XML-Document-Filtering Automaton

Panu Silvasti
Helsinki University of Technology
psilvast@cs.hut.fi
Supervisors: Seppo Sippu and Eljas Soisalon-Soininen

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. We present a new XML-document-filtering algorithm that is based on the classic Aho–Corasick pattern-matching automaton. The automaton has a size linear in the sum of the sizes of the filters. We assume that the XML documents all conform to a given DTD; our algorithm utilizes the DTD in the preprocessing phase of the automaton to prune out descendant axes (//) and wildcards (∗) from the XPath filters. The XPath subset currently supported consists of linear XPath expressions without predicates. In the case of a 683 MB protein-sequence database, we obtained a throughput of 18.8 MB/sec for 50 000 filters and 17.0 MB/sec for 500 000 filters, using a SAX parser with a throughput of 27 MB/sec.

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. Publish-subscribe systems have emerged in everyday use; examples include *Google alerts* and stock-information delivery by *Yahoo.com*. Designing efficient techniques for filtering of XML documents has received much attention in the research of data-stream management [2, 4, 6, 8, 9, 10, 11], and XML-filtering techniques have been applied in areas such as routing real-time air traffic control data [13].

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.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-306-8/08/08

The primary problem we address in this paper is defined as the *filtering problem*: given a set of XPath expressions, identify those expressions that match a given XML document. We study the filtering problem for linear XPath expressions. Linear XPath expressions do not have branches in their query trees and are described by the following grammar:

```
P := /E | //E | PP
E := label | *
```

where *label* denotes an XML-element label.

Several approaches to XML filtering use a finite automaton as a basis of the filtering algorithm. Diao et al. [6] report an XPath-query-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 prefix-sharing, that is, by merging states that correspond to common prefixes in different query paths.

The algorithm of Green et al. [8] 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 DTD that is nonrecursive or contains only simple cycles (a cycle is simple if its nodes do not occur in other cycles). The lazy DFA can process only linear XPath expressions. Onizuka [11] present another DFA-based algorithm for XML filtering; this algorithm can also process branching XPath expressions (or "twig filters"). Onizuka also proposes several techniques for improving the memory usage of the DFA.

Our approach to XML filtering is also based on the use of a DFA. The basis of our algorithm is the classic Aho–Corasick [1] pattern-matching automaton (PMA). The PMA is constructed from the set of filters in the preprocessing phase of the algorithm. The memory usage of our algorithm is linear in the sum of sizes of the filters. The idea of using Aho–Corasick for XML document filtering is presented in article [14].

In the preprocessing phase we utilize the DTD or schema of the XML documents in optimizing the XPath queries, when such a DTD exists. Our optimization method, called *filter pruning*, was inspired by the query-pruning technique by Fernández and Suciu [7], and it takes a linear XPath expression possibly containing "//" and "∗', and outputs a set of keywords that can be matched by using the standard pattern-matching algorithm, such as Aho–Corasick, when the input stream for the algorithm consists of SAX-parser events. Our current implementation of the algorithm can filter XML documents that conform to a nonrecursive DTD.

Our preliminary experimental results show that our algorithm provides good throughput for filtering, regardless of the number of filters. In the case of a 683 MB protein-sequence database [15], we obtained a throughput of 18.8 MB/sec for 50 000 filters and 17.0 MB/sec for 500 000 filters, using a SAX parser with a throughput of 27 MB/sec on the input document. Benchmarking with YFilter also produces good results, our PMA being 10.7 to 26.5 times faster than YFilter.

Our paper is organized as follows. In Sec. 2 we present our algorithm, and in Sec. 3 experimental results of the memory consumption and filtering speed, as well as limited benchmarking with YFilter. In Sec. 4 we present conclusions and in Sec. 5 we describe our upcoming experiments and some ideas for improving the filtering algorithm even further.

2. THE ALGORITHM

The standard Aho–Corasick PMA is a recognizer for a finite set of nonempty keywords $w_1 \cup \ldots \cup w_n$ over an input alphabet Σ [1]. In our setting, the alphabet Σ contains the set of elements occurring in the DTD plus an additional symbol # denoting the root root element of any XML document. The keywords w_i are derived from the XPath filters provided by the subscribers. Each keyword may come from one or more such filter. We modify the standard PMA so that, upon recognition of a keyword, the PMA is able to report exactly which of the filters match.

An XPath filter containing no "//" or "∗" gives rise to a single keyword w in Σ^*. For example, the filters $P_1 = /a/b/f$ and $P_2 = /a/c/f$ are represented as keywords $w_1 = \#abf$ and $w_2 = \#acf$. Here XML element names are represented as unique lexical symbols in our alphabet Σ, rather than as strings of characters.

Our goal is to express an XPath filter containing "//" or "∗" as a disjunction of keywords. This can be done by utilizing the DTD to "prune out" the operators "//" or "∗". We call this technique *filter pruning*, motivated by the query-pruning technique presented by Fernández and Suciu [7].

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, representing the root element of an XML document, that has no incoming edges. G cannot be cyclic, since our current implementation does not yet allow recursive DTDs. Figure 1 shows a sample DTD and its graph schema.

Now each keyword defines a path in G. A linear XPath filter consists of successive keywords $w_1, \ldots, w_k$ separated by "//" or "∗". For example, for the filter $P = //a//f/i$ the successive keywords are $w_1 = a$ and $w_2 = fi$. The idea is to use G to "fill in the gap" between any two successive keywords w_i, w_{i+1} by finding out what actual element strings x_{i+1} can appear between w_i and w_{i+1} in paths of G and by replacing the keyword pair by a set of single keywords $w_i x_{i+1} w_{i+1}$. Table 1 shows some XPath filters and the corresponding sets of keywords that denote paths in G.

```
<!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*)>
```

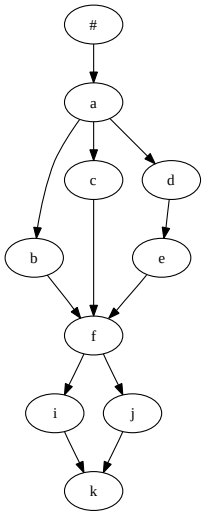

Figure 1: A DTD and its graph schema.

XPath filter	Keywords
$/a/b/f$	$\#abf$
$//b/f$	bf
$/a//f$	$\#abf \cup \#acf \cup \#adef$
$//c/f//k$	$cfik \cup cfjk$
$/*/b$	$\#ab$
$/a/*$	$\#ab \cup \#ac \cup \#ad$
$/a/*/f$	$\#abf \cup \#acf$
$/*/*/*/*$	$\#abfi \cup \#abfj \cup \#acfi \cup \#acfj \cup \#adef$

Table 1: XPath filters and corresponding pruned keywords. "#" symbol in a keyword denotes the XML documents root element.

In the preprocessing phase, our filter-pruning algorithm takes XPath filters possibly containing "//" and "∗', and outputs a set of keywords. The Aho–Corasick PMA is constructed for the set of all keywords thus obtained. Figure 2 shows the PMA for matching keywords $w_1 = \#abf, w_2 = bf, w_3 = \#acf$ and $w_4 = \#adef$. Each keyword w is given a unique identifier, $id(w)$, and the output set associated with each state contains the identifiers of keywords recognized at that state.

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 automaton is pushed onto a stack, and the symbol corresponding to the element name is consumed by the automaton. The automaton changes its state according to Aho–Corasick's *goto* and *fail* functions, and keeps track of the matching filters. On each *end element* tag, the current state of the automaton is set to the state on top of the stack, and the stack is popped. When the input document has been processed, the algorithm reports the filters that match the input the document.

Our algorithm uses the following data structures, where

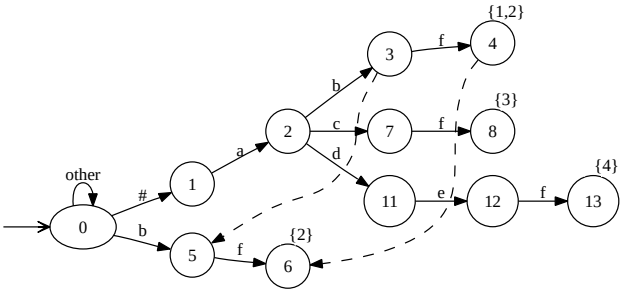

Figure 2: The Aho–Corasick automaton for matching keywords $w_1 = \#abf, w_2 = bf, w_3 = \#acf$ and $w_4 = \#adef$. The dashed lines denote fail arcs. There is also a fail arc to the initial state 0 from states 1, 2, 5, 6, 7, 8, 11, 12 and 13.

#words is the number of distinct keywords, *#filters* is the number of filters and *#states* is the number of states.

goto$[1 \ldots \#states]$: an array representing the *goto* function of the Aho–Corasick PMA. For state q, the entry *goto*$[q]$ is a hash table of pairs (a, q') indexed by input symbols a. We use Java's library implementation of the hashtable.

fail$[1 \ldots \#states]$: an array representing the *fail* function of the PMA.

filters$[1 \ldots \#words]$ is an array where *filters*$[id(w)]$ contains the numbers of those filters that contain the given keyword w.

result$[1 \ldots \#filters]$ is a boolean array where *result*$[i]$ is *true* if i'th filter matches the input document.

output_visited$[1 \ldots \#states]$ is a boolean array where *output_visited*$[s]$ is *true* if state s has been visited during the processing of an input document. When state s has been visited, we set *output_visited*$[s]$ to *true* and do not scan the same output set again for the input document.

The operating cycle of the algorithm is presented in Algorithm 1. The input for the procedure are the tokens produced by the SAX parser.

Algorithm 1 The operating cycle of the PMA

```
if inputToken is startDocument then
  initialize()
  stack.push(state)
  sym ← #
  state ← goto(state, sym)
else if inputToken is endDocument then
  print_result()
else if inputToken is startElement(elName) then
  stack.push(state)
  sym ← symbol_table(elName)
  while goto(state, sym) = fail do
    state ← fail(state)
  end while
  state ← goto(state, sym)
  report_output(state)
else if inputToken is endElement then
  state ← stack.pop()
end if
```

Algorithm 2 Procedure *report_output*(*state*).

```
if output_visited[state] = false then
  for each id in output[state] do
    for each i in filters[id] do
      result[i] ← true
    end for
  end for
  output_visited[state] ← true
end if
```

Algorithm 3 Procedure *initialize*.

```
state ← initial_state
for i = 1 to #states do
  output_visited[i] ← false
end for
for i = 1 to #filters do
  result[i] ← false
end for
```

3. EXPERIMENTAL RESULTS

We implemented the algorithm in Java, and tested it on a 10 MB input document that is part of a protein-sequence database of obtained from the Database Research Group of the University of Washington [15]. The data has a maximum nesting depth of 7, average depth of 5.15, and the protein DTD contains 66 elements. Figure 3 shows the graph schema of the protein DTD. The filter workload was generated using the XPath query generator described by Diao et al. [6]. The workload of linear XPath expressions was parameterized by the following parameters: *prob*(//), the probability of "//" being the operator at a location step, and *prob*(∗), the probability of wildcard "∗" occurring at a location step. The filter workload does not contain predicates; it may contain duplicate filters, which is most likely the case with real-world filters. Figure 4 shows part of a workload. With these settings more than 90 % of the filters matched the input document.

The tests were run on a Dell PowerEdge SC430 server with 2.8 GHz Pentium 4 processor, 1 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 input document was 21.4 MB/sec. For each measurement, the results are averages of five independent test runs.

The linear size of the PMA with respect to the size of the filters was evident in our experiments, as shown in Figure 5. Figures 6 and 7 show the size of the PMA with respect to *prob*(∗) and *prob*(//). The measurements indicate that when *prob*(∗) increases, the size of the PMA grows. 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 wildcard operators in the XPath queries. An example of such a filter is the last on in Table 1. However, the size of the PMA is not so sensitive to *prob*(//).

The number of distinct keywords in the PMA for 500 000

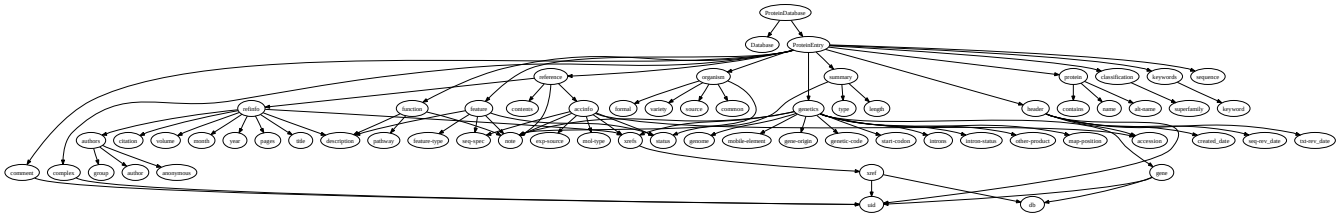

Figure 3: The graph schema of the protein DTD. The DTD has a maximum nesting depth of 7, and it contains 66 elements. The protein data has an average depth of 5.15.

```
/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 4: Part of filter workload generated with $prob(//) = prob(*) = 0.2$.

filters having $prob(//) = prob(*) = 0.2$ is 327 and the number of states is 352. The PMA has not very many states even for a large number of filters. The most memory is consumed by the auxiliary data structure *filters*.

Figure 8 shows the throughput of filtering calculated from the time spent on filtering the single 10 MB input document. The throughput decreases from 14.2 MB/sec for 50 000 filters to 11.6 MB/sec for 500 000 filters. The reason for decreased throughput is that processing also includes reporting matching filters, which is dependent of the number of filters. The time complexity of Aho–Corasick is not entirely linear in the length of the input, but $O(x \log |\Sigma| + k)$, where x is the size of the input, $|\Sigma|$ the size of the alphabet and k the number of matched patterns.

The throughput of filtering was also measured with the whole protein-sequence database of 683 MB as the input document. In this case the parser throughput was 27 MB/sec and the filtering throughput was 18.8 MB/sec for 50 000 filters and 17.0 MB/sec for 500 000 filters. The throughput is better for bigger input documents, since reporting of the matched filters, the initialization of the PMA (Algorithm 3) and the warm-up phase of the SAX parser are amortized by the length of the input document.

Table 2 shows absolute matching times for the 10 MB input document, comparing our algorithm with Yfilter [6]. For our algorithm, the preprocessing phase of the PMA was excluded from the filtering time, but the time spent on parsing the input document is included. For Yfilter, the parsing time is excluded from the filtering time. The results show that the speed of filtering of the PMA is nearly independent of the number of filters and of the number of "//" and "∗" operators in the filters. The filtering speed is even slightly better with 100 000 filters than with 10 000 filters.

	PMA		YFilter	
XPath expressions	10k	100k	10k	100k
$prob(*) = 0.2, prob(//) = 0.2$	0.79	0.72	8.52	16.72
$prob(*) = 0.2, prob(//) = 0.4$	0.80	0.72	9.43	19.09
$prob(*) = 0.2, prob(//) = 0.6$	0.81	0.74	8.67	18.91
$prob(*) = 0.4, prob(//) = 0.2$	0.65	0.72	8.89	17.53
$prob(*) = 0.6, prob(//) = 0.2$	0.67	0.75	7.55	15.14

Table 2: Filtering time (in seconds) of the 10 MB input document compared with YFilter. With these settings more than 90 % of the filters matched the input document.

This behaviour happens only with a relatively small number of filters, and, as Figure 8 shows, the filtering speed (or throughput) will decrease when the number of filters increases beyond 100 000. The results of Table 2 also indicate that with current settings the filtering speed of the PMA is better than that of YFilter.

Green et al. [8] have measured the filtering speed of their lazy DFA algorithm and YFilter with the protein-sequence database. They also generated the filter workload with YFilter's generator having $prob(*) = prob(//) = 0.1$. With these settings and 100 000 filters the lazy DFA was 8.3 times faster than YFilter. With nearly same settings ($prob(*) = prob(//) = 0.2$) our PMA is 23.2 times faster than YFilter.

4. CONCLUSIONS

We have presented a new algorithm for XML document filtering. The algorithm is based on the Aho–Corasick [1] pattern-matching automaton and it has a size linear in the sum of the sizes of the pruned filters. Our current implementation of the algorithm supports a subset of XPath filters, namely linear XPath expressions without predicates. Filtering of XML documents that conform to a nonrecursive DTD is currently supported. The algorithm utilizes the DTD to prune the subscriber-provided filters.

Our experiments with a protein-sequence database show that our algorithm provides good throughput for XML filtering. Benchmarking with YFilter [6] also produces good results, our algorithm being 10.7 to 26.5 times faster than YFilter. The lazy DFA [8] was not compared directly with our algorithm, but experiments with YFilter indicate that our algorithm could be 2.8 times faster than the lazy DFA on the protein-sequence database and with 100 000 filters. However, the lazy DFA is more general than our algorithm; it does not need the DTD, it can also process recursive XML documents, and it supports value-based predicates in filters.

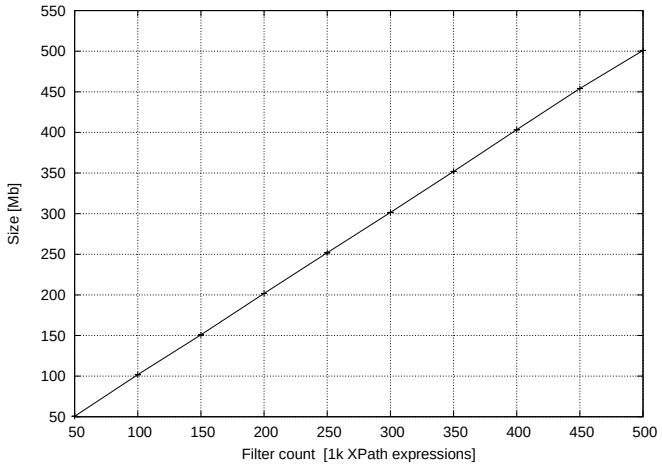

Figure 5: Size of the PMA in relation to the number of filters. $prob(//) = prob(*) = 0.2$**.**

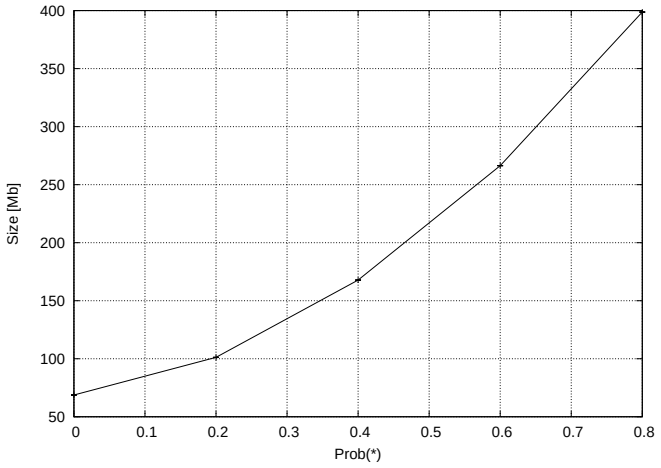

Figure 6: Size of the PMA in relation to $prob(*)$**.** $prob(//) = 0.2$ **and the number of filters is 100k.**

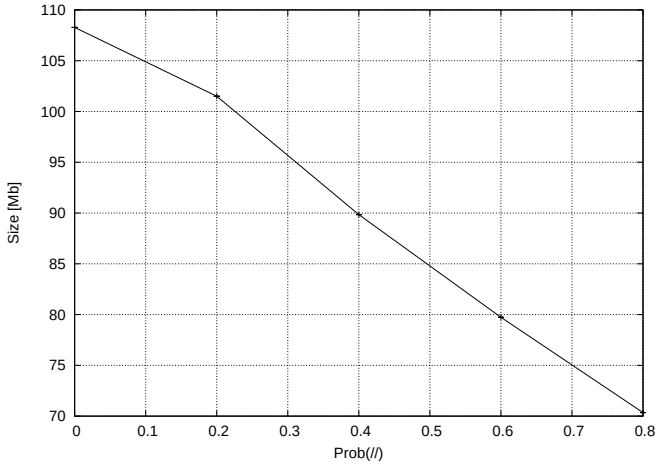

Figure 7: Size of the PMA in relation to $prob(//)$**.** $prob(*) = 0.2$ **and the number of filters is 100k.**

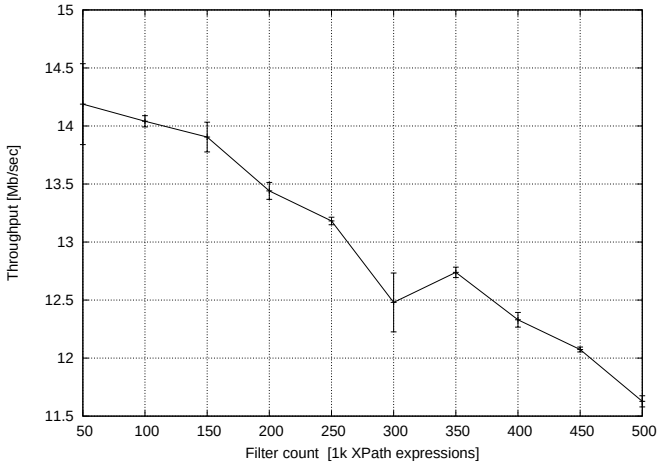

Figure 8: Throughput of filtering in relation to the number of filters. $prob(//) = prob(*) = 0.2$**. The results are averages of five independent test runs. Error bars denote standard deviation.**

5. FUTURE WORK

In order to obtain more convincing results it is necessary to experiment with several other DTDs and datasets available [15], besides the protein-sequence database. Also synthetic XML data could be used. Green et al. [8] used input documents as big as with a size of hundreds of megabytes in experimenting with the lazy DFA. However, the experiments performed with e.g. YFilter [6] and FiST [10] have been done with smaller documents (with a size of tens of kilobytes). We feel that in our upcoming experiments the throughput should also be measured by filtering a continuous stream of many smaller XML documents rather than one big document of many megabytes. As our preliminary experiments indicate, the throughput is expected to decrease, when the document size decreases.

For some DTDs the filter pruning can produce theoretically an exponential number of keywords for some XPath filters. A simple example is a 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, $i = 1, \ldots, k$. Pruning the filter $/a_1//a_{k+1}$ would result in a filter of size $\Theta(k2^k)$, when the DTD is of size $O(k)$. In another submitted paper, we have presented a method that has a polynomial upper bound on the size of the pruned filters (and the resulting PMA) with respect to the size of the original, subscriber-provided filters. Our future plans include implementation and experimentation of this algorithm. We also believe that processing of XML documents conforming to a recursive DTD can be handled with this approach.

The XPath subset considered here can be extended with filters having value-based predicates of the simple forms */a/b[text()='value']* and */a/b[@attr='value']*. The evaluation of such predicates can be done with the same PMA as the structural analysis by encoding the predicates as a part of the PMA so that the characters contained in the predicates give rise to state transitions of the PMA. The time complexity of this method is independent of the number of filters.

We have also developed a method for evaluating more complex value-based predicates. The predicates of a filter are evaluated whenever the structure part matches. This method is very similar to the method of Diao et al. [6]; its time complexity is dependent on the number of filters.

Extending the XPath subset with branching XPath filters (or "twig filters") [6, 10, 11] is a challenging problem, because automata-based filtering algorithms such as ours are designed to find only the first matching occurrences of the filters.

Diao et al. [6] have measured the maintenance cost of YFilter, that is, the cost of adding and deleting filters. Our aim is to address this question as well in the future.

Salmela et al. [12] have experimented with string-matching algorithms that are faster than the Aho–Corasick PMA for matching multiple string patterns. Their algorithms match overlapping q-grams instead of single characters, so that, for example, the word "pony" might be transformed into the 2-gram string "po-on-ny", before giving it as input to the pattern-matching algorithm. These new pattern-matching algorithms provide better preprocessing and matching times, and require less memory than the Aho–Corasick PMA. A related idea is considered by Dharmapurikar et al. [5], who modify the Aho-Corasick PMA to consider multiple characters at a time for content filtering. These approaches might also prove useful in XML filtering.

6. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. 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*, pages 53–64, 2000.

[3] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding structure to unstructured data. In *ICDT '97: Proceedings of the 6th International Conference on Database Theory*, pages 336–350, London, UK, 1997. Springer-Verlag.

[4] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal*, 11(4):354–379, 2002.

[5] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for content filtering. In *ANCS '05: Proceedings of the 2005 symposium on Architecture for networking and communications systems*, pages 183–192, New York, NY, USA, 2005. ACM Press.

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

[7] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE*, pages 14–23, 1998.

[8] 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.

[9] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 419–430, New York, NY, USA, 2003. ACM Press.

[10] J. Kwon, P. Rao, B. Moon, and S. Lee. FiST: scalable XML document filtering by sequencing twig patterns. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 217–228. VLDB Endowment, 2005.

[11] M. Onizuka. Light-weight xPath processing of XML stream with deterministic automata. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 342–349, New York, NY, USA, 2003. ACM Press.

[12] L. Salmela, J. Tarhio, and J. Kytöjoki. Multipattern string matching with q-grams. *J. Exp. Algorithmics*, 11:1.1, 2006.

[13] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using XML. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 160–173, New York, NY, USA, 2001. ACM Press.

[14] E. Soisalon-Soininen and T. Ylönen. On Classification of Strings. In *SPIRE*, pages 321–330, 2004.

[15] D. Suciu. XMLData Repository – The Database Research Group of University of Washington, 2006. `http://www.cs.washington.edu/research/xmldatasets/`.