# Schema-Conscious Filtering of XML Documents

Panu Silvasti
Helsinki University of Technology
psilvast@cs.hut.fi

Seppo Sippu
University of Helsinki
sippu@cs.helsinki.fi

Eljas Soisalon-Soininen
Helsinki University of Technology
ess@cs.hut.fi

## ABSTRACT

In a publish-subscribe system based on filtering of XML documents, subscribers specify their interests with profiles expressed in the XPath language. The system processes a stream of XML documents and delivers to subscribers a notification or content of documents that match the profiles. For filtering with profiles expressed as linear XPath queries, automaton-based approaches exist where the intractable size growth of a preconstructed deterministic finite automaton is avoided by using a nondeterministic automaton. In this article we examine how these general approaches, which do not assume the existence of any specific schema or document type definition (DTD), might benefit from the knowledge that all the XML documents to be filtered obey a given DTD.

We present an algorithm that utilizes the DTD in the preprocessing phase of the filtering automaton to prune out descendant operators (//) and wildcards (∗) from the linear XPath filters. Experiments with data obtained from the XML Data Repository of the Univ. of Washington indicate that filter pruning can increase the throughput of the nondeterministic YFilter automaton by Diao et al. by a factor of 2 to 20. We also present a new filtering algorithm that is based on a backtracking deterministic finite automaton derived from the classic Aho–Corasick pattern-matching automaton. This automaton has a size linear in the sum of the sizes of the filters. For our algorithm, we obtained a throughput of 15 MB/sec for filters pruned from one million original filters (with all wildcards and non-leading descendant operators eliminated), representing an improvement by a factor of 2 to 3 upon the throughput of YFilter.

## 1. INTRODUCTION

A publish-subscribe system consists of one or more publishers and many subscribers, where the publishers provide a stream of documents and the subscribers specify their interests with filters that match some of those documents [2]. Publish-subscribe systems have emerged in everyday use; examples include *Google alerts* and stock-information delivery by *Yahoo.com*.

In a publish-subscribe system based on XML filtering, the profiles are usually specified by filters written in the XPath language. The system processes the stream of XML documents and delivers to subscribers a notification or the content of those documents that match the filters. The number of interested subscribers and their stored profiles can be very large, thousands or even millions. In this case the scalability of the system is critical.

The primary problem we address in this paper is defined as the *filtering problem for XML streams*: given a set of XPath expressions and a stream of XML documents, for each document in the stream identify those expressions that match the document. More specifically, we study the filtering problem for linear XPath expressions, that is, XPath expressions that do not have branches in their query trees. Linear XPath expressions without predicates are described by the following grammar:

$$path \rightarrow / \ step \mid // \ step \mid path \ path$$
$$step \rightarrow label \mid *$$

where *label* denotes an XML-element label.

Several approaches to XML filtering with XPath filters use an automaton as a basis of the filtering algorithm [2, 4, 6, 7, 9]. Diao et al. [4] report an evaluation method called YFilter that applies nondeterministic finite automata (NFAs). YFilter is an improvement upon its predecessor, called XFilter [2], which uses a separate NFA for each filter but executes them simultaneously in processing the input document. YFilter uses a single NFA that combines the effect of the individual NFAs and achieves considerable improvements in performance by path-sharing, that is, by merging states that correspond to common prefixes in different query paths, while still retaining the linear size of the NFA with respect to the filter descriptions.

The algorithm of Green et al. [6] is based on a single deterministic finite automaton (DFA). The state explosion of the DFA is tackled by constructing the DFA lazily. In other words, the DFA is constructed runtime, on demand: if in processing the stream of XML documents, no next state is defined on the current input symbol, the corresponding new state will be computed and the process is continued at this new state. While exponential in the worst case, this approach works extremely well in many cases, when the incoming XML documents obey a schema or document type definition (DTD) that is nonrecursive or contains only simple cycles (a cycle is simple if its nodes do not occur in other cycles).


*EDBT 2009*, March 24–26, 2009, Saint Petersburg, Russia.

The filtering methods cited above are general in that the input documents to be filtered are not required to comply with any predefined schema or DTD; they are only expected to obey the generic syntax of XML. However, in practice the documents published by a specific site may very well be restricted to a few number of different topics and described by a specific XML schema. This raises the question whether filtering speed or throughput could be improved by utilizing knowledge about a DTD in building the filtering automaton.

Fernández and Suciu [5] have presented a technique called "query pruning" for optimizing regular path expressions with graph schemas. Inspired by their work, we have developed a new optimization method, called *filter pruning*, that takes as input a DTD and a set of linear XPath filters and produces a set of "pruned" linear XPath filters that contain as few descendant operators "//" and wildcards "∗" as possible. The set of pruned filters is equivalent to the set of original filters in that each original filter is represented by the union of a set of pruned filters that match the same set of XML documents, provided that the documents obey the DTD. Imposing some simple conditions stating when an operator may be eliminated we can guarantee a polynomial bound on the total size of the pruned filters.

Our experiments with two XML data sets obtained from the XML Data Repository of the Univ. of Washington [12] indicate that filter pruning can increase the throughput of the nondeterministic YFilter automaton by Diao et al. [4] by a factor of 1.3 to 22.8. The DTD of one of these data sets is nonrecursive, allowing exhaustive elimination of all "∗" and "//" operators, while the other DTD is slightly recursive, so that 14–33 % of the "//" could not be eliminated. The filter workloads were generated by the XPath query generator described by Diao et al. [4].

We have also designed a filtering algorithm that seems to be especially amenable to filter pruning. The basis of our algorithm is the classic Aho–Corasick [1] pattern-matching automaton (PMA), turned from a language recognizer to a filtering automaton with a backtracking facility. The filter-pruning optimization is first applied to eliminate all wildcards "∗" and as many descendant operators "//" as possible from the linear XPath filters. Then the PMA is constructed for the set of keywords formed from the XML element strings separated by the "//" operators that possibly remain in the pruned filters. If all non-leading "//" operators can be eliminated from the filters, our basic algorithm is able to report exact matches for all filters. To handle any remaining non-leading "//" operators we have developed an extended algorithm that can be used to match sequences of keywords.

Our filtering method shares with NFA-based methods such as YFilter [4] the guarantee of a polynomial worst-case size bound of the filtering automaton with respect to the original filters, while gaining from the determinism offered by a preconstructed DFA-like PMA. For our algorithm, we obtained a throughput of 15 MB/sec for filters pruned from one million original filters (with all wildcards and non-leading descendant operators eliminated), representing an improvement by a factor of 2 to 3 upon the throughput of YFilter.

The idea of using the Aho–Corasick PMA for XML document filtering is presented by Soisalon-Soininen and Ylönen [11]. Preliminary results of our pruning and filtering algorithms were presented at the VLDB 2008 Ph.D. Workshop [10].

Our paper is organized as follows. In Sec. 2 we present our pruning algorithm, and in Sec. 3 we show how the Aho–Corasick PMA can be turned to a filtering automaton for tree-structured text such as XML. In Sec. 4 we present our extended algorithm for filtering with sequences of keywords, to be used when pruning cannot eliminate all non-leading descendant operators from the filters. In Sec. 5 we report results from experiments with our algorithms, such as memory consumption and filtering speed, as well as a limited benchmarking with YFilter. In Sec. 6 we compare our method with relevant previous work.

```
<!ELEMENT a (b|c|d)>
<!ELEMENT b (f*)>
<!ELEMENT c (f*)>
<!ELEMENT d (e*)>
<!ELEMENT e (f*)>
<!ELEMENT f (i|j)*>
<!ELEMENT i (k*)>
<!ELEMENT j (k*)>
```

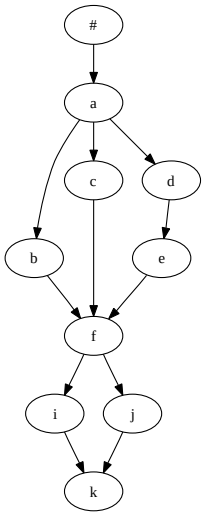



**Figure 1: A nonrecursive DTD and its graph schema.**

## 2. FILTER PRUNING

Assume an NFA-based filtering algorithm such as YFilter [4] where a single NFA is constructed for a set of linear XPath filters. To such an NFA, occurrences of wildcards "∗" in the filters add transitions on a large set of symbols, and occurrences of the descendant operator "//" add loops on a large set of symbols. These transitions and loops cause a lot of nondeterminism, and if the NFA is to be transformed into a DFA, a state explosion may result, although only a small subset of the states and transitions might be visited in running the automaton on actual input documents.

Assuming that the XML documents to be filtered all conform to a given document type definition (DTD), we can expect improvements in filtering speed if we manage to restrict the transitions of the filtering automaton to those that may actually be traversed when filtering an input document that conforms to the DTD. Green et al. [6] showed that their approach of constructing the DFA lazily results in a small and fast filtering automaton when the input documents conform to a DTD that is nonrecursive or contains only simple cycles.

In this section we show how to utilize the DTD to eliminate or "prune out" as many of the wildcards and (non-leading) descendant operators as possible from the subscriber-provided linear XPath filters. We call the process *filter pruning*, in analog with "query pruning" presented by Fernández and Suciu [5]. Our ultimate goal is a set of pruned filters in which all wildcards and all non-leading descendant operators have been eliminated, but this may not always be possible

because of recursion in the DTD or because the pruning may result in too many pruned filters or in too large pruned filters.

Given a DTD or schema, let $G$ be its *graph schema* [3], that is, the directed graph whose set of nodes is the set of XML elements in the DTD and that contains a directed edge from node $a$ to node $b$ if and only if $b$ is a child element of $a$. There is a distinguished node, labeled with # and representing the root element of an XML document, that has no incoming edges. Fig. 1 shows a sample DTD and its graph schema. This DTD is nonrecursive, and hence its graph schema is acyclic.

We say that a linear XPath filter $V$ is *consistent* with the DTD, if it represents at least one path in the graph schema $G$ of the DTD; in other words, we can find XML elements substituting for the occurrences of the wildcard "$*$" in $V$, and strings of XML elements separated by child operators "/" substituting for the occurrences of the descendant operator "//" in $V$, such that the resulting string (of XML elements and child operators "/"), when stripped of all child operators, is a path in $G$ from the root element down to some element.

As with YFilter [4], we assume that each subscriber-defined XPath filter is first rewritten into an equivalent form in which (1) each occurrence of "//$*$" is turned into "/$*$//", (2) each maximal substring of multiple consecutive substrings "/$*$//" is turned into a single "/$*$//", and (3) "//" is removed from "/$*$//" occurring at the end of the filter.

**Algorithm 1** Checking a linear XPath filter $V$ for consistency with a DTD.

```
S ← V
Accessible ← {#}
while Accessible is nonempty and S is nonempty do
  if S is of the form /bS' where b is an element then
    if b ∈ children(Accessible) then
      Accessible ← {b}
    else
      Accessible ← empty
    end if
  else if S is of the form /*S' then
    Accessible ← children(Accessible)
  else if S is of the form //bS' where b is an element
  then
    if b ∈ descendant(Accessible) then
      Accessible ← {b}
    else
      Accessible ← empty
    end if
  end if
  S ← S'
end while
return(Accessible is nonempty and S is empty)
```

Algorithm 1 checks a filter $V$ for consistency with a DTD, using the graph schema of the DTD. For element set $E$, the function *children*$(E)$ returns the the set of all children of all elements in $E$, and the function *descendant*$(E)$ returns the set of all descendants of all elements in $E$. The algorithm processes filter $V$ from left to right, extracting step by step a prefix of $V$, and maintaining a set *Accessible* that contains the set of elements accessible in the graph schema from the root element upon reading the so-far-extracted prefix of $V$. Filter $V$ is consistent with the DTD if and only if the algorithm reaches the end of $V$ with the set *Accessible* nonempty. The variable $S$ stores the non-yet-processed suffix of $V$.

| Original filter | Pruned filter |
|---|---|
| $/a//f$ | $/a/b/f \cup /a/c/f \cup /a/d/e/f$ |
| $//c/f//k$ | $//c/f/i/k \cup //c/f/j/k$ |
| $/*/b$ | $/a/b$ |
| $/a/*$ | $/a/b \cup /a/c \cup /a/d$ |
| $/a/*/f$ | $/a/b/f \cup /a/c/f$ |
| $/*/*/*/*$ | $/a/b/f/i \cup /a/b/f/j \cup /a/c/f/i \cup /a/c/f/j \cup /a/d/e/f$ |

**Table 1: Original XPath filters and corresponding pruned filters obtained by pruning with the DTD of Fig. 1. All wildcards "$*$" and all non-leading descendant operators "//" were eliminated.**

Pruning a filter $V$ with respect to a DTD is just finding all combinations of substituting elements for as many occurrences of "$*$" as possible, and all substituting element strings for as many (non-leading) occurrences of "//" as possible, such that the pruned filters $V_1, \ldots, V_n$ obtained by those substitutions are consistent with the DTD and that their union, denoted by $V_1 \cup \ldots \cup V_n$, is *equivalent* to the original filter $V$, that is, any XML document that conforms to the DTD matches with $V$ if and only if it matches with one of the filters $V_i$, $i = 1, \ldots, n$.

Tab. 1 shows a set of filters and the result of pruning them with respect to the nonrecursive DTD of Fig. 1 when all the "$*$" operators and all non-leading "//" operators are eliminated. For example, in the case of the original filter $/a//f$, we find that the graph schema of the DTD contains three paths from element $a$, the child of root element #, to element $f$, namely *abf*, *acf*, and *adef*. Thus the element strings to be substituted for the occurrence of "//" are $/b$, $/c$, and $/d/e$, resulting in the pruned filter $/a/b/f \cup /a/c/f \cup /a/d/e/f$.

The ultimate goal of exhaustive elimination of "//" operators naturally cannot be achieved when the DTD is recursive, that is, when the graph schema is cyclic, but exhaustive elimination of "//" or "$*$" may also be infeasible with nonrecursive DTDs. A simple example of a case in which exhaustive elimination results in an exponential increase in the size of filters with respect to the combined size of the DTD and the filters is the DTD having elements $a_1, \ldots, a_k, a_{k+1}$, $b_1, \ldots, b_k$, $c_1, \ldots, c_k$, where $b_i$ and $c_i$ are children of $a_i$, and $a_{i+1}$ is a child of both $b_i$ and $c_i$, $i = 1, \ldots, k$. Eliminating "//" from the filter $/a_1//a_{k+1}$ results in a union of $2^k$ pruned filters of size $\Theta(k)$, although the original filter is of size $O(1)$ and the DTD is of size $O(k)$. The same union of pruned filters is also the result of exhaustive elimination of "$*$" from the filter $/a_1/*/a_2/*/\ldots/*/a_{k+1}$, which is of size $O(k)$. Obviously, we must control the number and size of pruned filters to be created.

For pruning a set of filters with respect to a DTD, we precompute a two-dimensional array, *substitutes*, indexed by pairs of elements in the DTD. For an element pair $(a, b)$, the entry *substitutes*$[a, b]$ will contain the set of all strings $/c_1/c_2/\ldots/c_n$ such that $ac_1c_2\ldots c_nb$ is a path in the graph schema of the DTD, if the number of such strings is finite and the sum of their lengths falls below a preset limit; otherwise, the entry will be set empty.

The contents of the array *substitutes* only depend on the

DTD and on the preset limits used to keep the size of the array reasonable. Setting a constant limit on the size of all entries of this array guarantees a worst-case size bound on the array that is quadratic in the size of the DTD.

Different heuristics can be used to regulate cases when "$*$" and "//" may be eliminated and cases when they may not. Algorithms 2 and 3 represent a recursive formulation of a pruning algorithm in which all "$*$" operators are eliminated exhaustively, while the elimination of "//" operators is controlled by the precomputed array *substitutes*. The algorithm can be used to prune with recursive DTDs, but substrings $a//b$ are, naturally, left unpruned if the DTD contains cycles on paths from element $a$ to element $b$, or if there are simply too many paths from $a$ to $b$ in the DTD so that the entry $substitutes[a, b]$ has been set empty.

A recursive call $prune(S, a, P)$ takes as arguments a suffix $S$ of the filter $V$ to be pruned, the corresponding pruned prefix $P$ of $V$ (pruned in ancestor calls), and the last element $a$ in $P$. The call extracts a prefix, $/b$, $/*$, or $//b$, from $S$, prunes it if needed and possible, and concatenates the result to $P$, to be used as an argument to further recursive calls of the procedure. A recursion path terminates when the suffix $S$ becomes empty; then the argument $P$ represents a complete pruned filter (i.e., one disjunct in the final union of pruned filters) and is written to the output of the algorithm. The pruning of filter $V$ in the main program is started by the call $prune(V, \#, \epsilon)$, where # denotes the root element and the empty string $\epsilon$ indicates a so-far-empty pruned prefix of $V$.

**Algorithm 2** Procedure $prune(S, a, P)$

```
if S is empty then
  output P
else if S is of the form /bS' where b is an element then
  if b ∈ children(a) then
    prune(S', b, P/b)
  end if
else if S is of the form /*S' then
  for all b ∈ children(a) do
    prune(S', b, P/b)
  end for
else if S is of the form //bS' where b is an element then
  if substitutes[a, b] is nonempty then
    for all x ∈ substitutes[a, b] do
      prune(S', b, Px)
    end for
  else
    prune(S', b, P//b)
  end if
end if
```

**Algorithm 3** Pruning a linear XPath filter $V$.

```
Rewrite V
prune(V, #, ε)
```

Algorithm 2 is formulated so that it can also eliminate leading occurrences of the descendant operator "//", although for the purposes of our filtering algorithms of Secs. 3 and 4 it is sufficient to eliminate only non-leading occurrences of that operator. However, some filtering methods may also gain from eliminating leading occurrences. To prevent leading occurrences of "//" from being eliminated, it is sufficient to set entries $substitutes[\#, b]$ empty for all elements $b$.

In the worst case, Algorithm 2 is exponential in the combined size of the DTD and the original subscriber-provided filter, because the size of the output can be that large, even if the DTD is nonrecursive, as we have seen. However, imposing further restrictions on cases in which operators may be eliminated and making restrictive assumptions about the DTD or the original filters we may derive conditions under which the algorithm is guaranteed to run in polynomial time.

First, if the graph schema $G$ of the DTD happens to be a tree, then all "$*$" and "//" operators can be eliminated from any linear XPath filter $V$ in time $O(|V| + |G|)$, where $|V|$ is the size of $V$ and $|G|$ is the size of $G$. In this case there is no need to precompute the array *substitutes* (which is of size $O(|G|^2)$).

In fact many real DTDs are tree-like, composed of mostly nonrecursive elements and having only few elements with many incoming edges. These properties are exemplified by the nonrecursive DTD for the protein-sequence database, one of the DTDs from the XML Data Repository at the University of Washington [12] that we have used in our experiments (see the graph schema in Fig. 2).

A simple stringent way to guarantee that the size of the result of pruning remains polynomial in the combined size of the DTD and the original filters is to require that all entries in the array *substitutes* be singleton sets and that in Algorithm 2 a filter substring "$/*S'$" may be pruned only if the last element $a$ of the already pruned prefix of the filter is the only parent of all the child elements $b$ to be substituted for "$*$", unless those child elements $b$ are all leaf elements without outgoing edges (in which case pruning is always allowed). This will guarantee the worst-case time bound $O(|V| \cdot |G|^2)$ for filter pruning.

It is possible to adjust Algorithm 2 for a specific DTD so that it will produce a sufficiently pruned reasonable-sized filters. We may adjust the entries of the array *substitutes* so that for some specific elements a larger set of substitutes are allowed, while for others pruning will be prevented even for a smaller number of substitutes. The elimination of "$*$" operators could be controlled by allowing pruning for the children of some specific elements, while disallowing it for others.

To experiment with the pruning algorithm, we used the XPath query generator described by Diao et al. [4] to generate workloads of consistent linear XPath filters for DTDs obtained from the XML Data Repository [12]. The generation of filters was parameterized by the number and maximum nesting depth of filters and by the following parameters: $prob(//)$, the probability of "//" being the operator at a location step, and $prob(*)$, the probability of "$*$" occurring at a location step. The filter workloads may contain duplicate filters, which is most likely the case with real-world filters.

Two sets of workloads of 100 000 filters were generated and pruned, one set for the nonrecursive 66-element, 83-arc DTD of protein-sequence data (Figs. 2 and 3) and the other set for a slightly recursive 61-element, 82-arc DTD of NASA astronomical data [12]. The NASA DTD (created from the data) has one cycle that makes the DTD recursive. In pruning the protein-sequence workloads, both "$*$" and "//" operators were eliminated exhaustively, while in prun-

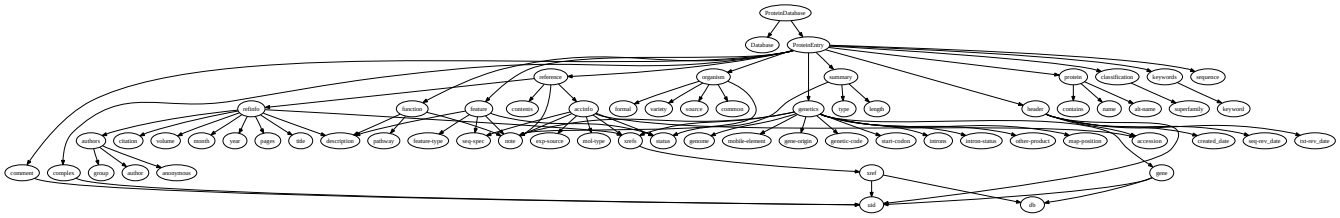



**Figure 2: The graph schema of a 66-element DTD for protein-sequence data.**

```
/ProteinDatabase/ProteinEntry/summary/status
//uid
/*/ProteinEntry/feature/note
/ProteinDatabase/ProteinEntry/function/pathway
/*/ProteinEntry/*/accinfo//seq-spec
/ProteinDatabase//keyword
/ProteinDatabase/ProteinEntry//superfamily
/*/ProteinEntry/organism//formal
/ProteinDatabase/*/*/note
/ProteinDatabase/ProteinEntry/organism/variety
//ProteinDatabase/ProteinEntry/*/*
/*/*/*
```

**Figure 3: Part of a filter workload generated for the protein-sequence DTD with $prob(//) = prob(*) = 0.2$.**

ing the NASA workload, the wildcards "∗" were eliminated exhaustively while not all descendant operators "//" could be eliminated because of the recursion. For the NASA workloads, the pruning was regulated by setting $substitutes[a, b]$ empty for element pairs $(a, b)$ involved in a recursion cycle.

We measured the number of pruned filters, that is, the total number of pruned XPath expressions in the union filters produced for the 100 000 original filters. In the case of the protein-sequence workloads, the number of pruned filters varied between 224 579 and 622 649 when $prob(*)$ varied between 0.2 and 0.6 and $prob(//)$ between 0.0 and 0.6. Tab. 2 shows the number of pruned filters and the number of remaining descendant operators for the NASA workloads. Also leading occurrences of descendant operators were eliminated. The effect of pruning in the number of filters is moderate: the number of pruned filters is not more than 2 to 6 times the number of original filters.

We have also included the size of the NFA used by YFilter [4] in each case, both for the set of unpruned filters and for the set of pruned filters. The size of the NFA was measured by examining Java's used heap space before and after the construction of the NFA. The effect of pruning on the size of the NFA seems to be insignificant.

Workloads generated with greater $prob(*)$ values tend to result in greater numbers of pruned filters, while greater $prob(//)$ values seem not to have a similar effect. The tree-like shape of the DTDs used in the experiments is the reason for the fact that the number of pruned filters stays within moderate limits even for high values of $prob(*)$. Experimental results on the effect of pruning on filtering speed are reported in Sec. 5.

## 3. SINGLE-KEYWORD-FILTER PMA

The classical Aho–Corasick pattern-matching automaton (PMA) [1] for a finite set $W$ of nonempty strings called *keywords* over a finite alphabet $\Sigma$ is a deterministic linear-time finite-state recognizer of the regular language $\Sigma^* W \Sigma^*$. The size of the PMA is $O(|W|)$, where $|W|$ denotes the sum of the lengths of all keywords in $W$. In processing input string $x$, the PMA makes at most $2|x|$ moves.

In this section we show how the Aho–Corasick PMA can be modified to filter XML documents with pruned linear XPath filters. The PMA is constructed for the set of filters obtained by pruning the subscriber-provided filters with respect to a DTD, using the algorithm described in the previous section, assuming that all wildcards "∗" are eliminated. Given an XML document that conforms to the DTD, the PMA will determine the exact set of pruned filters that match the document if all non-leading descendant operators "//" have been eliminated, and a superset of pruned filters that match the document otherwise.

To turn a PMA that recognizes linear text to a PMA that filters tree-structured text such as XML, we must do the following. First, a backtracking facility must be added in order that after scanning a substring consisting of an entire sub-element (subtree) of the input document the state at which the PMA was just before scanning the start-element tag of the sub-element can be restored, to make it possible to continue the matching process with the next sibling sub-element. Second, the output function of the PMA must be modified to report exactly which filters match the input document, instead of the simple yes/no answer telling whether or not some keyword matches the input.

In filtering XML documents, the alphabet $\Sigma$ contains the set of elements occurring in the DTD plus the additional symbol # denoting the root element of any XML document. The set $W$ of keywords is derived from the pruned filters, such that each maximal nonempty substring not containing the descendant operator "//" gives rise to one keyword, namely the substring stripped of all child operators "/", and that the first such substring is prefixed with "#" if the filter begins with the child operator "/". Thus, the pruned filter "$//a/b//c/d$" gives rise to the keywords $ab$ and $cd$, while the pruned filter "$/a/b//c/d$" gives rise to the keywords $\#ab$ and $cd$.

For each prefix $y$ of some keyword in $W$, the Aho–Corasick PMA has a unique state, denoted by $state(y)$, different from all $state(y')$ where $y' \neq y$. The state $state(\epsilon)$, where $\epsilon$ is the empty string, is the *initial state* of the PMA. Clearly, the number of states in the PMA is at most $|W| + 1$. The states are numbered with positive integers. Fig. 4 gives the transition diagram of the PMA for a set of filters.

The *goto function* of the PMA is defined by the equation $goto(state(y), a) = state(ya)$, where $ya$ is a prefix of some keyword and $a$ is a symbol in $\Sigma$. For any state $q$ we denote

| Workload parameters | | Number of | Number of "//" operators | | | | Size of YFilter's NFA | |
|---|---|---|---|---|---|---|---|---|
| $prob(*)$ | $prob(//)$ | pruned filters | unpruned | leading | pruned | leading | unpruned | pruned |
| 0.2 | 0.0 | 199529 | 0 | 0 | 0 | 0 | 5.52 MB | 8.68 MB |
| 0.2 | 0.2 | 228133 | 62528 | 15823 | 8895 | 1238 | 10.58 MB | 9.89 MB |
| 0.2 | 0.4 | 235729 | 98779 | 32013 | 13208 | 2426 | 11.18 MB | 10.19 MB |
| 0.2 | 0.6 | 233143 | 118775 | 47991 | 15863 | 3580 | 10.24 MB | 10.13 MB |
| 0.4 | 0.2 | 346684 | 51203 | 12078 | 10868 | 918 | 10.79 MB | 14.96 MB |
| 0.6 | 0.2 | 538364 | 37712 | 7966 | 12433 | 652 | 9.22 MB | 22.89 MB |

**Table 2: Characteristics of pruned filters for workloads of 100 000 filters generated for the NASA DTD. The maximum depth of the generated filters was set to 8, the maximum depth of the NASA data. The columns titled "leading" give the numbers of leading "//" operators in unpruned and pruned filters, respectively.**

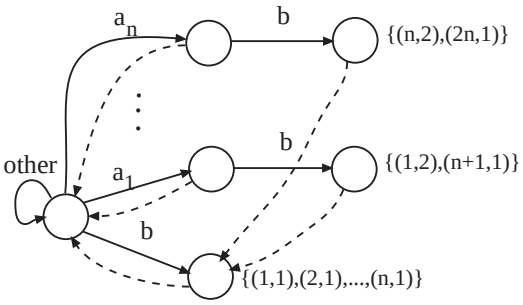



**Figure 4: Filtering PMA for a collection of $2n$ filters: $//b//a_1/b, \ldots, //b//a_n/b, //a_1/b, \ldots, //a_n/b$. The dashed arrows denote fail arcs. Only the nonempty output sets are shown. For example, $\mathit{output}(\mathit{state}(a_nb)) = \{(n,2),(2n,1)\}$, indicating recognition of the 2nd keyword ($a_nb$) of the $n$th filter and the 1st keyword ($a_nb$) of the $2n$th filter.**

by $string(q)$ the unique string $y$ with $state(y) = q$. Thus, $string(q)$ is the string upon which state $q$ is reached from the initial state via the goto function. We denote by $depth(q)$ the length of $string(q)$. In our experiments, the goto function is organized as an array indexed by state numbers. For state $q$, the entry $goto[q]$ is a hash table of pairs $(a, q')$ indexed by symbols $a$ in $\Sigma$. We use Java's library implementation of the hash table.

The *fail function* of the PMA is defined by the equation $fail(state(uv)) = state(v)$, where $uv$ is a prefix of some keyword and $v$ is the longest proper suffix of $uv$ such that $v$ is also a prefix of some keyword. For non-negative integer $k$, we denote by $fail^k$ the fail function applied $k$ times: $fail^0(q) = q$ and $fail^{k+1}(q) = fail(fail^k(q))$. The fail function is organized as an array indexed by state numbers.

The *output function* of our PMA is defined by setting for each state $q$:

$$output(q) = \{(i,j) \mid string(q) \text{ is the } j\text{th keyword in filter } i\}.$$

A pair $(i,j)$ is called an *output tuple*; the tuple signals the recognition of the $j$th keyword $string(q)$ of filter $i$ at state $q$. For example, if the number of filter "$/a/b//c/d$" is $i$ and $state(\#ab) = q_1$ and $state(cd) = q_2$, then $(i,1) \in output(q_1)$ and $(i,2) \in output(q_2)$.

The size of the output function is at most $O(|W|)$. Note that the number of states is at most $|W| + 1$ and that the total number of output-tuple instances in the output sets is at most the total number of keyword instances in all the filters, that is, the sum of the numbers of keywords in all filters.

When the PMA reports the output tuples in $output(q)$ at state $q$ then it must also report the output tuples in $output(fail^k(q))$ for the states $fail^k(q)$, $k > 0$, on the fail path from $q$ to the initial state. To avoid traversing the fail path in output reporting, we could include in $output(q)$ the output tuples from all states $fail^k(q)$ on the fail path. However, doing so would in some cases make the size of the output function quadratic in the size of $|W|$. This would happen for example in the case of the filters of Fig. 4. The size of the filter collection is $O(n)$, as is the size of the PMA with its goto, fail, and output functions, but copying the output of $state(b)$ to the output of all $state(a_jb)$, $j = 1, \ldots, n$, would make the output function of size $\Theta(n^2)$.

The price paid for keeping the size of the output function linear in $|W|$ is that the complete output for state $q$ must now be collected from the output sets on the fail path from state $q$ to the initial state. To avoid visiting states with an empty output set, we define the function *output_fail* by setting for state $q$: $output\_fail(q) = fail^k(q)$, if $k$ is the greatest integer less than or equal to $depth(q)$ such that $output(fail^m(q))$ is empty for all $m = 1, \ldots, k-1$. We call the path from state $q$ consisting of the *output_fail* arcs the *output path* of $q$.

The input stream for the PMA consists of tokens produced by a SAX parser. When the SAX parser encounters a start-element tag, the current state of the PMA is pushed onto a stack, and the symbol corresponding to the element name is consumed. The PMA changes its state according to the goto and fail functions, and keeps track of the matching filters. When the parser encounters an end-element tag, the current state of the PMA is set to the state on top of the stack, and the stack is popped.

The operating cycle of the PMA is given in Algorithm 4. The output set for state $q$ is scanned in the procedure call *report_output*$(q)$ (see Algorithm 5). To avoid scanning the same output set twice, a boolean array *output_scanned*, indexed by state numbers, is maintained. An array *result*, indexed by filter numbers, is used to store information about matched keywords in the filters. For filter number $i$, the entry $result[i]$ contains the set of indices $j$ for which the $j$th keyword in filter $i$ has been matched. The contents of the arrays *output_scanned* and *result* are initialized by the procedure call *initialize()* (Algorithm 6).

When the input document has been processed, the result

**Algorithm 4** Operating cycle of the backtracking PMA.

```
scan_next_input_token(token)
while token was found do
  if token is a start-document tag then
    initialize()
    state ← initial_state
    stack.push(state)
    sym ← #
    state ← goto(state, sym)
    report_output(state)
  else if token is an end-document tag then
    print_result()
  else if token is the start-element tag of element E then
    stack.push(state)
    sym ← symbol_table(E)
    while goto(state, sym) = fail do
      state ← fail(state)
    end while
    state ← goto(state, sym)
    report_output(state)
  else if token is an end-element tag then
    state ← stack.pop()
  end if
  scan_next_input_token(token)
end while
```

of the filtering can be read from the array *result*. For filter $i$ that consists of a single keyword, a match has been found if and only if $1 \in result[i]$. For filter $i$ that consists of $m$ keywords, a possible match has been found if $j \in result[i]$ for all $j = 1, \ldots, m$. Such a possible match is not always an exact match, because the matched keywords may not appear on the same path, nor in the specified order.

The following theorem states that our algorithm runs in time linear in the sum of the size of the filter collection and the length of the input document:

**Theorem 1.** When the size of the alphabet $\Sigma$ (or the number of distinct XML elements in the filters) is considered a constant, the time complexity of our filtering algorithm is $O(|W| + |x|)$, where $|W|$ denotes the sum of the lengths of all keyword instances appearing in the set of filters and $|x|$ is the length of input document $x$. □

To find out which possible matches of multi-keyword filters are true matches, the input document must be filtered through some more general filtering algorithm, such as the algorithm to be presented in the next section.

**Algorithm 5** Procedure *report_output*(*state*).

```
q ← state
while not output_scanned[q] do
  output_scanned[q] ← true
  for each (i, j) ∈ output(q) do
    insert j into result[i]
  end for
  q ← output_fail(q)
end while
```

## 4. MULTI-KEYWORD-FILTER PMA

In this section we extend the algorithm of the previous section so as to find exact matches for multi-keyword filters, that is, for filters that still contain one or more non-leading occurrences of the descendant operator "//" after being pruned by the algorithm of Sec. 2. This extended algorithm uses a PMA constructed for keywords appearing in all the pruned filters (whether single-keyword or multi-keyword) if a single-pass filtering process is used, or only for keywords appearing in multi-keyword pruned filters if a two-pass filtering process is used. In the latter case, the PMA is initialized to record matches of only those filters for which possible matches were found in the first pass.

**Algorithm 6** Procedure *initialize*().

```
for i = 1 to number_of_states do
  output_scanned[i] ← false
end for
for i = 1 to number_of_filters do
  result[i] ← empty
end for
```

To arrange that the keywords of each filter are matched against the same root-to-leaf path of the input document and in the order in which they appear in the filter, and to keep track of which keyword of each filter is currently being considered for matching, we maintain a vector *frontier* that records for each filter $i$ the number $j$ of the keyword currently being considered for matching with filter $i$. Initially, $frontier[i] = 1$ for all filters $i$ (see Algorithm 7). When the PMA is at state $q$, the output path of $q$ is traversed and for each state $q'$ in the path the output of $q'$ is scanned; for each output tuple $(i, j)$ in $output(q')$ where $j = frontier[i]$, a match of the $j$th keyword of filter $i$ is signalled and the frontier is advanced by setting $frontier[i] \leftarrow j + 1$ (see Algorithm 8). A match for the entire filter $i$ is found when a match is recorded for $(i, j)$ where $j$ has reached the number of keywords in filter $i$.

Since the set of keywords being considered for matching changes each time the frontier is advanced and only a subset of the output tuples of each visited state is reported at a time, we can no longer avoid repeated scanning of output sets. Thus the optimization that was possible for single-keyword filters cannot be used here; instead, each procedure call *report_output*($q$) must traverse the output path of state $q$ to find if some keyword matches are found for the current frontier (see Algorithm 9).

To avoid doing too much work in repeated scannings of the output sets, we make the output set of each state change dynamically whenever the frontier of some filter is advanced, so that the output set, now called the *current output set*, of each state contains only those output tuples $(i, j)$ from the static output set $output(q)$ for which $j = frontier[i]$. Thus, whenever the frontier of filter $i$ is advanced from $j$ to $j + 1$, the tuple $(i, j)$ must be deleted from the current output of all states, and the tuple $(i, j + 1)$ must be inserted into the current output of all states whose static output contains $(i, j + 1)$.

To make the updating of the current output sets efficient, we store each set $current\_output(q)$ as a doubly linked list of triples $(i, j, q)$. When the frontier of filter $i$ is advanced from $j$ to $j + 1$, triples $(i, j, q)$ are deleted from all lists that contain $(i, j, q)$ for some state $q$. This is accomplished by additionally maintaining for each pair $(i, j)$ a circular list of triples $(i, j, q)$. Thus, when a match of the $j$th keyword of filter $i$ is found at state $q$, the circular list is traversed

**Algorithm 7** Procedure *initialize*().

```
for all states q do
  current_output[q] ← empty
end for
for all filters i do
  if filter i is considered for matching then
    match[i] ← false
    frontier[i] ← 0
    update_frontier(i)
  else
    frontier[i] ← 1 + the number of keywords in filter i
  end if
end for
```

**Algorithm 8** Procedure *update_frontier*$(i, j)$.

```
for all states q with (i, j) ∈ current_output(q) do
  delete (i, j) from current_output(q)
end for
if j is the number of keywords in filter i then
  match[i] ← true
end if
if (i, j + 1) needs a counter then
  create a counter for filter i with initial value 0
end if
frontier[i] ← j + 1
for all states q with (i, j + 1) ∈ output(q) do
  insert (i, j + 1) into current_output(q)
end for
```

starting from $(i, j, q)$, and $(i, j, q')$ is deleted from all sets *current_output*$(q')$, that is, from doubly linked lists containing $(i, j, q')$. This is done in time linear in the number of triples $(i, j, q')$. As each such triple is touched in the deletion at most once, the total time taken by all the deletions is linear in the number of all keyword instances in the filters.

In addition to deleting triples $(i, j, q)$, triples $(i, j + 1, q')$ for all $q'$ such that the static output set of $q'$ contains $(i, j+1)$ must be inserted into the doubly linked list implementing the current output set of $q'$. These triples must also be added into the corresponding circular list of triples having $i$ as the first component and $j + 1$ as the second. As in the case of deletions, it is easy to see that the total time taken by inserting elements into current output sets is linear in the number of all keyword instances in the filters.

As such, this algorithm does not however always record correctly matches of filters in which a nonempty suffix of some keyword happens to be a prefix of the previous keyword, or if some keyword happens to include the previous keyword, in the filter. For example, with the input document $\langle doc\rangle\langle a\rangle\langle b\rangle\langle c\rangle\langle/c\rangle\langle/b\rangle\langle/a\rangle\langle/doc\rangle$ a match for the filter $//a/b//b/c$ is incorrectly recorded because when a match of the first keyword $(ab)$ is found at *state*$(ab)$, the frontier of the filter is advanced from 1 to 2, and then, at state *state*$(abc)$, a match of the second keyword $(bc)$ is incorrectly recorded. Similarly, a match for the filter $//b//a/b/c$ would be incorrectly recorded at *state*$(abc)$.

To solve the problem we maintain a *counter* for each filter $i$ if *frontier*$[i] = j > 1$ and if either a nonempty suffix of the $(j-1)$th keyword of filter $i$ is a prefix of the $j$th keyword of filter $i$ or the $(j-1)$th keyword of filter $i$ is a substring of the $j$th keyword of filter $i$. The counter for $i$ records the number of input symbols read since *frontier*$[i]$ was advanced from $j-1$ to $j$. When checking an output tuple $(i, j)$ in *output*$(q)$ at state $q$, if *frontier*$[i] = j$ and the value of the counter for $i$ is greater than or equal to *depth*$(q)$, we can safely record a match of the $j$th keyword of filter $i$ (see Algorithm 9).

**Algorithm 9** Procedure *report_output*(*state*).

```
q ← state
while q is not initial_state do
  for all (i, j) ∈ current_output(q) do
    if frontier[i] = j then
      if there is an active counter of filter i then
        if the value of the counter ≥ depth(q) then
          drop the counter
          update_frontier(i, j)
        end if
      else
        update_frontier(i, j)
      end if
    end if
  end for
  q ← output_fail(q)
end while
```

To avoid doing too much work in incrementing the counters, we take care that each counter is *active*, that is, in existence only for the time it is absolutely needed. We store the active counters in a doubly linked list that allows a counter to be created, retrieved and dropped in constant time and all the active counters to be incremented in time linear in the number of those counters. All active counters are incremented whenever a symbol is consumed by the PMA. The total time taken by the maintenance of the active counters, that is, creating, dropping, and increasing counters, in the processing of any input document is only linear in the number of all keyword instances.

The main program for the filtering PMA needs to be changed from Algorithm 4 as follows. First, a call for the incrementation of the counters is added just after the assignments of the variable *sym*. Second, besides the current state also the frontier advancements and counter droppings must be saved in the stack when the PMA encounters a start-element tag, and changed frontiers, current output sets of filters with changed frontiers, and the values of active counters must be restored using the information stored in the stack when the PMA encounters an end-element tag and is about to backtrack to the state popped from the stack.

The fact that backtracking now involves output-set updating and the fact that output paths must be traversed repeatedly introduce two new factors in the time complexity of our algorithm:

**Theorem 2.** When the size of the alphabet $\Sigma$ (the number of distinct elements) is considered a constant, our filtering algorithm determines in time $O(p_x|W| + k_W|x|)$ the subset of filters in $W$ that match a given input document $x$. Here $|W|$ denotes the sum of the lengths of all keyword instances appearing in the filters, $p_x$ is the number of root-to-leaf paths in $x$, $|x|$ is the length of $x$, and $k_W$ is the maximum, over all keywords $w$ in the filters, of the number of suffixes of $w$ that are also keywords (i.e., $k_W$ is the length of the longest possible output path). □

While the idea of making the output sets change dynamically when the frontier is advanced allows us to amortize the

complexity of scanning the output sets by $|W|$, so that the work done on output-set scanning is made independent of the length of the input document, the updating of the current output sets adds a significant overhead on the filtering, as compared with the algorithm of Sec. 3. Therefore we have also implemented a version of the algorithm in which output sets are not updated. Our experiments show that this simpler algorithm is in practice as efficient as (and sometimes even somewhat more efficient than) the more complicated algorithm that uses output-set updating.

## 5. EXPERIMENTAL RESULTS

We have implemented our algorithms in various versions and tested them on the protein-sequence and NASA data sets obtained from the Database Research Group of the University of Washington [12]. Workloads of linear XPath filters without predicates were generated using the XPath query generator described by Diao et al. [4], parameterized with the maximum depth of XPath queries and with the probabilities *prob*(//) and *prob*(∗) (see Sec. 2). The speed and throughput of filtering were measured using as input documents the entire 24 MB NASA data set and a 24 MB extract from the protein-sequence data set. The maximum nesting depth of the protein-sequence data is 7 and that of the NASA data is 8. More than 90 % of the filters matched the input document, when the filters had no predicates. When filters with value-based predicates are generated, the number of matching filters is smaller.

The tests were run on a Dell PowerEdge SC430 server with 2.8 GHz Pentium 4 processor, 3 GB of main memory, and 1 MB of on-chip-cache. The computer was running the Debian Linux 2.6.18 operating system with the Sun Java virtual machine 1.6.0_03 installed. In the tests the input document was read from the disk, but the overhead of the disk operations should be fairly small. The disk-read speed of the test hardware is more than 50 MB/sec. The throughput of the Java JAXP SAX parser (run in non-validating mode) on the two input documents was 25–28 MB/sec. For each measurement, the results are averages of five independent test runs.

Fig. 5 shows the effect of pruning on the filtering speed of YFilter [4]. The workloads of 10 000 to 100 000 XPath filters without predicates were generated with *prob*(∗) = *prob*(//) = 0.2. Pruning with the protein-sequence DTD eliminated exhaustively all "∗" and "//" operators, while pruning with the slightly recursive NASA DTD eliminated all "∗" operators and about 86 % of the "//" operators from the original filters. With the protein-sequence data set and with 10 000 filters, filter pruning increased the speed of YFilter by a factor of 3.5, and with 100 000 filters by a factor of 6.9. As seen from Fig. 5, with the NASA data set the speed-up was even more impressive: with 10 000 filters by a factor of 7.3, and with 100 000 filters by a factor of 22.8.

When filters may contain value-based predicates, the speed-up is not so impressive, because much of the total filtering time is spent on evaluating the predicates. However, our experiments with workloads of 10 000 filters having one predicate per filter (generated with *prob*(∗) = *prob*(//) = 0.2) show that the performance gain from filter pruning is still evident: the speed-up of YFilter was 1.3 protein-sequence data set and 2.0 for the NASA data set.

Tab. 3 gives times spent on pruning filters and on constructing the PMAs used by our two filtering algorithms.

| | protein | | NASA | |
|---|---|---|---|---|
| # XPath filters | 1000 | 10000 | 1000 | 10000 |
| Pruning time (sec) | 0.415 | 1.485 | 0.452 | 1.551 |
| PMA build time (sec) | 0.074 | 0.198 | 0.092 | 0.261 |
| # states in the PMA | 92 | 92 | 141 | 145 |
| Size of the PMA (MB) | 0.49 | 1.75 | 0.68 | 2.12 |

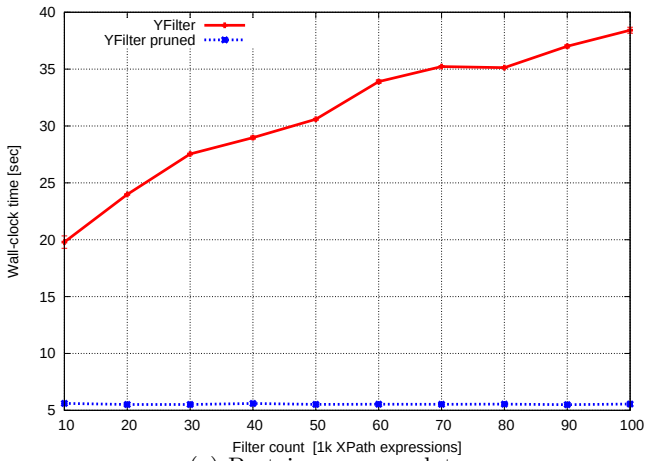
**Table 3: Times spent on pruning filters generated with *prob*(∗) = *prob*(//) = 0.2, and on constructing the single-keyword-filter PMA (for pruned protein-sequence filters) and the multi-keyword-filter PMA (for pruned NASA filters).**

These are preprocessing tasks that are done only at system startup, when new filters are added into the publish/subscribe system or when new types of documents are introduced into the stream. These filter workloads contain only distinct filters; no duplicates are included. For the filter workloads generated and pruned using the nonrecursive protein-sequence DTD the single-keyword-filter PMA of Sec. 3 was built, and for the recursive NASA DTD the multi-keyword-filter PMA of Sec. 4 was built.

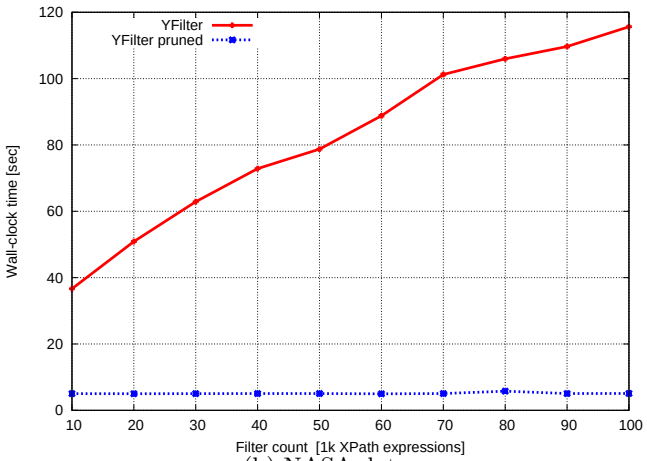
To keep the output sets of the PMAs as small as possible, all instances of a pruned filter resulting from pruning different original filters are given the same filter number if their XPath expressions are equal. The mapping from pruned filters to original filters must thus be maintained by an array *original_filters*, where an entry *original_filters*[$i$], for pruned filter number $i$, contains the numbers of original filters from which filter $i$ was pruned. This optimization is important because the same XPath expressions are shared by many filters.

The size of the PMA for the single-keyword-filter algorithm of Sec. 3 for pruned filters remained linear even with respect to the original unpruned filters: for a workload of $1000n$ filters generated with *prob*(//) = *prob*(∗) = 0.2 from the protein-sequence DTD, the size of the PMA was about $0.15n$ MB, for $n = 100, 200, \ldots, 1000$. The size of the PMA increases with *prob*(∗): for workloads of 100 000 filters with *prob*(//) = 0.2, the size of the PMA increased from 10 MB to 65 MB when *prob*(∗) was increased from 0 to 0.8. The obvious reason for this is that the filter-pruning algorithm produces growing numbers of possible paths (and keywords for the PMA) when there are more wildcards in the XPath filters [10].

The size of the PMA is not so sensitive to *prob*(//): for workloads of 100 000 pruned filters for the protein-sequence data generated with *prob*(∗) = 0.2, the size of the PMA decreased from 15.7 MB to 13.2 MB when *prob*(//) was increased from 0.2 to 0.8, and for corresponding workloads for the NASA data, the size of the PMA varied between 16.9 MB and 17.7 MB when *prob*(//) varied between 0.2 and 0.8. The number of distinct keywords resulting from pruning a million filters generated with *prob*(//) = *prob*(∗) = 0.2 from the protein-sequence DTD is 90 and the number of states in the PMA is 92. Since leading occurrences of descendant operators were also eliminated in the pruning, the maximum number of distinct keywords for the PMA is the number of different paths starting from the root element of the nonrecursive DTD. (When the DTD is recursive, the maximum number of distinct keywords is the number of different paths



(a) Protein-sequence data



(b) NASA data

**Figure 5: Filtering time of a 24 MB XML document extracted from (a) the protein-sequence data set and (b) the NASA data set, using YFilter with original unpruned filters ("YFilter") and with pruned filters ("YFilter pruned"). The XPath filters without predicates were generated with $prob(*) = prob(//) = 0.2$.**

without cycles in the DTD.) Due to the fact that keywords are shared by many pruned filters, the PMA has not very many states even for a large number of filters. The most memory is consumed by the data structure used to store the output sets of states.

We measured the times spent by our filtering algorithms on various workloads with unpruned and pruned filters and compared the times with those spent by YFilter [4]. With sets of 100 000 unpruned filters generated from the NASA DTD with $prob(*) = 0$ (i.e., no wildcards) and $prob(//) = 0.2$, 0,4 and 0.6, our multi-keyword-filter algorithm of Sec. 4 exhibited a performance comparable to that of YFilter when the maximum depth of the generated filters was set to that of the input document. With the nonrecursive protein-sequence data our algorithm performed better than YFilter, but with the highly recursive Treebank data [12] YFilter performed better than our algorithm. For these tests, our PMA was built using the optimization that for each different XPath expression only one filter number is maintained in the output sets; these filter numbers are mapped to corresponding numbers of original filters using the table *original_filters* (see above).

With pruned filters our algorithm exhibits a better performance than YFilter. Tab. 4 shows filtering times of the 24 MB input document extracted from the protein-sequence database when filtered with pruned filters using our single-keyword-filter algorithm of Sec. 3 and YFilter. Workloads of 10 000 and 100 000 filters were generated from the nonrecursive protein-sequence DTD with different values of $prob(*)$ and $prob(//)$, and the filters were pruned eliminating all wildcards "$*$" and all descendant operators "//". The figures show that the filtering speed of our algorithm is 2 to 3 times that of YFilter. Curiously enough, for both algorithms, the filtering speed is in some cases slightly better with 100 000 filters than with 10 000 filters. Experiments done by Green et al. [6] show a similar behavior. They report that the filtering speed of YFilter on the protein-sequence data is higher with 10 000 filters than with 1 000 filters for filters generated with $prob(*) = prob(//) = 0$.

Tab. 5 shows filtering times of the NASA 24 MB input document when filtered with pruned filters using our multi-keyword-filter algorithm of Sec. 4 and YFilter. Workloads of 10 000 and 100 000 filters were generated from the recursive NASA DTD with the same values of $prob(*)$ and $prob(//)$ as with the protein-sequence data. In this case the filters were pruned eliminating all wildcards "$*$", but because of recursion some descendant operators "//" were left in the filters. The figures show that our algorithm and YFilter have more or less the same performance on this data. For greater $prob(*)$ values our algorithm is slightly better than YFilter while for greater $prob(//)$ values YFilter is slightly better than our algorithm. For our algorithm, greater $prob(*)$ values mean more pruning (since all wildcards are eliminated) while greater $prob(//)$ values mean that more descendant operators remain in the pruned filters (cf. Tab. 2).

As both our filtering algorithms allow leading occurrences of the descendant operator to remain in the filters, we also ran tests on pruned workloads in which the elimination of descendant operators was restricted to non-leading occurrences. With 100 000 filters for the NASA data, the performance gain from eliminating also leading descendant operators was 37 % for YFilter and 30 % for our multi-keyword-filter algorithm. Eliminating also leading descendant operators decreased the number of distinct keywords from 707 to 143, and the number of states in the PMA from 729 to 145. Tests with the protein-sequence data showed that for our single-keyword-filter algorithm the effect of elimination of leading descendant operators is insignificant.

When the maximum depth of the generated XPath filters was set to a much higher value than the actual depth of the data, the performance of our filtering algorithm decreased. An obvious reason for this is that the number of matching filters decreases when the workload contains such "too deep" filters; such filters cause the frontier and output sets to be updated back and forth. The performance of YFilter somewhat increased when the workload contained "too deep" filters.

We also measured the throughput of filtering using the en-

| | | our PMA | | YFilter | |
|---|---|---|---|---|---|
| $prob(*)$ | $prob(//)$ | 10 000 | 100 000 | 10 000 | 100 000 |
| 0.2 | 0.2 | 2.026 | 2.062 | 5.610 | 5.473 |
| 0.2 | 0.4 | 2.039 | 2.076 | 5.556 | 5.593 |
| 0.2 | 0.6 | 2.063 | 2.046 | 5.609 | 5.562 |
| 0.4 | 0.2 | 2.023 | 2.105 | 5.536 | 5.616 |
| 0.6 | 0.2 | 2.069 | 1.882 | 5.555 | 5.618 |

**Table 4: Filtering time (in seconds) of an 24 MB input document with 10 000 and 100 000 pruned XPath filters generated from the nonrecursive protein-sequence DTD, for the single-keyword-filter algorithm of Sec. 3 and for YFilter.**

| | | our PMA | | YFilter | |
|---|---|---|---|---|---|
| $prob(*)$ | $prob(//)$ | 10 000 | 100 000 | 10 000 | 100 000 |
| 0.2 | 0.2 | 3.390 | 5.694 | 5.099 | 5.107 |
| 0.2 | 0.4 | 4.647 | 6.625 | 5.765 | 5.063 |
| 0.2 | 0.6 | 4.927 | 7.170 | 5.142 | 5.023 |
| 0.4 | 0.2 | 3.730 | 5.286 | 5.038 | 5.063 |
| 0.6 | 0.2 | 4.465 | 4.719 | 5.040 | 5.100 |

**Table 5: Filtering time (in seconds) of an 24 MB input document with 10 000 and 100 000 pruned XPath filters generated from the recursive NASA DTD, for the multi-keyword-filter algorithm of Sec. 4 and for YFilter.**

tire protein-sequence database of 683 MB as the input document. In this case the parser throughput was 27 MB/sec and the filtering throughput (with single-keyword filters) was 15.7 MB/sec for one million filters. The throughput is better for bigger input documents, because the reporting of the matched filters, the initialization of the PMA (Algorithm 6) and the warm-up phase of the SAX parser are then amortized by the length of the input document.

Green et al. [6] have compared the filtering speed of their lazy DFA algorithm with that of YFilter on the protein-sequence database. They also generated the filter workload with YFilter's generator with $prob(*) = prob(//) = 0.1$. With these settings and 100 000 unpruned filters the lazy DFA was 8.3 times faster than YFilter. Our preliminary experiments with the lazy DFA indicate that also this method can gain from filter pruning.

## 6. RELATED WORK

This work was inspired by recent automata-based methods for XPath query evaluation [4, 6, 9], and by the query-pruning technique of Fernández and Suciu [5] who used graph schemas to optimize regular path expressions. The idea of query pruning is that the selectivity of the schema is embedded into the queries. The technique of Fernández and Suciu [5] takes a user-provided query (a path expression) and a graph schema as input and constructs the product automaton of two NFAs: one that accepts the paths denoted by the query and another that accepts the paths denoted by the graph schema; from this product NFA, a pruned query is constructed by taking into account only those paths in the NFA that lead from the initial state to one of the final states. Lee et al. [8] present a very similar idea for pruning XPath filters with a DTD: they construct the product automaton of an automaton representing the XPath filters and an automaton representing the DTD.

The pruning techniques of Fernández and Suciu [5] and Lee et al. [8] thus use the graph schema to add selectivity to the filters while in all cases keeping the size of the pruned filters polynomial in the sum of the sizes of the original filters and the graph schema. Our approach to pruning XPath filters is different (Sec. 2). Our goal is use the DTD to eliminate certain operators altogether ("$*$") and to reduce the number of certain operators to the extent possible (non-leading "//"). In this way it may be possible to use a more efficient filtering algorithm such as the single-keyword-filter algorithm of Sec. 3 (possible when no non-leading "//" operators remain) and to speed up a general filtering algorithm such as the multi-keyword-filter algorithm of Sec. 4 (which gains even from partial elimination of "//" operators).

The elimination of all descendant operators "//" is only possible for nonrecursive DTDs, and eliminating all wildcards "$*$" may lead to pruned filters of size exponential in the sum of the sizes of the original filters and the DTD, even for nonrecursive DTDs. However, several heuristics can be used to restrict operator elimination in our pruning algorithm to keep the size of the pruned filters polynomial. On the other hand, in our experiments thus far the PMA representation of pruned filters has remained moderate even if all wildcards are eliminated.

We have shown that, besides our filtering algorithms, also the NFA-based YFilter algorithm of Diao et al. [4] can gain significantly from filter pruning. When all wildcards and all non-leading descendant operators can be eliminated from the filters, the path-sharing principle used in the construction makes YFilter's NFA come close to our PMA. However, a marked difference between YFilter's NFA and our PMA is that our PMA is always deterministic, no matter how many descendant operators remain in the pruned filters. On the other hand, YFilter allows wildcards to remain in pruned filters, while our algorithms do not.

In the DFA-based filtering algorithm of Green et al. [6], the goal is to circumvent the exponential growth of the DFA by constructing it lazily. They have shown that the size of the DFA is guaranteed to remain within tractable limits if the DTD is nonrecursive or contains only simple cycles. However, in the case of a more complex DTD the algorithm may still run out of memory. To improve the memory usage of DFA-based algorithms, Onizuka [9] suggests partitioning the set of XPath filters into clusters and constructing a DFA for each cluster.

Our filtering algorithms are derived from the classical Aho–Corasick PMA [1], with added facilities for reporting matching filters, backtracking for matching paths in tree-structured input, and for matching sequences of keywords in paths. In all cases, our PMA is of size linear in $|W|$, the sum of the sizes of the (pruned) filters. Moreover, our single-keyword-filter algorithm (which can be used when all non-leading descendant operators are eliminated) runs in time linear in $|W| + |x|$, and even our multi-keyword-filter algorithm has a worst-case time bound that is below $O(|W| \cdot |x|)$, the time spent on simulating an NFA of size $|W|$ on an input string of length $|x|$.

Our method can be considered to somehow fall in between the lazy-DFA method and the path-sharing NFA-

based YFilter. Our algorithms are partly independent of the number of filters, because the number of steps performed by the PMA only depends on the length of the input string, but our algorithms depend on the number of filters in reporting the matched filters. Thus the basic automaton-based structural analysis is as by a DFA, but in our multi-keyword-filter algorithm the output sets change dynamically over time.

Diao et al. [4] show how value-based filtering can be efficiently combined with the structural analysis of YFilter. As yet, our filtering algorithms accept only linear XPath filters without predicates; we are currently experimenting with different ways to include the evaluation of value-based predicates in our algorithms [10]. YFilter can also process branching XPath filters (twig filters), by decomposing them into linear filters. These linear filters are then matched by using the NFA, and matching twig filters are identified at a post-processing phase. In this case it is not sufficient to only search for first occurrences of the linear component filters, because a match of a twig filter may only be found with a combination of other than first occurrences of the component filters. Onizuka [9] presents a DFA-based algorithm for filtering with twig filters. Filter-pruning algorithms such as ours can readily be applied to twig filters, because the linear filters decomposed from a twig filter can always be pruned independently.

# 7. CONCLUSIONS

We have presented DTD-conscious algorithms for the filtering problem of XML documents in publish/subscribe systems where subscribers specify their interests with linear XPath expressions. Our filter-pruning algorithm (Sec. 2) can be used as a preprocessing task of any XML filtering algorithm to eliminate wildcards "$*$" and descendant operators "//" from the original subscriber-provided XPath filters when the XML documents are known to conform to a DTD. The algorithm allows for different heuristics to be used to regulate the elimination of "$*$" and "//" operators when exhaustive elimination is impossible (for "//") or would result in a set of pruned filters whose total size is too large. Our experiments conducted with YFilter [4] on XML data sets obtained from the XML Data Repository of the Univ. of Washington [12] indicate that filter pruning can significantly speed up the filtering process. The effect of pruning on the number of filters remained moderate, so that the number of pruned filters was not more than 2 to 6 times the number of original filters. The effect of pruning on the size of YFilter's NFA remained insignificant.

Our filtering algorithm (Secs. 3 and 4) uses a deterministic automaton derived from the Aho–Corasick PMA [1] by adding a facility for reporting matching filters and a backtracking facility for recognizing tree-structured input. The algorithm comes in two versions, one of which finds exact matches for pruned filters in which all wildcards and non-leading descendant operators are eliminated (Sec. 3), while the other can find exact matches for pruned filters in which non-leading descendant operators cannot all be eliminated (Sec. 4). In both cases the PMA is of size linear in the sum of the sizes of the pruned filters, and stringent polynomial upper bounds hold on the time of filtering; the time bound for the single-keyword-filter algorithm of Sec. 3 is linear in the size of the collection of pruned filters plus the length of the input document (Theorem 1), and the time bound for the multi-keyword-filter algorithm of Sec. 4 is well below the bound for simulation of nondeterministic automata (Theorem 2). Our filtering method is especially amenable to filter pruning. Although with unpruned filters our algorithm does not exhibit a performance distinctly superior to YFilter, with pruned filters in which all wildcards and all non-leading descendant operators have been eliminated our algorithm can outperform YFilter by a factor of two or three. Our algorithm also competes with YFilter in cases in which some non-leading descendant operators remain in the pruned filters.

# 8. ACKNOWLEDGEMENTS

The work of Eljas Soisalon-Soininen was supported by the Academy of Finland.

# 9. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communcations of the ACM*, 18(6):333–340, 1975.

[2] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB 2000, Proc. of 26th Internat. Conf. on Very Large Data Bases*, pages 53–64, 2000.

[3] P. Buneman, S. B. Davidson, M. F. Fernández, and D. Suciu. Adding structure to unstructured data. In *ICDT'97, Proc. of the 6th Internat. Conf. on Database Theory*, pages 336–350, 1997.

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

[5] M. F. Fernández and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proc. of the 14th IEEE Internat. Conf. on Data Engineering*, pages 14–23, 1998.

[6] 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, 2004.

[7] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. of the 2003 ACM SIGMOD Internat. Conf. on Management of Data*, pages 419–430, 2003.

[8] D. Lee, H. Shin, J. Kwon, W. Yang, and S. Lee. SFilter: schema based filtering system for XML streams. In *MUE 2007, Internat. Conf. on Multimedia and Ubiquitous Engineering, Seoul, Korea*, pages 266–271, 2007.

[9] M. Onizuka. Light-weight XPath processing of XML stream with deterministic automata. In *Proc. of the 2003 ACM CIKM Internat. Conf. on Information and Knowledge Management*, pages 342–349, 2003.

[10] P. Silvasti, S. Sippu, and E. Soisalon-Soininen. XML-document-filtering automaton. *Proc. of the VLDB Endowment*, 1(1):1666—1671, 2008.

[11] E. Soisalon-Soininen and T. Ylönen. On classification of strings. In *String Processing and Information Retrieval, 11th Internat. Conf., SPIRE 2004, Proceedings*, pages 321–330, 2004.

[12] D. Suciu. XML data repository. The Database Research Group, University of Washington, 2006. `www.cs.washington.edu/research/xmldatasets/`.