SFilter: A Simple and Scalable Filter for XML Streams

Abdul Nizar M., G. Suresh Babu, P. Sreenivasa Kumar

Indian Institute of Technology Madras
Chennai - 600 036
INDIA
nizar@cse.iitm.ac.in, sureshbabuau@gmail.com, psk@cse.iitm.ac.in

Abstract

XML stream querying problem involves evaluating a given, potentially large, set of query expressions on a continuous stream of XML messages. Since the messages arrive continuously, it is essential that the query processing rate matches the data arrival rate. Therefore, it is necessary to index the given set of query expressions appropriately to enable real-time processing of the streaming XML data. In this paper we propose a simple and scalable system for the XML stream querying problem. The system indexes the queries compactly using a query guide and uses simple integer stacks to efficiently process the stream. Our experiments demonstrate that the new system outperforms the classical stream query processor YFilter by sizeable margins without asking for more index space. Also, the system shows good time and space scalability with respect to query workload and stream size.

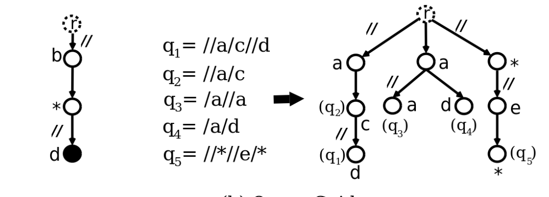
1 Introduction

As XML(eXtensible Markup Language) became a widely accepted standard for exchange of information over the internet, XML stream processing systems gained major attention of researchers for many years. Several applications of querying XML streams have emerged recently like selective dissemination of information, subscribing to real time news, monitoring stock market data *etc.* In these applications, the user is interested in data satisfying certain structural and value constraints. The user is required to be notified when an event satisfying the given constraints occurs. The event occurances are available in the form of an XML stream, quite often. And the user interest can be expressed as XPath[3] or XQuery[4] queries.
An XML document has a completely nested structure and modelled as an ordered, node-labelled tree. A path expression, which is the basic building block of XPath and XQuery, is used to navigate through an XML document tree and select nodes. For instance, the path expression *//section/title* selects titles of sections in an XML document representing, say, a journal paper or book while the path expression *//section[name = "Motivation"]/figure* returns all *figure* elements present in the section titled *Motivation.* Such path expressions consisting of *child* ('/') and *descendant* ('//') axes and predicates ('[...]') are conventionally represented using tree structures known as *twigs* while expressions without predicates are represented using linear paths known as *path trees.*
In this paper we propose a simple, scalable stream query processor, SFilter, that can effectively process large number of path queries with *child* and *descendant* axes against XML streams. The algorithm avoids the need for backtracking and is scalable in time and run-time space with respect to the query workload and stream size and has lower index space demands.

15th International Conference on Management of Data
COMAD 2009, Mysore, India, December 9–12, 2009
©Computer Society of India, 2009

2 Background and Related Work

We use the conventional well-formed XML message model, where each message in the stream is an ordered tree of elements. The beginning of each element e is marked with a *start-tag* and its end is marked with an *end-tag*; all the descendant elements of e have their start and end tags between the two tags of e leading to a nested structure.
In this paper we consider the fragment of XPath which includes *child*('/') and *descendant*('//') axes and wildcards('*') and does not include predicates. The queries represented by this XPath fragment are called *path queries.* For instance, *//section/title* is a path query with two axis steps while */book/*/fname* has three steps.
A path query can be modelled as a tree consisting of a linear path called *path tree.* Figure 1(a) shows the path tree representation of the path query `/b//*/d`. In the figure, edges without labels (parent-child (P-C) edges) represent *child* axes while edges with '//' as label (ancestor-descendant (A-D) edges) represent *descendant* axes. There is a dummy root node labelled `r` which is used to distinguish between trees representing



(a)Query //b/c//d (b) Query Guide

Figure 1: Path Tree and Query Guide

expressions starting with '/' and those starting with '//'. Node d shown in black is the result node.

Match of a query tree against an XML document is formally defined as follows:

Definition 1. *Given a path tree t and a document D, a match μ of t in D is a mapping from nodes of t to the nodes of D respecting the following constraints: (i) A node with label l in t is mapped to a node with label l in D (ii) A Node with wildcard ('*') label maps to any node in D (iii) The P-C and A-D relationships between nodes in t are satisfied by the mapped nodes.*

Note that, each match according to Definition 1 is a structural match. However, XPath's evaluation model assumes navigational semantics and produces unique document nodes matching the result node alone. To be consistent with this XPath evaluation model, we can define an evaluation function ϵ for path trees as follows:

Definition 2. *Let t be a path tree and s be its result node. Let $\{\mu_1, \mu_2, \ldots, \mu_n\}$ be the set of matches of t against some document tree D. Then $\epsilon(t,D) = \bigcup_{i=1}^{n}\{\mu_i(s)\}$*

We address the following stream querying problem in this paper.

> *Given a an XML stream D and a set of n path queries $Q = \{q_1, q_2, \ldots, q_n\}$ with path tree representations $T = \{t_1, t_2, \ldots, t_n\}$, for all $t_i, 1 \leq i \leq n$, return $\epsilon(t_i,D)$.*

2.1 Related Work

An XML filtering system, identifies if the query has *at least* one match in the document and routes the document to the user. An XML querying system, on the other hand, retrieves *all* the document fragments in the stream satisfying the query. Streaming algorithms read data sequentially and only once. At anytime only a small subset of the data, whose size of this subset is proportional to maximum depth of the tree model of the stream, is kept in memory.

Many approaches have been proposed for the XML stream querying problem. The approaches which are close to our work are XFilter [8], YFilter [11] XTrie [10], XPush [12] and AFilter [9]. All these approaches solve the stream querying problem in which large collection of query expressions are indexed to find match against a streaming XML document. All these approaches are either automaton-based or stack-based.

3 Query Representation

We construct a query guide to represent all the path trees representing path queries. It is an ordered tree representation of all the path trees that exploits the pre-fix commonality between the path trees such that (i) Root of G is the same as the dummy root 'r' of the path trees and (ii) The root-to-result node path of each path tree appears in G as a path that starts at node 'r' and ends at a descendant node and the path has the same node labels and edge constraints (i.e., P-C or A-D edge) of the path tree. Figure 1(b) shows a the query guide for the given set of path expressions. Note that the path tree for each path query is completely represented by a path in the query guide starting at the root node r. For instance, the path tree r–//–a–c–//–d for query q_1 is represented using the left-most path of the query guide.

The query id of a query is attached to the the query guide node representing the result node of path tree representing the query. For instance, in Figure 1(b), the query id q_1 is attached to the node labelled c as it represents the result node of the path tree r–//–a–c–//–d representing query q_1. The root of the query guide is the dummy root node r.

4 Query Matching

Our basic approach is to process the streaming XML document one tag at a time using the query guide representing the given path queries. At any time during execution, the algorithm maintains a sequence of elements S in the stream whose open-tags have been seen but close-tags are yet to arrive. It may be noted that, in the tree model of XML stream, this sequence of elements represents a path in the tree from root of the tree to some descendant node.

The algorithm maintains an integer stack at every query guide node to keep track of the current sequence of tags S in the stream. Each value in the stack represents the depth of an element in the stream that matches with the query guide node to which the stack is associated. Note that this number can uniquely identify a node in the stream as there will be exactly one node at a given depth in the current (or active) path in the document tree, represented by S.

The input XML stream is first parsed by a SAX parser that generates a stream of SAX events, which is input to the query processor. The algorithm starts by pushing a depth value 0 into the stack for the root node r of the query guide. It then proceeds by responding to the *open-tag* and *close-tag* events generated by the SAX parser.

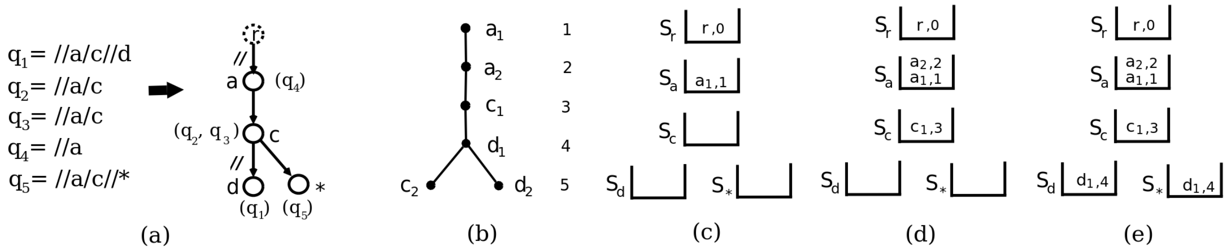


Figure 2: Illustrating Query Matching

4.1 Event Handlers

Abstractions of *open-tag* event processing and *close-tag* event processing are shown as Algorithms 1 and 2, respectively.

The intuitive idea behind open tag handling is as follows: Let G_p be the parent of node G_j in the query guide. The depth value (level number) of an element e in the stream is pushed to $G_j.stack$ when the element represented by the top frame in $G_p.stack$ and e satisfy the relationship between nodes G_p and G_j (Algorithm 1, lines 4–7). This condition ensures that when a value representing a document node is pushed into the stack of G_j, there is a chain of values in the stacks of nodes in the path from r to G_j, satisfying the A-D and P-C relationships specified along the path. This avoids unnecessary pushing of a lot of frames and lightens the 'work load' of *close-tag* event handler, leading to algorithm speed up. For the same reason, the newly inserted element's id value can be output as result for all the query ids attached to node G_j (line 9).

For the close-tag of an element e with close-tag

Algorithm 1: Open-Tag Event Handler

```
1 Open-Tag-Handler($e$) : $e$ has open-tag $\langle x \rangle$
2 Let $G_j$ be the query guide node with label $x$
3 Let $G_p$ be the parent of $G_j$
4 if $G_p$.Stack is empty then exit
5 if there is a P-C edge from $G_p$ to $G_j$ then
6     if the difference between the top element of
      $G_p$.stack and e.depth is >1 then exit
7 Push e.depth to $G_j$.Stack
8 foreach $q_i \in G_j.qIdList$ do
9     Output($\langle q_i, e_{id} \rangle$)
```

$</x>$ and depth d, the close-tag handler simply removes the topmost depth value from the stack of the query guide node whose label is x provided the depth value is equal to d (Algorithm 2, lines 2–3).

Algorithm 2: Close-Tag Event Handler

```
1 Close-Tag-Handler($e$) : $e$ has close-tag $\langle /x \rangle$
2 if $G_j$.Stack is empty or top element of $G_j$.Stack
  $\neq$ e.depth then exit
3 Pop $G_j$.stack
```

Note that YFilter may visit each NFA state multiple times and possibly insert them to the run-time stack during open-tag processing. Thus there will be an exponential blow-up of the number of active states, particularly for recursive and deep documents. Also, this can lead to the time overhead during close-tag event processing to restore the system to the state it was before the corresponding open-tag event was processed. SFilter removes these overheads by maintaining separate stacks of depth values at query guide nodes and using them to effectively take care of relationship constraints between the nodes. This approach avoids processing of elements which are not contributing to result evaluation, leading to notable performance gain. This observation is corroborated by the results of the experiments in the next Section.

Example 1. *Figure 2 illustrates execution of the algorithm. Figure 2(c)–(d) show some snapshots of stacks associated with query guide nodes during the evaluation of the queries in Figure 2(a) against the document in Figure 2(b). Depth values of document nodes are shown to the right of the document tree in Figure 2(b). Document nodes are subscripted with numbers for easy reference. In the stacks, the node label is also shown along with depth value for easy visual identification.*

Figure 2(c) shows the stack contents after the open-tag event for node a_1 has occurred. At this point the a_1 forms a path satisfying the A-D edge between node r and node a. Since q_4 is associated with node a, $\langle q_4, a_1 \rangle$ can be produced as output. similarly, $\langle q_4, a_2 \rangle$ can be produced as output (not shown in the figure).

When open-tag of c_1 is seen, its depth value is pushed to S_c (Figure 2(d)). Since contents of stacks S_r, S_a and S_c form a path that satisfies the relationship constraints in the path r–//–a–c of the query guide, $\langle q_2, c_1 \rangle$ and $\langle q_3, c_1 \rangle$ can be produced as output.

At the open-tag of d_1, $\langle q_1, d_1 \rangle$ and $\langle q_5, d_1 \rangle$ are produced as output (Figure 2(e)). Similarly $\langle q_1, d_2 \rangle$, $\langle q_5, c_2 \rangle$ and $\langle q_5, d_2 \rangle$ are also produced as output.

Correctness: It can be seen that, before pushing the depth value of a node in the document tree to the stack of a query guide node G_j, the algorithm ensures that there exists at least one sequence of document nodes represented by the depth values in stacks satisfying the constraints specified along the path from the

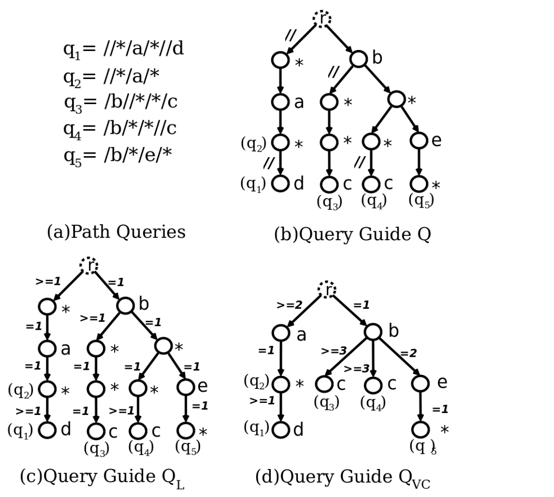


Figure 3: Illustrating Vertical Compression

root r to G_j (lines 4–7 of Algorithm 1). Hence the document node with the newly added depth value can be decalred as result for each query whose id is associated with node G_j.

4.2 Optimization

One problem with the basic query guide and the corresponding algorithmic approach discussed above is the overhead associated with wildcard node processing. Note that, since wildcard matches any tag, the query guide nodes with wild card label are to be processed for *every* element in the stream. This overhead can be partly overcome by what we call the *vertical compression* of the query guide and slight modification of the event processors.

Figure 3 illustrates vertical compression of query guide. The query guide Q in Figure 3(b) can be equivalently represented using query guide Q_L in Figure 3(b) where the query edges are labelled with *expected depth*. An edge between a node b and and its parent a in the query guide with an expected depth '≥1' indicates that the document nodes x and y matching with query nodes a and b should be such that the depth of y is *at least* one more than the depth of x while the edge label '=1' indicates that depth of y is *exactly* one more than depth of x. Obviously, '≥1' and '=1' have the same interpretation as A-D and P-C edges, respectively.

In vertical compression, we vertically collapse paths in the query guide by eliminating wildcard nodes. Consider a path $a \rightsquigarrow * \rightsquigarrow b$ in the query guide. We can collapse this path into a path without the wild card node. While doing so, the expected depth labels in paths a⇝b and $* \rightsquigarrow b$ are combined. For instance, when collapsing the wildcard node between r and a in the leftmost path from r in the queryguide Q_L of Figure 3(c), the expected depths ≥1 and =1 are combined to form ≥2 (see Figure 3(d)). As a is expected to be exactly one level below * and * should be at least one level below r, after collapse, a should be at least two levels below r. We can similarly collapse other and P-C and A-D edge combinations. Note that a wildcard node can not be eliminated if it holds query ids as the the results identified for those query ids are output at that node only.

Snippet 3 shows modification needed to the open-tag even handler (Algorithm 1) to process the compressed query guide against the stream. The snippet uses two abstract functions:- (i) *existsGe*(d, l, G_p): Checks if there exists a depth value v in the stack of G_p such that the difference between the depth value d of the current node being processed and v is greater than or equal to l. (ii) *existsEq*(d, l, G_p): checks if their exists a depth value v in the stack of G_p such that $d - v$ is equal to l. This condition can be checked by examine *atmost i* values starting with the top value in the stack.

Snippet 1: Extending Open-Tag Handler

```
   -- Replace Lines 5-6 of Algorithm 1 --
switch (exp. depth $d_e$ between $G_p$ and $G_j$) do
    case '$\geq l$'
        if !existsGe(e.depth, $l$, $G_p$) then exit;
    case '$= l$'
        if !existsEq(e.depth, $l$, $G_p$) then exit;
```

5 Experimental Evaluation

In this section we compare performance of SFilter with YFilter[11]. Java implementation of the YFilter is publicly available [5]. SFilter was also implemented in Java. Xerces SAX parser [2] was used to parse the XML documents. We conducted all the experiments on a 2 GHz Core2 Duo machine with 2GB memory running Linux. Java virtual machine (JVM) version 1.5 was used for conducting the tests. To make the comparisons uniform, we excluded the time needed for query pre-processing, stream parsing and result output from execution time. Table 1 shows notational conventions used in this section.

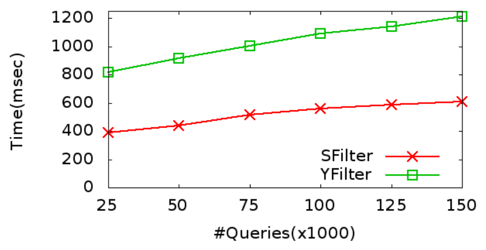
The experiments were conducted using NITF [1] dataset, which is is used to represent news articles in XML format and was developed by the International Press Telecommunications Council.

s	Stream size
q	Number of queries
d	Maximum query depth
$p(*)$	Probability of Wildcard
$p(//)$	Probability of //-axis
w-x-y-z	$w\times$1000 queries with max. depth x, $p(*)$=*0.y* and $p(//)$=*0.z*

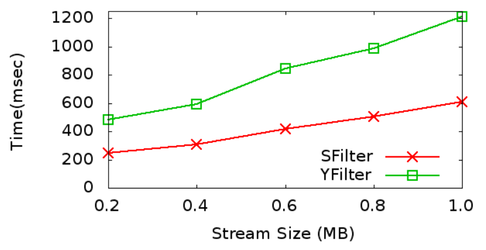
Table 1: Notations Used

Scalability

We compared scalability of SFilter with YFilter using query workloads of size 25K to 150K in steps of



(a) Queries vs Time (s=1MB)



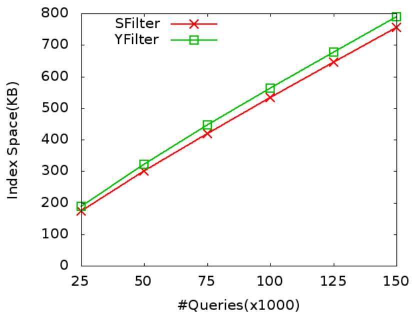
(b) Stream Size vs Time (q=150K)

Figure 4: Scalability

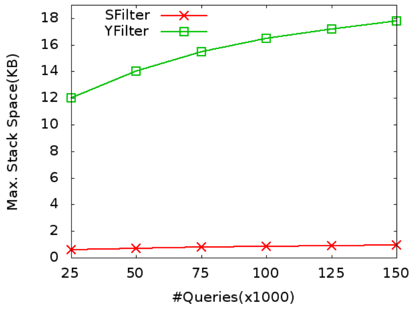
25K with $d = 5$, $p(*)$=$p(//) = 0.2$. The stream size was varied from 0.2MB to 1.0MB in steps of 0.2MB. The results are shown in Figure 4. Figure 4(a) shows scalability with respect to query workload for stream size of 1.0MB while Figure 4(b) shows scalability with respect to stream size for 150K queries. In all cases, SFilter performs better than YFilter. As mentioned earlier, the automata-based YFilter needs to visit and cache NFA states multiple times to the run-time store. This can also lead to time overhead during close-tag event processing to restore the system state.

Space Requirement

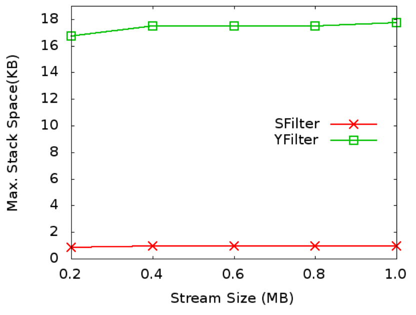
Graphs in Figure 5 compare the static and dynamic memory requirements of SFilter and YFilter. Static requirement involves the space needed for indexing the queries in main memory. In case of SFilter this is the space occupied by the query guide while for YFilter it is the space needed to hold the NFA structure. During run-time, YFilter uses a single stack where stack frame holds all the current active states in the NFA. SFilter uses integer stacks at every node of the query guide to keep track of depth of nodes in the tree model of the the stream.
In the experiment we used query work loads of size 25K to 150K with $d = 5$, $p(*) = 0.2$ and $p(//) = 0.2$. The index space needs of the two systems are comparable and SFilter performs better as the query workload size increases (See Figure 5(a)). The stack space requirement of SFilter is much less compared to YFilter and the former scales very well (Figure 5(b)). Note that YFilter has to cache large number of active states to the runtime stack. On the other hand, the number of depth values cached to the stacks of query guide nodes is proportional to the maximum depth of the document tree being streamed.
We also checked how the maximum stackspace need changes with changing stream size (Figure 5(c)). The stream size was varied from 0.2 MB to 1.0 MB. The query workload was 150K with $d = 5$, $p(*) = 0.2$ and $p(//) = 0.2$. In both the systems the stack space demand does not change considerably with increasing document size.



(a) Queries vs Index Space



(b) #Queries vs Stack Space (s=1MB)

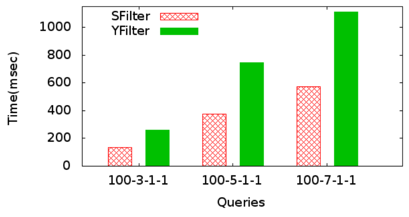


(c) Stream Size vs Stack Space (q=150K)

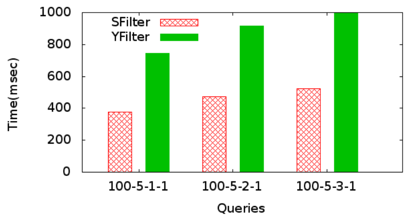
Figure 5: Space Scalability

Performance with varying query depth, p(*) and p(//)

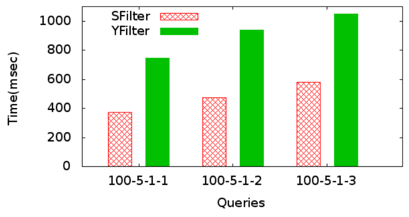
We also compared performance of SFilter and YFilter with query workloads that vary in maximum query depth, probability of wildcard and probability of *descendant* axis. The query workload size was kept as 100K in all the cases. Figure 6 shows the results (recall from Table 1 that *w-x-y-z* represents query work load of $w\times$ 1000 queries with maximum depth $(d) = x$, $p(*) = 0.y$ and $p(//) = 0.z$). In all the cases SFilter shows consistent performance gain. One of the



(a) d=3,5,7 $p(*)$=0.1,$p(//)$=0.1)



(b) d=5,$p(*)$=0.1,0.2,0.3,$p(//)$=1



(c) d=5,$p(*)$=0.1,$p(//)$=0.1,0.2,0.3)

Figure 6: Effect of d, $p(*)$ and $p(//)$

systems that achieves performance similar to the proposed system is AFilter [9]. The experiments reported in [9] indicate that the maximum stream size tested using AFilter is only 6K whereas we have run our system on 100K streams. It is unclear how the AFilter system would perform for large stream size. Also, from a close look of [9], it is clear that the approach is far more complicated than the one proposed in this paper.

6 Conclusion

In this paper, we proposed an algorithm for shared processing of path queries with child and descendant axes against XML streams. It was found that the proposed system, SFilter, is both efficient and scalable in time and space and outperforms currently available algorithm by sizeable margins without compromising on index space requirements. We are currently investigating how the current algorithm can be extended, using the representation scheme in [6, 7], to handle queries with other XPath axes and predicates.

References

[1] International Press Telecommunications Council. `http://www.iptc.org`.

[2] Xerces SAX Parser. Available at `http://xerces.apache.org/`.

[3] XPath Specification. `www.w3.org/TR/xpath`.

[4] XQuery Specification. `http://www.w3.org/TR/xquery/`.

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

[6] Abdul Nizar M. and P. Sreenivasa Kumar . Efficient Evaluation of Forward XPath Axes over XML Streams. In *14th International Conference on Management of Data (COMAD 2008)*, pages 217–228, 2008.

[7] Abdul Nizar M. and P. Sreenivasa Kumar . Ordered Backward XPathAxes Processing against XML Streams. In *6th International VLDB XML Database Symposium (XSym09)*, 2009.

[8] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64, 2000.

[9] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. Afilter: adaptable xml filtering with prefix-caching suffix-clustering. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 559–570. VLDB Endowment, 2006.

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

[11] 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.

[12] 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.