be efficiently used to identify XML documents which structurally match XPath queries but cannot be used to efficiently perform predicate evaluation.

Works that deal with the evaluation of value-based predicates in a centralized setting include Gupta and Suciu [19], Diao et. al [14] and Kwon et al. [24]. Gupta and Suciu [19] propose to perform predicate evaluation directly with automata by treating predicates as elements. For this purpose, they compute a lazy DFA with the hope that only a small fraction of the DFA states will be constructed during runtime. Diao et al. [14] propose two methods to deal with predicate evaluation in the centralized engine of YFilter. Method Inline processes predicates as soon as the relevant state is reached during NFA execution while Selection-Postponed delays predicate evaluation until after the whole structure of a query is matched. Kwon et al. [24] extend FiST, a centralized filtering engine [23], where queries and data are converted to Prüfer sequences. They propose pFist (predicate enabled FiST) which evaluates queries in a bottom-up fashion, meaning that value-based predicates are checked before the structural matching.

Vagena et al. [35] deal with XML message aggregation and describe the VA-RoXSum structure. VA-RoXSum aggregates structural information of XML data in a compact way, while Bloom filters are used to encode values of root-to-leaf XML data paths. Similarly to YFilter, in each broker queries are indexed using an NFA and value predicates are augmented with the final states of the NFA using Bloom filters. Also, Gong et al. [18] use Bloom filters to summarize path queries and build routing tables for efficiently filtering XML data. Finally, Bloom filters have been used by Koloniari and Pitoura [22] as summaries of XML data to efficiently route path queries in a P2P network.

To the best of our knowledge, the only approach that deals explicitly with the evaluation of value-based predicates in a distributed environment is the recent work by Chand and Felber [10]. The authors describe an XML content-based network, called XNet, which performs filtering using XTrie [8]. Queries are organized in a tree structure (called *factorization tree*) while aggregation techniques are applied to minimize the size of the routing tables kept by XML routers. Value-based predicates are handled in XNet by associating each tree node with a set of predicates. In addition, the authors consider that the overhead of predicate evaluation depends on the query workload and assume that it is typically very low compared to the cost of structural matching.

With respect to our selectivity estimation techniques, each peer uses the well-known algorithm of reservoir sampling and keeps its own independent sample for computing estimates. However, Pitoura and Triantafillou [29] focus on distributed sampling algorithms on top of DHTs and describe sampling algorithms to estimate peer loads. These algorithms can be easily adapted to obtain better estimates of predicate selectivities by combining the estimates computed by different peers. For instance, using the random walking algorithm described [29], each peer can collect the estimated selectivities from its neighbors and compute better estimates.

8. CONCLUSIONS AND FUTURE WORK

We describe, implement and study the performance of a system which combines both structural and value XML filtering on top of Pastry DHT. Our system performs structural matching using automata and supports several methods, each of which operates in a different fashion, namely top-down, bottom-up and step-by-step, for evaluating predicates. We propose an optimization for one of our methods,

which utilizes Bloom filters to summarize predicate information aiming to decrease the effort spent during value matching. We experimentally evaluate our approach on both PlanetLab and on a local cluster and demonstrate how our methods can scale in both the size of query set and the number of predicates per query. Our future work concentrates on extending our methods for range predicates. Additionally, we want to consider more fine-grained statistics for estimating predicate selectivities as well as other distributed sampling methods.

9. ACKNOWLEDGMENTS

We would like to thank Neoklis Polyzotis for useful comments and discussions regarding the issues raised in Section 5. Also, we would like to thank Mihalis Nicolaou for implementing the algorithms for structural matching [26].

10. REFERENCES

[1] FreePastry 2.1 release. `http://www.freepastry.org/FreePastry/`.

[2] IBM XML Generator. `http://www.alphaworks.ibm.com/xmlgenerator`.

[3] YFilter 1.0 release. `http://yfilter.cs.umass.edu/code_release.htm`.

[4] I. Aekaterinidis and P. Triantafillou. PastryStrings: A Comprehensive Content-Based Publish/Subscribe DHT Network. In *ICDCS 2006*.

[5] D. Barbosa, L. Mignet, and P. Veltri. Studying the XML Web: Gathering Statistics from an XML Sample. *World Wide Web*, 9(2):187–212, 2006.

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

[7] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. Index-Based XML Multi-Query Processing. In *ICDE 2003*.

[8] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *ICDE 2002*.

[9] C. Y. Chan and Y. Ni. Efficient XML Data Dissemination with Piggybacking. In *SIGMOD 2007*.

[10] R. Chand and P. Felber. Scalable Distribution of XML Content with XNet. *IEEE Transactions on Parallel and Distributed Systems*, 19(4):447–461, 2008.

[11] R. Chand and P. A. Felber. A Scalable Protocol for Content-Based Routing in Overlay Networks. In *NCA 2003*.

[12] J. Clark and S. J. DeRose. XML Path Language (XPath) Version 1.0. World Wide Web Consortium, Recommendation, November 1999.

[13] M. P. Consens and T. Milo. Optimizing Queries on Files. In *SIGMOD 1994*.

[14] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM TODS*, 28(4), 2003.

[15] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *VLDB 2004*.

[16] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[17] P. Felber, C.-Y. Chan, M. Garofalakis, and R. Rastogi. Scalable Filtering of XML Data for Web Services. *IEEE Internet Computing*, 7(1), 2003.

[18] X. Gong, W. Qian, Y. Yan, and A. Zhou. Bloom Filter-Based XML Packets Filtering for Millions of Path Queries. In *ICDE 2005*.

[19] A. K. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *SIGMOD 2003*.

[20] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[21] S. Hou and H.-A. Jacobsen. Predicate-based Filtering of XPath Expressions. In *ICDE 2006*.

[22] G. Koloniari and E. Pitoura. Content-based Routing of Path Queries in Peer-to-Peer Systems. In *EDBT 2004*.

[23] J. Kwon, P. Rao, B. Moon, and S. Lee. FiST: Scalable XML Document Filtering by Sequencing Twig Patterns. In *VLDB 2005*.

[24] J. Kwon, P. Rao, B. Moon, and S. Lee. Value-based Predicate Filtering of XML Documents. *Data and Knowledge Engineering*, 67(1):51–73, 2008.

[25] G. Li, S. Hou, and H.-A. Jacobsen. Routing of XML and XPath Queries in Data Dissemination Networks. In *ICDCS 2008*.

[26] I. Miliaraki, Z. Kaoudi, and M. Koubarakis. XML Data Dissemination using Automata on Top of Structured Overlay Networks. In *WWW 2008*.

[27] M. M. Moro, P. Bakalov, and V. J. Tsotras. Early Profile Pruning on XML-aware Publish/Subscribe Systems. In *VLDB 2007*.

[28] F. Peng and S. S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD 2003*.

[29] T. Pitoura and P. Triantafillou. Load Distribution Fairness in P2P Data Management Systems. In *ICDE 2007*.

[30] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware 2001*.

[31] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using XML. *SIGOPS Oper. Syst. Rev.*, 35(5), 2001.

[32] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymaki. Implementing a scalable XML publish/subscribe system using relational database systems. In *SIGMOD 2004*.

[33] C. Tryfonopoulos, S. Idreos, and M. Koubarakis. Publish/Subscribe Functionality in IR Environments using Structured Overlay Networks. In *SIGIR 2005*.

[34] H. Uchiyama, M. Onizuka, and T. Honishi. Distributed XML Stream Filtering System with High Scalability. In *ICDE 2005*.

[35] Z. Vagena, M. M. Moro, and V. J. Tsotras. Value-Aware RoXSum: Effective Message Aggregation for XML-Aware Information Dissemination. In *WebDB 2007*.

[36] J. S. Vitter. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.