# Early Profile Pruning on XML-aware Publish-Subscribe Systems

Mirella M. Moro
University of California,
Riverside, CA 92521, USA
mirella@cs.ucr.edu

Petko Bakalov
University of California,
Riverside, CA 92521, USA
pbakalov@cs.ucr.edu

Vassilis J. Tsotras*
University of California,
Riverside, CA 92521, USA
tsotras@cs.ucr.edu

## ABSTRACT

Publish-subscribe applications are an important class of content-based dissemination systems where the message transmission is defined by the message content, rather than its destination IP address. With the increasing use of XML as the standard format on many Internet-based applications, *XML aware* pub-sub applications become necessary. In such systems, the messages (generated by publishers) are encoded as XML documents, and the profiles (defined by subscribers) as XML query statements. As the number of documents and query requests grow, the performance and scalability of the matching phase (i.e. matching of queries to incoming documents) become vital. Current solutions have limited or no flexibility to prune out queries in advance. In this paper, we overcome such limitation by proposing a novel early pruning approach called *Bounding-based XML Filtering* or *BoXFilter*. The *BoXFilter* is based on a new tree-like indexing structure that organizes the queries based on their similarity and provides lower and upper bound estimations needed to prune queries not related to the incoming documents. Our experimental evaluation shows that the early profile pruning approach offers drastic performance improvements over the current state-of-the-art in XML filtering.

## 1. INTRODUCTION

Publish-subscribe applications (or simply pub-sub) are an important class of asynchronous content-based dissemination systems where the message transmission is defined by the message content, rather than its destination IP address. Examples of pub-sub systems include notification websites (such as *hotwire.com* or *ticketmaster.com*), where a user can subscribe for events of interest (new deals on their favorite travel destinations, concerts of their favorite artists in their local area etc.) and get automatic notifications when relevant events arrive in the system.

Many pub-sub systems have already been proposed with different architectures (centralized within a server [1, 9, 11] or distributed over a network of brokers [5, 8]) but they all follow the same asynchronous event-based communication paradigm. The events are announced with a *message* generated outside of the system by third-party applications referred as *publishers*. These messages are then selectively delivered to interested *subscribers* that have announced their interest by submitting *profiles* to the system. At the center of a pub-sub system is the *matching process*, which is in charge of finding (filtering) which messages satisfy which profile subscriptions. Figure 1 shows the interaction of the main components within a pub-sub system.

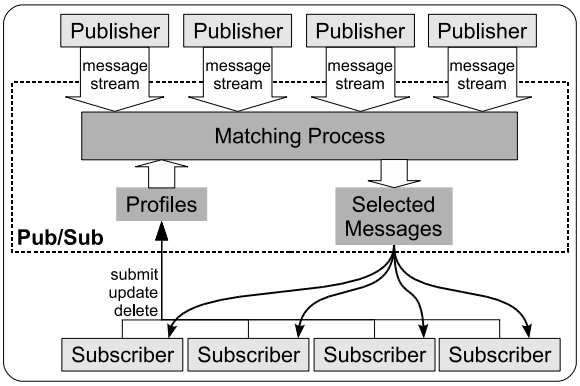



**Figure 1: Architecture of a pub-sub system.**

Regarding message content and profile definitions, pub-sub systems have evolved from simple topic-based communication [24, 35], to predicate-based systems [10, 11], to recently designed *XML-aware* systems [7, 8, 9, 18, 36, 37]. Given the adoption of XML as the standard format for data exchange, in this paper we focus on the XML-aware pub-sub systems. In such scenario, messages are encoded as XML documents and the profiles (query subscriptions) are expressed using XML query languages, such as XPath [3].

The efficiency of a pub-sub system is heavily dependable on how effective the *matching process* is, especially under very large volumes of messages and profiles. There have been three predominant strategies to design the matching process. The first one utilizes the standard relational approach and translates profiles and messages to the Relational Model [34]. Then, the matching can be expressed as join operations between the sets of messages and profiles. The second one is to aggregate the profiles using some indexing technique [6]. The matching process then reads the input message and traverses the index in order to select the profiles satisfied. Finally, the most common approach is based on Finite State Machines (FSM) [1, 9, 14, 17, 28, 36, 37], where the techniques differ in the type of FSM used. One or more FSMs are built and enable processing the common parts of the queries only once. Typically the processing is performed in a top-down fashion.

*Research partially supported by UC Micro, Lotus Interworks and NSF IIS-0534781


*VLDB '07,* September 23-28, 2007, Vienna, Austria.

A different perspective is to evaluate the XML document in a bottom-up manner, as presented in [20]. Such approach takes advantage of the (usual) fact that an XML document has its more selective elements located at its leaves. Hence, a bottom-up evaluation of both message documents and profile queries improves the matching performance by considering less queries than the traditional approaches that use top-down approaches. Moreover, a sequence matching technique may be employed, as opposed to the previous FSM-based ones.

All the previous approaches provide different advantages for various types of queries. However, they all have the same limitation: they do not provide any *early pruning* feature in their matching algorithm so as to quickly identify (and thus discard) queries that are bound not to match any documents. Considering the increasing volume of incoming message documents and incoming queries, an early pruning technique is essential for saving processing time on the matching process.

In this paper, we propose a new matching approach that focuses on early pruning. The approach is based on a new tree-like indexing structure that organizes the query requests based on their similarity (using their Prüfer code [29]) and provides lower bound estimation needed to prune queries not related to the incoming documents. Specifically, the contributions of this paper are as follows.

- We propose an index-based filtering technique, named *BoXFilter* (*Bounding-based XML Filtering*), that performs early pruning of the query profile space. The index enables a very efficient search for a match during the filtering process by grouping similar queries and by using lower and upper bounds to prune out many of those queries. The index structure is also able to accommodate a huge volume of profiles, even when exceeding the memory space available. Furthermore, the incoming documents can also be stored on a similar structure such that the matching phase can take advantage of batch processing as well.

- As a by-product of this research, we propose a new FSM-based approach, named *BUFF* (*Bottom-Up Filtering FSM*), that improves the performance of regular FSM by performing a bottom-up evaluation of the document. Hence, BUFF takes advantage of the bottom-up processing against the typical top-down processing of FSM-based methods. Moreover, it serves as a good representative of the bottom-up processing class in our experimental evaluation.

- We perform an extensive experimental evaluation that compares the performance of the BoXFilter with the state-of-the-art methods. This includes both top-down (such as Yfilter [9]) and bottom-up (BUFF) approaches. We consider a wide range of datasets and queries by varying different parameters that can influence the algorithms performance. We are able to verify the scalability and robustness of our new approach regarding the number of queries as well as the number of documents evaluated. Our results show that the early pruning of BoXFilter provides much better filtering performance over the previous solutions.

We proceed by presenting background information in Section 2. Then, section 3 details our new FSM-based approach (BUFF), while section 4 presents the BoXFilter approach. Section 5 provides the experimental study. Finally, section 6 presents related work, and section 7 concludes the paper.

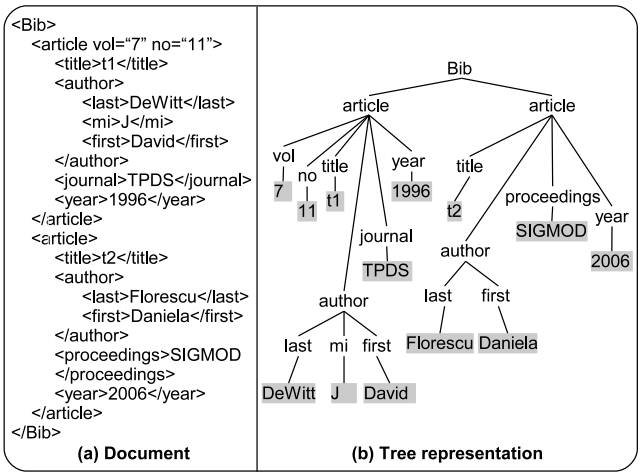



**Figure 2: XML document and its tree representation.**

## 2. BACKGROUND

**XML Data and Queries.** An *XML document* is formed by a sequence of elements that enclose text values and other elements. The XML document is typically represented by a tree structure where the *nodes* correspond to elements, attributes or text values, and the *edges* represent immediate element-subelement or element-value relationships [3]. An *XML database* is then modeled as a forest of unranked, node-labeled trees, one tree per document [3]. An example of this model is illustrated in Figure 2.

Data can be retrieved from an XML document by an *XML query language* (e.g. XPath [3], XQuery [4]). XML query languages consider the inherent structure of the data and enable querying on both structure and simple values. The structural constraints are usually specified by a *path expression* that may contain *predicates* for further refinement. For example, consider the path expression:

*//article[/author[@last="Smith"]]//procs[@conf="VLDB"]*

that requests all proceedings of articles that have an author with last name "Smith" and have appeared in a "VLDB" conference. This query consists of two types of conditions. First, *@last="Smith"* and *@conf="VLDB"* are *value-based* and select elements or attributes according to their values. Second, the path (or twig) *//article[/author]//procs* defines structural constraints as it imposes restrictions on the structure of the retrieved elements (e.g. a *procs* element must exist under an *article* that has least one *author* element as child).

The previous query example also shows the types of constraints we consider in this work. Specifically, we assume that the query expressions are formed by a path composed of elements that related to each other through ancestor-descendant axis ("//") and parent-child axis ("/") and by predicates that specify nested paths and branches as well as value constraints on both attributes and elements.

**Pub-Sub Systems.** Usually, a pub-sub system has a distributed architecture over a set of network nodes (brokers). Each broker is connected to a set of publishers (that inject messages through this broker), a set of subscribers (that are awaiting notifications) and other brokers within the network. We assume that the network of brokers is already specified. The process for building the network and the routing among the brokers is out of the scope of this paper and has been covered elsewhere (e.g. [33]). For our purposes, we focus on the matching process that takes place within a broker.

Every broker has a routing table that keeps information regarding the profile subscriptions and the messages destinations (possibly accessed via other brokers through the network).

A central function within an XML-aware pub-sub system is to perform *XML filtering* efficiently, i.e. matching the profile queries to the message documents. This is a more crucial issue than in topic- or predicate-based pub-sub systems, since the profiles are now more complex. Note that XML filtering differs significantly from the typical XML query processing, in which the main goal is to find the specific parts within an XML document where a query pattern occurs. Such "pattern discovery" is typically done in static environments where the documents are indexed for fast processing of the incoming queries (e.g. [19, 23, 25]). In the XML filtering context, the roles of the query and the document are reversed. Also, the documents usually arrive in the system in higher rates than the queries. Therefore, the query patterns are the ones to be indexed in order to determine which documents need to be filtered.

**Prüfer Sequence.** Prüfer sequence (or Prüfer code) is a term used in graph theory to describe a unique sequential encoding of a labeled tree [29]. The Prüfer sequence for a tree with $n$ nodes has length $n-2$, and can be generated by a simple algorithm. The algorithm iteratively removes nodes from the tree until all nodes but the last two have been removed. At each iteration, the algorithm finds and removes the leaf with the smallest label and adds to the Prüfer sequence the label of that leaf's parent.

Recently, Prüfer sequences have been successfully used in the XML domain in combination with a tree numbering scheme [20, 30]. The numbering scheme (typically a preorder) associates unique labels to the document tree nodes which are then processed to define the Prüfer code. There are two minor differences between the Prüfer code generated for XML documents and the original definition from graph theory. First, in the original definition of the Prüfer encoding, the leaf nodes do not participate in the sequence. Only nodes which have at least one child form the Prüfer sequence for the tree. To overcome this problem when processing XML data, the document tree is expanded by adding an artificial child node to each leaf in the tree (the tree then grows in height by one level). Second, in the XML domain, the deletion of nodes from the tree continues until only one node is left.

The sequence formed by the document numbering scheme is called *Numbered Prüfer Sequence* or NPS. The sequence formed by employing the actual XML tags in the encoding is called *Labeled Prüfer Sequence* or LPS. In this paper, we use the term *Prüfer sequence* to refer to the LPS of an XML document. Moreover, we define the LPS of a document by employing a SAX parser [31] to read the document and to create the sequences. For example, Algorithm 1 illustrates how Prüfer sequences can be generated for an XML document by reading that document with a SAX parser.

The following theorem guarantees the successful employment of the Prüfer encoding on XML query matching (we refer to [30] for its proof).

THEOREM 1. *If a query tree $Q$ is a subgraph of a document tree $D$ then the Prüfer encoding of $Q$ is a subsequence of the Prüfer encoding of $D$.*

It is possible to have subsequence matching between two Prüfer encoding of trees $D$ and $Q$ without $Q$ being a subgraph of $D$. In other words, Theorem 1 guarantees having no false dismissals but it is possible to have false positives. The main reason is that the Prüfer does not consider the structural features of neither the query nor the XML document. Therefore, after matching the Prüfer sequences from the queries and the documents, a post-processing phase is necessary to filter out the false positives.

**Algorithm 1** Prüfer code generation

**Require:** document $D$
**Ensure:** The Prüfer $V$ sequence encoding of $D$
1: Set $V$ = 0 ▷ the resultant Prüfer sequence
2: Set $S \leftarrow \emptyset$ ▷ auxiliary stack
3: saxParser.parse($D$)
4: **procedure** STARTELEMENT ▷ SAX parameters omitted for clarity
5: $S.push(node)$ ▷ $node$ is the element read
6: **procedure** ENDELEMENT ▷ SAX parameters omitted for clarity
7: node = $S.pop()$
8: $V$.append(node)
9: **if** the current element is leaf **then**
10: node = $S.pop()$
11: $V$.append(node)

# 3. BOTTOM-UP FILTERING FSM (BUFF)

FSM-based approaches are among the most common methods for the XML matching process, such as the *XFilter* [1] and the *YFilter* [9]. This type of approach evaluates the input document in top-down fashion (i.e. in-order or depth first order) while advancing the state machine for each XML element (or attribute) read. Even though they provide some performance guarantees (as for example, requiring one pass over the documents), they also need to evaluate all queries because they do not consider any form of early pruning.

On the other hand, FiST [20, 30] has recently indicated the advantages of a bottom-up approach for XML filtering. This approach takes into consideration the (usual) fact that an XML document has its more selective elements located at its leaves. Hence, such a bottom-up evaluation can improve the matching performance by considering less queries than a top-down approach.

When developing our early profile pruning solution (BoXFilter), we wanted to compare its performance with the best current solutions for both top-down and bottom-up approaches. Influenced by the FiST performance results, we decided to create an FSM-based technique that provides the bottom-up functionality as well. The result is the BUFF approach discussed next.

It is important to note that while BUFF also applies a bottom-up evaluation, there are various differences from FiST. For example, FiST translates the document and the queries to Prüfer sequences. Having the Prüfer sequences, it employs specialized data structures and algorithms to perform a progressive subsequence matching of the incoming streams of data. The main goal of our new FSM-based technique is to add an optimized bottom-up evaluation to the regular FSM without any unnecessary step. Therefore, BUFF avoids translating documents and queries to Prüfer sequences, and employs a more direct evaluation algorithm.

## 3.1 The Automaton

As the experimental evaluation will show, the BUFF approach takes advantage of the fact that deeper parts of an XML document are usually more selective than the upper parts due to the presence of optional elements. In other words, the selectivity of the document leaves is usually higher than the intermediate elements. However, it is important to notice that by changing the order in which the document is evaluated, we also need to change the order in which the query is evaluated. Therefore, instead of evaluating the query from the query root to the query leaves, we evaluate it from the query leaves to the query root. As a result, the respective FSM will be different from the ones used by previous approaches such as *XFilter* and *YFilter*.

The algorithm for building BUFF is similar to that of [9], however, the queries are inserted in their reverse order. For example, the query *//a/b/c//d* is inserted in BUFF as *//d//c/b/a*. Furthermore,

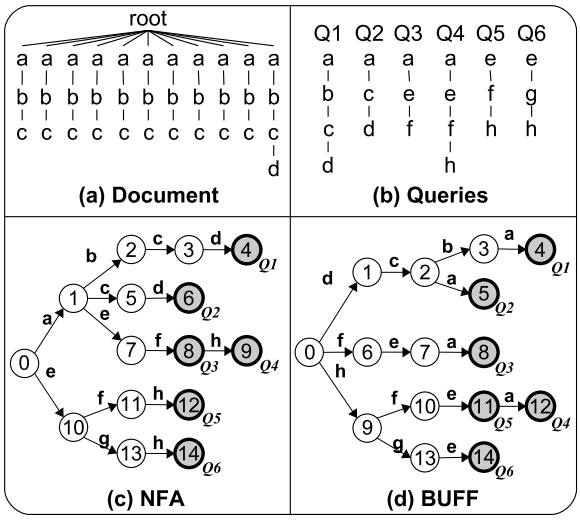



**Figure 3: An XML document, a set of queries and the respective regular simplified NFA, and reverse NFA employed by BUFF (states as circles, final states as gray circles, main transitions as arrows).**

**Algorithm 2** BUFF processing

**Require:** $D$: document, $B$: BUFF
**Ensure:** The set of profiles $S$ which are satisfied by $D$
1: saxParser.parse($D$)
2: forward $D$ to the subscriber of each query within $S$
3: **procedure** STARTELEMENT ▷ SAX parameters omitted for clarity
4: $curList \leftarrow B.initialState$
5: $RS.push(node, curList)$ ▷ $node$ is the element read
6: **procedure** ENDELEMENT ▷ SAX parameters omitted for clarity
7: $curList \leftarrow RS.top().stateList$
8: $newList \leftarrow B.move(node, curList)$ ▷ BUFF transition
9: $RS.pop()$
10: $RS.top().stateList.add(newList)$
11: **procedure** MOVE(node,stlist) ▷ BUFF machine transitions
12: **for all** states $s \in stlist$ **do**
13: $newState \leftarrow B.nextState(node, s)$
14: $result.add(newState)$
15: **if** $newState$ is final state **then** $S.add(queriesIdentifiers)$
16: **return** $result$

if the query starts at the root (e.g. */a/b/c*) then an additional boolean constraint is added at the final state of the NFA to check whether that element is the root or not.

For example, Figure 3b illustrates a set of path queries. Assume that, for simplicity, all queries consider only ancestor/descendant axis. The respective simplified NFA for that set of queries is illustrated in Figure 3c and the reverse NFA employed by BUFF is in Figure 3d. Both state machines have the same number of states, transitions and final states (note that only the main transitions are illustrated for clarity reasons). The difference between the machines is that one has the queries represented in a top-down order (i.e. from the query root to the query leaves) and the other has them in a bottom-up order (i.e. from the query leaves to the query root). Note also that while the regular NFA groups the queries according to their common prefixes, BUFF groups them according to their common suffixes.

## 3.2 The Automaton Matching Process

The intuition for the BUFF processing is that a bottom-up evaluation of a document should trigger less states than the traditional top-down approach. For example, considering the input document from Figure 3a, the top-down NFA (Figure 3c) executes transitions to states 1, 2 and 3 (after reading elements $a$, $b$ and $c$) eleven times (once for each path -a-b-c) before achieving the final state 4 for query $Q1$ (i.e. *//a//b//c//d*). On the other hand, *BUFF* executes transitions to states 1, 2 and 3 only once (when reading the last path of the document tree, -a-b-c-d), then achieves the final state 4 for the same query. Similar situation happens for query $Q2$.

**Structural Constraints.** The algorithm to process the query matching in BUFF is different than that proposed in [9]. More specifically, the document is parsed through a SAX parser [31], which defines events for specific marks in the XML document such as *startElement* when an opening tag is read (e.g. <element>) and *endElement* when a closing tag is read (e.g. </element>). The machine keeps a runtime stack $RS$ that stores the current document path being processed. In this case, for each opening tag, the respective element $e$ is pushed to $RS$. Similarly, for each closing tag, an element $e$ is popped from $RS$ and is employed to trigger a set of transitions $T$ (all transitions defined by reading that respective element) within the NFA. The main function of the stack $RS$ is to store the elements until they can be processed in a post-order (or bottom-up) manner. The size of the stack is bound by the height of the document tree. Algorithm 2 illustrates the process.

Note that keeping only the elements within the runtime stack is not enough. It is also necessary to maintain the transitions that those elements trigger, so that the machine works properly. With a top-down evaluation, that is easily done by keeping a list of current states, as informed in [25]. At each opening tag, an element is pushed to the runtime stack and employed to trigger the machine transitions, then updating both the runtime stack and the list of current states. On the other hand, with a bottom-up processing, one list of current states is not enough because an element triggers the state machine when its closing tag is read.

BUFF overcomes that problem by keeping one list of states per element in the runtime stack (each time an element is pushed to the stack, it is also associated to a list of states that contains only the NFA initial state). For example, figure 4a illustrates an input document and two queries already stored in a BUFF machine. Then, figures 4b through 4g illustrate different stages of the stack when the following tags are read: $<e_8>$, $</e_8>$, $</d_7>$, $</f_{10}>$, $</e_9>$ and $</c_8>$. As the opening tags for elements $a_1$ to $d_4$ are read, each element is pushed to the stack (line 5). The closing tag $</d_4>$ simply pops element $d$ from the stack since no transition is defined for that element (line 7-9). The same happens when reading $</c_3>$ and $</b_2>$. Then elements $b_5$ to $e_8$ are read and pushed to the stack. The first element to trigger a BUFF transition is $</e_8>$, which moves the machine to state 1 (line 8). Then, its current state list is stored at the top of the stack, i.e. in element $d$ (line 10). Likewise, $</d_7>$ moves the machine to state 2, which is then sto red at the top of the stack (element $c$). Note that when $f_{10}$ is popped from the stack, element $c$ has its own state list and element $e$ defines a new one to store the state list defined by element $f$ (figure 4f). Finally, as the closing tags for the remaining elements are processed, the state lists are updated, and the machine keeps moving until it gets to the final states. When a final state is reached, the identifiers of the queries defined for that state are added to the machine results (line-15).

**Value Constraints.** When the query contains predicates with value constraints, the respective values are also stored on the BUFF states. A considerable large number of value predicates can be added to the NFA state by using an auxiliary structure (e.g. a Bloom Filter [37]). In this case, the NFA only advances to the next

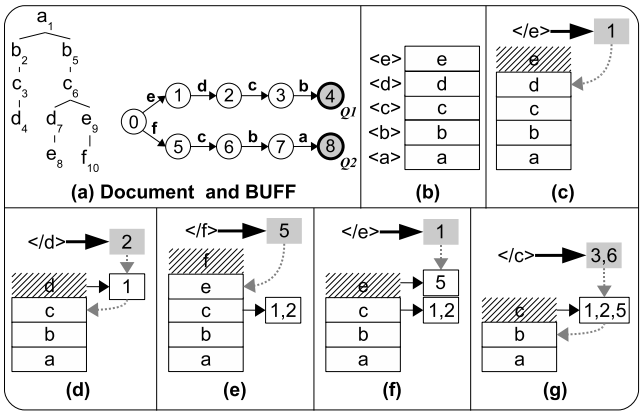



**Figure 4: Example of BUFF matching: different stages of the runtime stack.**

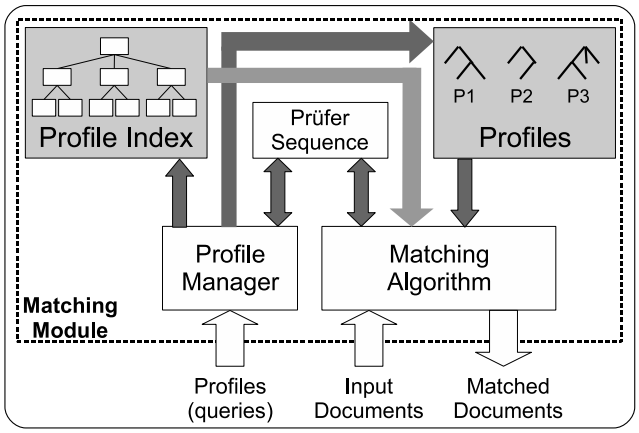



**Figure 5: Matching module components.**

state when the value constraint is also satisfied. Note that the SAX parser has a specific procedure (called *characters*) that deals with the value of an element. The values of an attribute are accessed within the *startElement* function (not shown for clarity reasons).

# 4. BOUNDING-BASED XML FILTERING (BOXFILTER)

We proceed with an overview of the core components in the *BoXFilter* approach. Then, we describe our novel indexing scheme that employs a holistic approach based on the Prüfer sequence representation of the profile patterns. Finally, we present the filtering algorithms for both the streaming and the batch processing mode that utilize this index. While BoXFilter translates documents and queries to Prüfer sequences like FiST [20], it is a different approach because it employs a novel pruning technique based on lower and upper bound estimates, such that it reduces the query space considerably. As it will be verified in the experimental section, such pruning provides drastic performance improvements.

## 4.1 BoXFilter Core Modules

An overview of the main modules and the data flows in the *BoXFilter* is depicted in figure 5. There are two major processes working asynchronously and in parallel: the profile management and the matching algorithm.

**Profile Management**. The first process deals with the profiles (expressed as XPath queries) and the profile modification requests that arrive in the Profile Manager module. During the first step of this process, the incoming profiles are parsed using an XPath parser and transformed into a Prüfer sequence code using the algorithm described in section 2. In order to generate small sequences, the tags in the XPath query are replaced with more compact symbolic representation (e.g. for example the tag "author" is replaced with the symbol $A$, the tag "book" is replaced with the symbol $B$ and so on). The mapping is then kept in a compact dictionary structure.

The Prüfer encoding of the profile is then inserted in a tree-based indexing structure called *BoXFilter tree* (described in section 4.3). The original profiles expressed in XPath and the destination address (where the message has to be forwarded to) are stored in the routing table. The information in the routing table is used by the filtering algorithms for verification and routing purposes. The *BoXFilter* tree and the routing table form the core data structures in the *BoXFilter* approach.

**Matching Algorithm**. The second process handles the actual document filtering. First, the stream of message documents arrives in the Matching Algorithm module (where the matching process is performed). Each incoming document is parsed and transformed into Prüfer sequence encoding in the same way as the profiles. Then, the filtering algorithm locates those Prüfer sequence encoding of profiles that are subsequences of the sequence encoding of the document (i.e. it performs a subsequence matching). This step is done by traversing the *BoXFilter* tree. According to the Theorem 1 in section 2, if a profile encoding is not a subsequence of the document encoding, the corresponding XPath query is not a subtree of the original document tree. Hence, we can focus only on those profiles returned by the subsequence matching. Those profiles are called *candidate profiles*, because even though the subsequence matching between the document and the profile encodings is a necessary condition, it is also not sufficient (the subsequence matching does not consider the structural constraints of the query). Then, the *BoXFilter* approach verifies the candidate profiles in order to guarantee that they also satisfy the query structural constraints. Finally, the documents that matched a profile are routed to the respective profile's destination.

## 4.2 Sequence Envelope

The *BoXFilter* index tree is based on the concept of a *Sequence Envelope* that is introduced next. For simplicity, assume that all sequence encodings of profiles have the same length $l$ (the extension to cover sequences with different lengths is straightforward). Consider a set of $k$ sequences representing Prüfer encoding of profiles, $S1, .., Sk$. We propose that this set of sequences can be employed to derive two new sequences (with length $l$) called the *upper* and the *lower bound*, or $U$ and $L$ respectively. The $L$ sequence is derived as follows: for each position $i$ (for all $i$ from 1 to $l$), the element in position $i$ is the smallest element on this position for all $k$ sequences. Likewise, the $U$ sequence is derived by taking the largest element for each position. Therefore:

$$L_i = min(S1_i, .., Sk_i)$$

$$U_i = max(S1_i, .., Sk_i)$$

In order to determine which one is the smallest and which one is the largest element, we use alphabetical order (however, other orderings may be employed as well). Figure 6 illustrates the structural constraints of three profiles. The Prüfer sequence encoding for

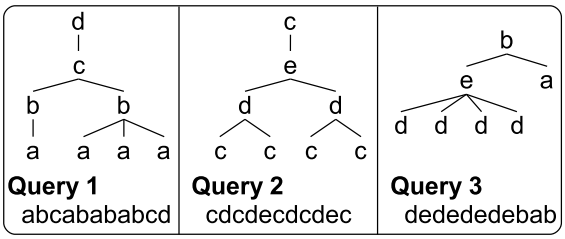



**Figure 6: Example of query profiles.**

each profile is also shown; all sequences have length 11 elements. The upper and the lower bounds for this set of three sequences are:

$$L = ABCABABABAB$$

$$U = DEDEEEDEDED$$

Note that $L$ and $U$ form the smallest possible bounding envelope that encompasses all members of the set of sequences $S1, .., Sk$. Figure 7 illustrates a visual interpretation of the bounding envelope for these three profile encodings. The profiles are shown as sequences in a 2-dimensional space, where one dimension corresponds to the discrete set of symbolic values and the other to all possible positions in a sequence. As a result, $\forall i \;\; L_i \leq S1_i, \cdots Sk_i \leq U_i$. We use the term *sequence envelope* for the combination of $L$ and $U$, and denote it as $SE$:

$$SE \equiv (L, U)$$

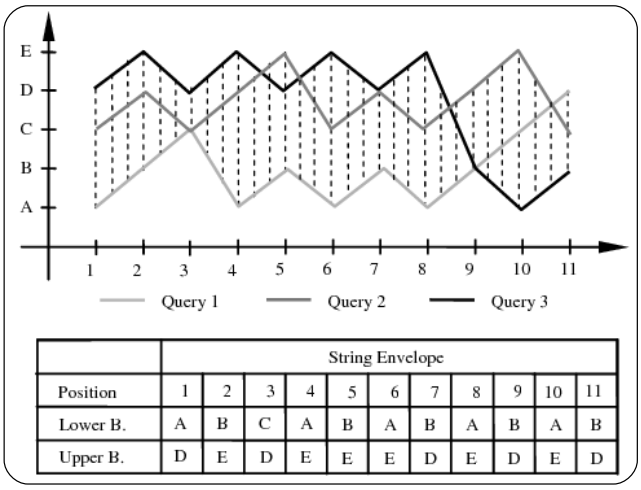
An important property of the sequence envelope structure is that it can be used as an aggregation of the sustaining set of sequences. Suppose that we have a Prüfer sequence encoding of a document $D$ to be matched against the set of profile encodings $S1, .., Sk$. The profile encodings can be combined into a single sequence envelope $se$. Then, the document encoding $D$ can be matched against $se$. If there is no subsequence $D_{j1}, D_{j2}, \cdots, D_{jl}$ of length $l$ in the document $D$, such that every element in the subsequence is between the lower and upper bound on the corresponding position in the sequence envelope ($\forall i \;\; L_i \leq D_{ji} \leq U_i$), then there is no subsequence matching between the document and any of the profile encodings $S1, .., Sk$. We can thus avoid the costly comparison between the document encoding $D$ and each profile sequence $Si$. The above discussion is summarized in the Lemma below (proof is omitted).

LEMMA 2. *If there is no subsequence matching between the Prüfer sequence of the document $D$ and a sequence envelope $SE$, then there is no matching between $D$ and any of the queries whose sequences are within $SE$.*

Another important property of sequence envelopes is that they can be nested. This property allows the creation of a hierarchical index structure over the envelopes, where a parent node in the index tree has an envelope that contains all the envelopes of its children. This property emanates from the fact that a single sequence is a special case of a sequence envelope where both the upper and lower bounds are identical to the sequence ($\forall i \; L_i = S_i = U_i$). Thus a set of sequence envelopes $SE1, .., SEk$ can be combined into a single one by finding maximum and minimum values for each position, from all envelopes, such that:

$$L_i = min(L1_i, .., Lk_i)$$

$$U_i = max(U1_i, .., Uk_i)$$




| | String Envelope | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Lower B. | A | B | C | A | B | A | B | A | B | A | B |
| Upper B. | D | E | D | E | E | E | D | E | D | E | D |

**Figure 7: Sequence envelope.**

## 4.3 BoXFilter Tree

The *BoXFilter* index organizes efficiently the Prüfer encoding of the system profiles based on their similarity. We utilize the nesting property of the sequence envelopes to organize them hierarchically into a height balanced tree structure, similar to an R-tree [16, 21]. The tree structure splits the value/position space into hierarchically nested, and possibly overlapping, sequence envelopes.

The nodes in the *BoXFilter* tree have a variable number of elements (between some pre-defined minimum and maximum values). Each leaf node in the *BoXFilter* tree stores a profile encoded as a Prüfer sequence as well as a pointer to the routing table where its original description is stored. Each entry within a non-leaf node has two fields: a pointer to a child node, and the bounding envelope of all entries within this child node. The proposed structure is *dynamic*, which means that insert and delete operations can be intermixed with search ones.

**Tree Search**. The matching operation in the BoXFilter tree is done in way similar to the R-tree family. However, here we are interested in subsequence matching as opposed to overlapping of MBR rectangles. The input to the algorithm is a search sequence $R$ while the output is the set of sequences $S1, .., Sk$ in the leaves of the BoXFilter tree such that $Si$ is a subsequence of the search sequence $R$. The algorithm traverses the index tree from the root to the leaf nodes. At each step, the algorithm checks the bounding envelopes at the current node: if there is subsequence matching between a bounding envelope and the search sequence $R$, the algorithm is executed recursively for its corresponding subtree. Due to the overlapping sequence envelops, the algorithm may need to visit more than one subtree under the current node. In the worst case, the algorithm may have to traverse the whole tree. Nevertheless, this scenario is quite infrequent. In our experimental evaluations, the search algorithm was able to eliminate large portions of the value/position space, resulting in fewer comparisons performed.

**Envelope Insertion**. Insertions in the BoXFilter tree follow the R-tree approach. The algorithm examines the bounding envelopes in the non-leaf nodes, so as to find an envelope that overlaps with the new sequence. If there are more than one envelopes that overlap the new sequence, then the algorithm picks the one with smallest number of children. If there is no such envelope, the algorithm chooses the one that needs least enlargement to include the new sequence. After choosing an appropriate envelope, the algorithm is executed recursively for the corresponding subtree. When the leaf

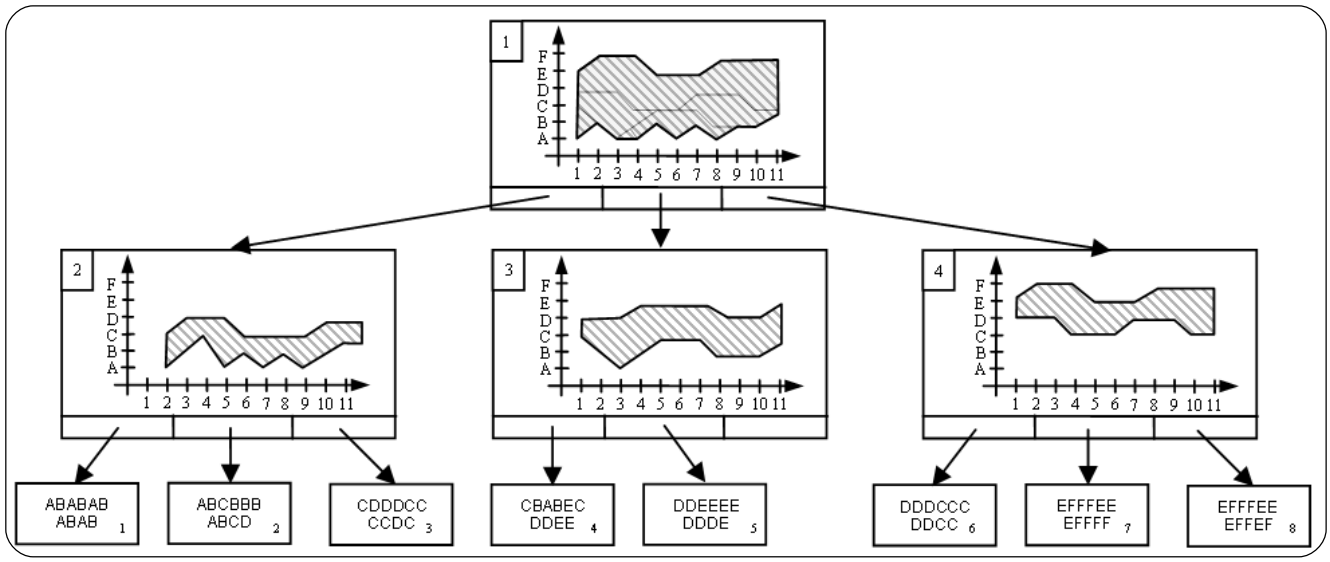



**Figure 8: Sequence envelope tree.**

**Table 1: Routing table for sequence envelope tree.**

| | Routing Table |
|---|---|
| 1 | /Book/Author[//Author][/Author][//Author][//Author] |
| 2 | /Book/Corrector//Chapter/[Author]/[Corrector]//[Chapter]/Author |
| 3 | Corrector/Book[//Corrector][/Book]//Corrector[//Corrector]/Corrector |
| 4 | Publisher//Publisher/Book[//Publisher/Chapter[//Corrector]/Author]/Book//Corrector |
| 5 | Publisher//Book[/Book][//Publisher[/Publisher]//Publisher]/Book |
| 6 | Corrector/Corrector[Book//Book/[Book]/Book] [/Corrector]/Corrector//Corrector |
| 7 | Distributor//[Distributor]/Distributor/Publisher[//Distributor][//Publisher//Publisher//Publisher] |
| 8 | Distributor//[Publisher]/Distributor/Publisher[//Distributor][//Publisher//Publisher//Publisher] |

level is reached and there is space available in the current node, the new element is inserted there.

If the leaf node is full, it splits to accommodate the new envelope. In this case, we adopt the R*-tree approach [21] to determine the distribution of the sequence envelopes between the nodes which minimizes the overlapping between the sequence profiles. The intuition behind is that the probability for two sequence envelopes to satisfy simultaneously the subsequence criteria is proportional to their overlapping area. The process of splitting nodes is propagated up to the root of the tree as necessary.

**Envelope Deletion**. For removing an entry (envelope) from the BoXFilter tree, the first step is to locate the leaf where it is contained and then delete it. If removing such an entry from the index causes the leaf to underflow, then the remaining entries are redistributed and the leaf node itself is deleted. Such node elimination is propagated upward as necessary. An envelope update is simply treated as a deletion followed by an insertion.

**Discussion**. There are two important advantages of the BoXFilter tree structure. The first one is its dynamic nature, where insert and delete operations can be intermixed with search ones. Note that, this differentiates the BoXFilter from the existing NFA-based structures, which first go through a phase of construction (insertions) and then become operational (for searches). The second advantage lies in its scalability. The BoXFilter tree structure can efficiently handle situations where the available main memory is not sufficient to accommodate all its entries (in this case, the profile queries). In such situations, the nodes of the BoXFilter tree can be paged to secondary storage and loaded upon request.

## 4.4 Filtering algorithms

Depending on how documents are processed, there are two variations of the filtering algorithm: (i) sequential, and (ii) batch processing. We illustrate the filtering process using an example. Assume a pub-sub system which informs its subscribers about newly published books. There are eight profiles submitted to the system, as described by the routing table in Table 1. The original XML tags in the profiles are substituted with symbols using the mapping shown in Table 2. Then the profiles are transformed into Prüfer sequences and are inserted into the BoXFilter tree shown in Figure 8. For each index node, the figure shows the sequence envelope that contains all envelopes from its children.

**Sequential Processing**. In this mode of matching, the incoming documents are processed sequentially, one after the other. All profiles are encoded in Prüfer sequences and organized in a BoXFilter structure. This matching process is described by Algorithm 3.

The input to the sequential matching algorithm is the root of the tree and a document. The first step is to replace the original XML tags and to convert the document into Prüfer sequence encoding (line 2). Assume that the document processed by the system has the following Prüfer code:

$$D = ABCFABABABABF$$

**Algorithm 3** Document Filtering (sequential mode)

**Require:** $\bar{D}$: document, R: BoXFilter tree root, T: routing table
**Ensure:** The set of profiles $\bar{S}$ which are satisfied
1: Set $\mathcal{S} \leftarrow \emptyset; \mathcal{N} \leftarrow R$,
2: D = ConvertToPruferSeq($\bar{D}$)
3: **while** $\mathcal{N}$ is not empty **do** ▷ BoXFilter tree traversal
4: C = $\mathcal{N}$.pop()
5: **for** each child E of node C **do**
6: **if** E is a leaf **then**
7: **if** E's sequence is a substring of D **then**
8: $\mathcal{S}$.push(E)
9: **else**
10: **if** E's sequence envelope is a substring of D **then**
11: $\mathcal{N}$.push(E)
12: **end for**
13: **end while**
14: **while** $\mathcal{S}$ is not empty **do** ▷ Verification step
15: S = $\mathcal{S}$.pop()
16: $\bar{S}$ = T.LookUp(S)
17: **if** $\bar{D}$ satisfies $\bar{S}$ **then**
18: $\bar{\mathcal{S}}$.push($\bar{S}$)
19: **end while**

After converting the document, the algorithm moves on with a traversal of the BoXFilter tree by keeping two stacks. The first stack $\mathcal{S}$ keeps pointers to the leaf nodes that contain Prüfer encoding of a profile and is a subsequence of the document encoding $D$. The second stack $\mathcal{N}$ keeps a list of internal nodes that have a sequence envelope that is a subsequence of the document encoding $D$. In every iteration an element $C$ is popped from $\mathcal{N}$ and its children are examined (lines 3-13). If the examined element is a leaf and the profile sequence in this node is a subsequence of the document encoding $D$, then this element is placed in the stack with candidate profiles $\mathcal{S}$. If the examined element is an intermediate node and the sequence envelope for this node is a subsequence of the document encoding $D$, then this element is placed in the stack with subtree roots that need to be further examined $\mathcal{N}$.

After the traversal of the BoXFilter tree, there is a verification step (lines 14-19) for examining the content of $\mathcal{S}$. For each candidate profile, the raw profile data $\bar{S}$ is loaded from the routing table (using the function *LookUp*()) and it is compared with the document $\bar{D}$ in XML format. If the profile satisfies this verification, it is placed in the result set.

For the example in Figure 8, we start by comparing the tree root in document $D$ to the query root sequence envelope. Since there is a match we examine the root's three children nodes. We have a subsequence matching only with the first child (node 2) so we ignore the subtrees rooted at the second and the third children nodes. Then, we examine the children of node 2, which are now leaf nodes. We do subsequence matching between the document and each of the strings in these leaf nodes. In this example, there is a match between the document and leaf number 1.

**Table 2: Symbols from mapped XML tags**

| Symbol | XML tag substituted with that symbol |
|---|---|
| A | $Author$ |
| B | $Chapter$ |
| C | $Corrector$ |
| D | $Book$ |
| E | $Publisher$ |
| F | $Distributor$ |

**Algorithm 4** Document Filtering (batch mode)

**Require:** R: query BoXFilter root, M: document BoXFilter root, T: routing table, H document table
**Ensure:** The set of profiles $\bar{S}$ which are satisfied
1: Set $\mathcal{S} \leftarrow \emptyset; \mathcal{N} \leftarrow (R, M)$,
2: **while** $\mathcal{N}$ is not empty **do** ▷ Tree Join step
3: ($C_1$,$C_2$) = $\mathcal{N}$.pop()
4: **for** each child $E_1$ of node $C_1$ **do**
5: **for** each child $E_2$ of node $C_2$ **do**
6: **if** both $E_1$ and $E_2$ are leafs **then**
7: **if** $E_1$'s sequence is a substring of $E_2$ **then**
8: $\mathcal{S}$.push($E_1$,$E_2$)
9: **else**
10: **if** $E_1$ is leaf **then**
11: LSearch($E_1$,$E_2$,$\mathcal{S}$)
12: **else**
13: **if** $E_2$ is leaf **then**
14: RSearch($E_1$,$E_2$,$\mathcal{S}$)
15: **else**
16: **if** $E_1$'s sequence envelope is a substring of $E_2$ sequence envelope **then**
17: $\mathcal{N}$.push($E_1$,$E_2$)
18: **end for**
19: **end for**
20: **end while**
21: **while** $\mathcal{S}$ is not empty **do** ▷ Verification step
22: (S,D) = $\mathcal{S}$.pop()
23: $\bar{S}$ = T.LookUp(S)
24: $\bar{S}$ = H.LookUp(D)
25: **if** $\bar{D}$ satisfies $\bar{S}$ **then**
26: $\bar{\mathcal{S}}$.push($\bar{S}$)
27: **end while**

**Batch Processing**. A second alternative is to have documents being processed in batches. In this mode, documents are organized in a BoXFilter tree in the same way the user profiles are organized. That is, incoming documents are parsed, the original XML tags are replaced with symbols, and each document is converted into Prüfer sequence encoding. The original XML representation of a document is stored in a hash table called *document table*. The document encodings are inserted in a BoXFilter tree structure referred as the *document tree*.

The matching process is performed by joining the BoXFilter tree containing profile envelopes and the document tree, as illustrated by Algorithm 4. The join traverses simultaneously both trees in a top down manner by pairing (at each level) nodes from the document tree $E_2$ and the query tree $E_1$, i.e. it checks whether the sequence envelope of the query node is a substring of the sequence envelope of the document tree (lines 3-20). This parallel traversing continues until the leaf level in one of the trees is reached (it is possible that the document and the query tree have different heights).

After this phase, the join becomes a search operation on the remaining subtree using the key in the leaf node, which is performed by the functions *LSearch* and *RSearch* (lines 11 and 14). The joined pairs of documents and profiles still need to be verified if they actually match using the raw profile and document data from the routing table and the document table (lines 21 - 27).

# 5. EXPERIMENTAL EVALUATION

We have built a simulator of a pub-sub system in order to empirically study the viability of our approaches. We then performed a series of experiments to assess the behaviors of the new algorithms BUFF and BoXFilter. Previous work on XML-enabled pub-sub systems can be broadly categorized into top-down methods (perform an in-order evaluation of the document) and bottom-up meth-

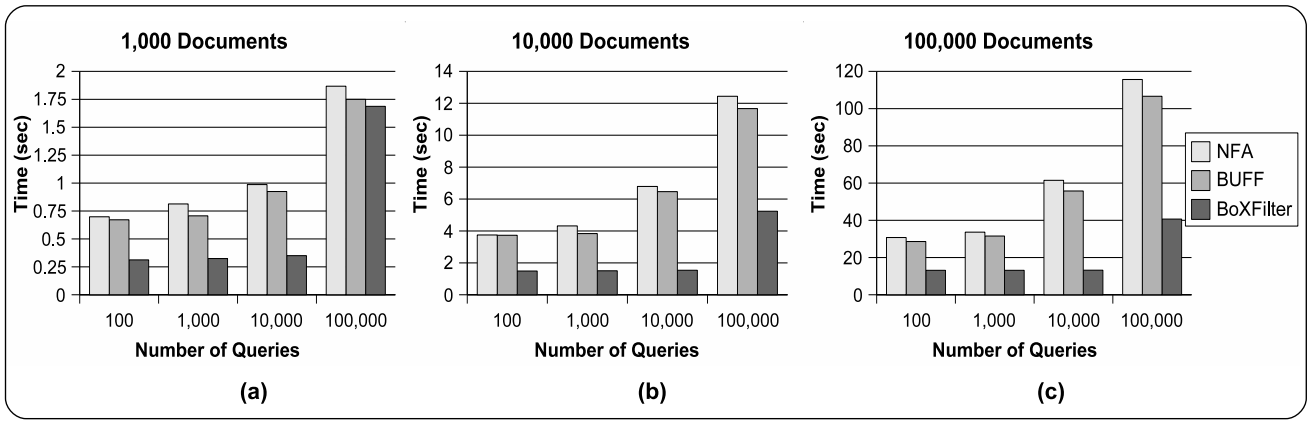



**Figure 10: Performance results when varying the number of queries.**

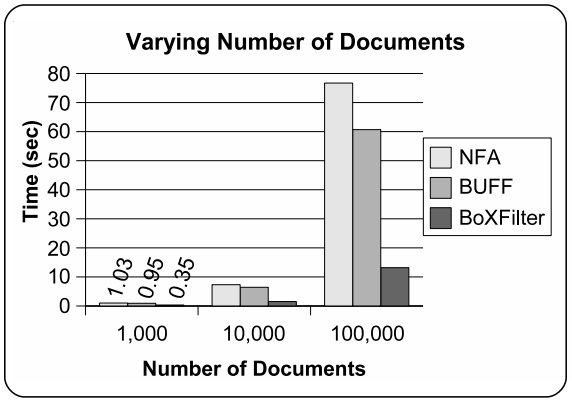



**Figure 9: Performance results when varying the number of documents.**

ods (perform a post-order evaluation of the document). A traditional top-down NFA (referred as *NFA*) is used for representing all previous top-down FSM-based approaches. As a representative of the bottom-up approaches we used the proposed BUFF method (as mentioned earlier, BUFF avoids FiST's translating of documents and queries to Prüfer sequences). The main goal of this experimental section is to show clearly the advantages of our early pruning methods (BoXFilter and B-BoXFilter) against methods that do not perform early pruning.

The performance measure is the time (in seconds) from when a set of documents enters the system, until when these documents are filtered by the matching process (note that we do not include the parsing time because that is the same for all approaches). We conclude this section by presenting a comparison between the regular BoXFilter approach, which processes the documents sequentially, and the batching approach (referred as *B-BoXFilter*).

## 5.1 Experimental Setup

We have generated datasets with 1000, 10000 and 100000 small documents (with up to 8KB). Specifically, we collected 20 different DTDs with variable structures and used them to generate the input messages, with the aid of the ToXGene XML document generator [2]. The queries were then specified for those datasets considering paths with 3 to 10 elements.

All algorithms were implemented in Java using Sun JDK version 1.4.0. Finally, the experiments were conducted on an Intel Pentium IV, 2.6GHz machine, with 1GB of memory.

## 5.2 Experimental Results

The experiments are divided into 3 groups that stress each feature of the pub-sub systems individually. The first group evaluates the scalability of the approaches regarding the number of documents processed. The second group evaluates the scalability regarding the number of queries processed. Then, the third group evaluates the ability of the approaches for pruning out the queries, by varying the selectivity of the group of queries. Each of the experiments evaluates the performance of a generic top-down state machine (*NFA*), our bottom-up NFA machine (*BUFF*), and our bound-based approach (*BoXFilter*).

**Varying the Number of Documents**. We first vary the number of documents processed while keeping the number of queries fixed to around 10 thousand, and the document selectivity to 50% (i.e. half of the documents satisfy any of the queries). Also, the number of queries matched to the documents is around 25% (i.e. around 2500 of those 10 thousand queries have a match with one of the documents). Figure 9 illustrates the results.

Note that for few documents, the performance of the approaches is similar. As the number of documents increases, so does the difference of the methods performances. Specifically, for evaluating 100,000 documents, BUFF is better than NFA by 20%. Nonetheless, BoXFilter performance is better than BUFF by a difference of 80%. Such enormous difference is justified by evaluating a relatively small number of queries, compared to the number of documents. The next group experiments will make that point clear.

**Varying the Number of Queries**. Now, we vary the number of queries processed, while keeping the selectivity fixed to 50% (i.e. half of the documents satisfy any of the queries). Furthermore, for each set of queries, around 25% have a match with any of the documents. We also vary the number of documents evaluated. Specifically, Figures 10a to c illustrate the results when evaluating 1000, 10000, and 100000 documents, respectively.

This set of experiments illustrates that the performance of both NFA and BUFF are linear to the number of documents and queries evaluated. On the other, BoXFilter has a constant performance for a relatively small number of queries. Then, its performance starts to suffer when evaluating 100 thousand queries, even though it is still much better than the FSM-based approaches.

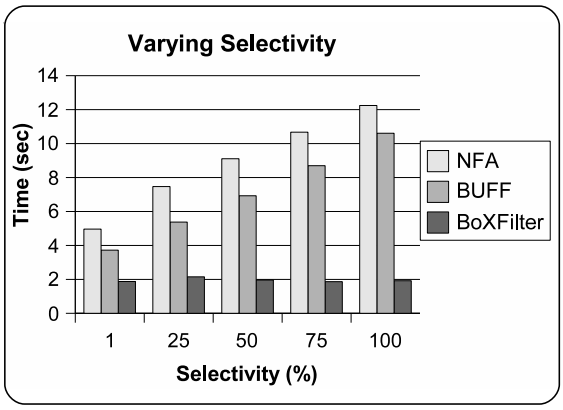



**Figure 11: Performance results when varying the document selectivity.**

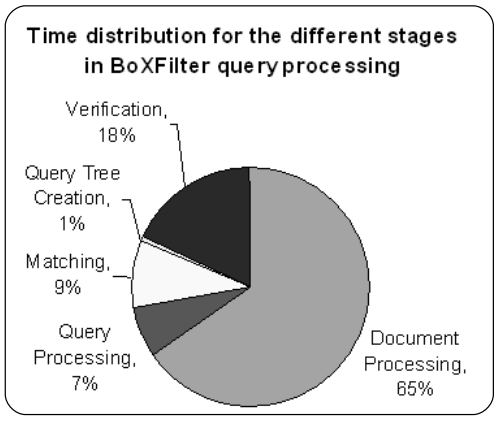



**Figure 12: Time spent for the different stages in the BoXFilter.**

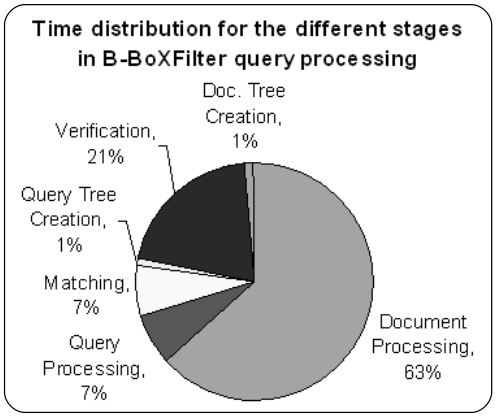



**Figure 13: Time spent for the different stages in the B-BoXFilter.**

**Varying the Selectivity**. In this set of experiments, we vary the number of documents that match any of the profile queries. In other words, we vary the selectivity of the set of queries, such that selectivity 1% means that one percent of the documents satisfy *any* of the queries. Furthermore, we keep the number of queries and the number of documents fixed to 10000. Figure 11 shows the results.

This set of experiments complements the first two groups. Note that the performance of both NFA and BUFF are linear to the selectivity of the documents. This is justified by the way the state machines work. Even though the order in which they evaluate the queries and documents differ, the central process (i.e. state transitions) is the same. Finally, these results also show that the performance of BoXFilter does not depend on how many documents have a match.

## 5.3 Batching BoXFilter

Figures 12 and 13 provide a description of the processing time spent for the different stages in the BoXFilter (sequential process) and B-BoXFilter (batch process) query processing algorithms, respectively. The prevailing cost in the algorithms (around 65%) is the cost of the document processing, i.e., the encoding using Prüfer sequences. This cost (expressed as a percentage) is similar for both the BoXFilter and the B-BoXFilter and it is based on a fixed algorithm (i.e. we cannot optimize it further).

The second major share is the time needed for verifying the results generated by the Prüfer subsequence matching process. Like the cost for document processing, the verification cost is also unavoidable. However, unlike the cost of document encoding, the verification process is not a function of the input size. Rather, it depends more on the query selectivity and the similarity between the documents and profiles sequence encodings. Nevertheless, the query selectivity is given by the application and is thus out of our control.

Next, the cost for query processing (i.e., the encoding of profiles to Prüfer sequences) is similar for both the BoXFilter and B-BoXFilter since (like the document encoding) is based on a fixed algorithm. Similarly, both approaches have the same query tree creation cost (which is actually very small, around 1%). As a result, the only cost that can be further optimized is the Matching (query/document) cost.

Note that in the matching phase, the B-BoXFilter has an extra step - to create the document envelope tree. For small batches of documents, this document tree construction time is minimal (around 1%) of the total time. However, with the increase of the batch size, this cost increases as well and thus the performance of the algorithm is affected. On the other hand, the matching time in the B-BoXFilter is smaller than the matching time in the BoXFilter because the documents have been already clustered according to their similarity in the document tree. During the join phase which finds the matches between the document envelope tree and the query envelope tree, only nodes from these trees that contain similar subsequences are paired. Since the documents are clustered, we can traverse the query tree with an envelope of documents (instead of a single document) and hence the B-BoXFilter reduces the cost for document/query matching. As a result, the B-BoxFilter is advantageous when the time spent for the extra step (the document envelope-tree creation) is less than the benefit in the matching time. We thus expect the B-BoxFilter method to provide improvement for relatively small batches of documents (when compared to the number of queries).

Figure 14 shows a comparison between the BoXFilter and the B-BoXFilter for different number of queries and for fixed number of documents (1000 documents). The selectivity of the queries is set to 25%. The figure confirms the performance benefit of the B-BoxFilter: (while the document batch size remains the same and relatively small) as the query size increases, B-BoxFilter provides better overall performance because of the improvement in the matching phase due to the document clustering.

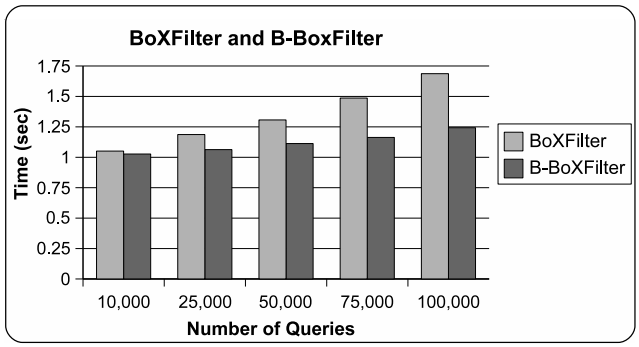



**Figure 14: BoXFilter vs. B-BoXFilter.**

# 6. RELATED WORK

A considerable number of proposals for designing pub-sub systems have already appeared. Early systems were topic-based, where the topic of the message would define its destination [24, 35]. Then, the messages evolved to conjunctions of {attribute, value} pairs and the profiles to predicates over those pairs [5, 10, 11]. Comparing to the focus of our work (*XML-aware* pub-sub systems) those systems are much simpler and limited, since XML allows more expressive messages and profiles.

Among the earlier work on XML filtering, *XFilter* is probably the most relevant [1]. It defines a finite state machine for each query, then it proposes an index over those machines. The machines are then executed concurrently for each document. When an accepted state is reached, the document is reported as a match to the corresponding profile. Later on, *YFilter* improved the matching performance by proposing one unique FSM for all profiles, which allowed common query paths to be processed only once [9]. Finally, other FSM-based approaches use different techniques for building the machine as well as different types of machines. For example, [15] lazily builds single deterministic pushdown automaton, [13] employs a lazily built Deterministic Finite Automaton (DFA), [22] builds a transducer (which employs a DFA with a set of buffers), and [28] employs a hierarchical organization of pushdown transducers with buffers.

Note that neither *XFilter* nor *YFilter* employs any holistic information during the matching process. To overcome this problem, *FiST* employs a different, bottom-up approach based on sequencing the twig patterns using Prüfer sequences [20]. Our approaches differ in two aspects. First, BUFF employs a FSM to perform the matching. It still is bottom-up, but it avoids translating documents and queries to Prüfer sequences. Second, BoXFilter does translate documents and queries to Prüfer sequences but it employs a novel pruning technique based on lower and upper bound estimates, such that it reduces the query space considerably.

Even though those methods differ from each other in the type of FSM employed and the query semantics they support, all of them suffer from a common weakness. In order to determine if a document satisfies a profile, they try to match the document with all profiles, which can be inefficient considering the high number of profiles in a typical pub-sub system.

A different approach utilizes the standard relational model to implement the matching process of pub-sub systems. For example, the work in [34] translates the profiles and the messages to the Relational model. The matching then can be expressed as a join operation between the sets of messages and profiles. The main problem with this method is that the number of joins necessary is proportional to the size of the query, then hampering the matching performance for long queries.

Yet another methodology is to aggregate the profiles using some indexing technique [6]. The matching process then reads the input message and traverses the index in order to select the queries satisfied. Moreover, the index takes advantage only of common parent/child relationships. Nonetheless, our pruning approach is able to identify earlier on the profiles that will not provide any matches, then incurring less index probes.

Additionally, pub-sub applications provide other challenges besides efficient matching process (which is the focus of our work). Recently, [36, 37] proposed *RoXSum*, a data structure that is able to optimize not only the matching process, but also the routing of XML documents within pub-sub systems. *RoXSum* aggregates the incoming documents and processes all profiles in the aggregated structure (i.e. the profile matching uses batching processing). Moreover, it routes the documents using the aggregated content as well. In this paper, we propose an extension of the *BoXFilter* that processes many documents at once. However, the difference is that the routing is performed for each document individually, rather than the aggregated content. The integration of the batched-matching process with batched-routing is left as future work.

Other challenges of pub-sub systems design include the construction of the overlay network structure [12, 26, 33], the distribution of the profiles [8], and message routing policies [27, 32]. However, the matching process is *complementary* to those issues. For example, we can consider that the network is built using an approach similar to [12] and that the profiles are distributed according to [8].

Finally, the notion of sequence envelopes is reminiscent of the atomic wedgies presented in [38]. However, wedgies are used for bounding time series for the purpose of finding similar series. While in this work, envelopes are used for representing XML queries and finding exact matches.

# 7. CONCLUSION

In this paper we considered the problem of XML filtering for publish-subscribe systems. We first proposed a FSM-based approach (BUFF) that evaluates the documents in a bottom-up order. We then introduced the idea of early profile pruning and proposed a sequence-based index (BoXFilter) which allows to prune out queries very efficiently. First, documents and queries are transformed into sequences and grouped into envelopes. Then, the queries can be pruned out by evaluating the lower and upper bounds of their envelopes. This is the first time that a concept of envelopes in employed for XML query processing. Finally, our experimental evaluation shows that, even though BUFF offers performance advantages over the traditional FSM-based approach, the pruning offered by BoXFilter provides drastic performance improvement.

# 8. ACKNOWLEDGMENTS

We would like to thank Eamonn Keogh, Li Wei and Stefano Lonardi for the many discussions on the subsequence matching and the sequence index construction.

# 9. REFERENCES

[1] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of VLDB*, 2000.

[2] D. Barbosa et.al. Toxgene: A Template-based Data Generator for XML. In *Proc. of WebDB*, June 2002.

[3] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML Path Language (XPath) 2.0. In *W3C Proposed Recommendation, http://www.w3.org/TR/xpath20*, November 2006.

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. In *W3C Proposed Recommendation, http://www.w3.org/TR/xquery*, November 2006.

[5] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A Routing Scheme for Content-Based Networking. In *Proc. of INFOCOM*, 2004.

[6] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of ICDE*, pages 235–244, 2002.

[7] C. Y. Chan and Y. Ni. Efficient XML Data Dissemination with Piggybacking. In *Proc. of SIGMOD Conference*, 2007.

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

[9] Y. Diao et. al. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM TODS*, 28(4), Dec 2003.

[10] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[11] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In *Proc. of SIGMOD*, 2001.

[12] W. Fenner et. al. XTreeNet: Scalable Overlay Networks for XML Content Dissemination and Querying. In *Proc. of WCW*, 2005.

[13] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Trans. Database Syst.*, 29(4):752–788, December 2004.

[14] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *Proc. of ICDT*, 2003.

[15] A. K. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *Proc. of SIGMOD Conference*, pages 419–430, 2003.

[16] A. Guttman. R-trees: a Dynamic Index Structure for Spatial Searching. In *Proc. of SIGMOD Conference*, pages 47–57, 1984.

[17] B. He, Q. Luo, and B. Choi. Cache-Conscious Automata for XML Filtering. In *Proc. of ICDE*, 2005.

[18] M. Hong et al. Massively multi-query join processing in publish/subscribe systems. In *Proc. of SIGMOD Conference*, 2007.

[19] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proc. of VLDB*, pages 273–284, 2003.

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

[21] G. M. Lohman et.al. Query Processing in R*. *Query Processing in Database Systems*, pages 31–47, 1985.

[22] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proc. of VLDB*, pages 227–238, 2002.

[23] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proc. of ICDT*, pages 277–295, 1999.

[24] T. Milo, T. Zur, and E. Verbin. Boosting Topic-Based Publish-Subscribe Systems with Dynamic Clustering. In *Proc. of SIGMOD Conference*, 2007.

[25] M. M. Moro, Z. Vagena, and V. Tsotras. Tree-Pattern Queries on a Lightweight XML Processor. In *Proc. of VLDB*, 2005.

[26] O. Papaemmanouil, Y. Ahmad, U. Çetintemel, and J. Jannotti. Application-aware Overlay Networks for Data Dissemination. In *Proc. of ICDE Workshops*, page 76, 2006.

[27] O. Papaemmanouil and U. Centintemel. SemCast: Semantic Multicast for Content-based Data Dissemination. In *Proc. of ICDE*, 2005.

[28] F. Peng and S. S. Chawathe. XPath Queries on Streaming Data. In *Proc. of SIGMOD Conference*, pages 431–442, 2003.

[29] H. Prüfer. Neuer Beweis eines Satzes ber Permutationen. *Archives of Mathematical Physics, 27*, pages 742–744, 1918.

[30] P. Rao and B. Moon. Sequencing XML data and query twigs for fast pattern matching. *ACM Trans. Database Syst.*, 31(1):299–345, 2006.

[31] SAX. Simple API for XML. In *http://sax.sourceforge.net*.

[32] R. Shah, Z. Ramzan, R. Jain, R. Dendukuri, and F. Anjum. Efficient Dissemination of Personalized Information Using Content-Based Multicast. *IEEE Transactions on Mobile Computing*, 3(4):394–408, October 2004.

[33] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-Based Content Routing using XML. In *Proc. of SOSP*, pages 160–173, 2001.

[34] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymaki. Implementing a Scalable XML Publish/Subscribe System Using Relational Database Systems. In *Proc. of SIGMOD*, 2004.

[35] TIBCO. *TIB/Rendezvous*. White paper. TIBCO, Palo Alto, CA, 1999.

[36] Z. Vagena, M. M. Moro, and V. J. Tsotras. RoXSum: Leveraging Data Aggregation and Batch Processing for XML Routing. In *Proc. of ICDE*, 2007.

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

[38] L. Wei, E. J. Keogh, H. V. Herle, and A. Mafra-Neto. Atomic Wedgie: Efficient Query Filtering for Streaming Times Series. In *Proc. of ICDM*, pages 490–497, 2005.