A New Approach to Filtering of XML Streaming Data

Preeti Saxena and Raj Kamal
School of Computer Science and Information Technology,
Devi Ahilya University, Indore 452001, India
preeti_ms@rediffmail.com

Abstract—Information processing and retrieval in many applications needs filtering of the XML streams. A stream-filter system examines queries on a continuous stream of XML documents and delivers matched content to the user. This paper proposes a new algorithm named PFilter for stream filtering systems. The PFilter processes a large amount of XPath query expressions to provide the desired XML nodes. PFilter processes the structure matching as well as value-based predicates in the expressions. It is a sequence-based algorithm. It converts the query expressions into sequences of nodes to provide efficient and fast search in the XML streaming document using an appropriate indexed structure. The main feature of the algorithm is that it reads the XML document once and does not require a post-filtering phase. Experiment as well as mathematical analysis shows that performance of PFilter is better compared to a state-of-the-art open-domain YFilter. The proposed algorithm can also be used in many applications requiring fast and selective XML data dissemination from stream of XML documents.

Index Terms—XML filtering, XPath query expression, SAX parser, publish-subscribe system

I. INTRODUCTION

XML [1] has been widely accepted as a language for data storage and retrieval in many applications. Its self-describing format represents structure and contents in an easily understandable format. Examples of such applications are selective dissemination of information such as stock and news updates, and traffic information. These are referred as publish-subscribe systems. Subscribers (Users) subscribe to the applications to get the information of their interest. Publishers (Service Providers) in response distribute these messages to the subscribers. The application-subscriber is required to be notified when an event satisfying their interest occurs. These applications hence require XML stream filtering systems. The system continuously serves the subscribers.

XML filtering is a process of identifying a large number of sequence patterns in a stream of XML document. The user interests are prescribed in the form of the user’s profile in a system based on XML filtering. A user profile is expressed using XPath [2] Language that is a query language for selecting nodes from an XML document.

XPath provides the way to traverse the document as a tree and select the nodes. The important operators used in the language are parent-child operator (’/’), ancestor-descendant operator (’//’ ) and wildcard operator that matches any node name (’*’ ). The symbols [‘[‘ and ‘]’] are used to embed value-based predicate or branch of
Expressions consisting of these operators, nodes and text values are the path expressions [2]. /a/b/c and /a/b]/c are examples of path expressions. Expression /a/b/c selects c nodes that are children of b node that are children of a node. Here, node a is the outermost node of the document. Another expression /a/[b]/c selects b and c nodes that are children of outermost node a of the document. The former expression is an example of linear expression while the later is non-linear or branched expression.

The expressions are executed on streaming data to extract user interested data in a stream filtering systems. Thus filtering process renders the selective data as per user profile. Fig. 1 shows an XML stream filtering process.

An algorithm, named PFilter is proposed that provides an efficient matching of XML documents to large numbers of XPath expressions. PFilter uses a sequence-based approach to quickly traverse and identify related profiles for structure matching and content (value-based predicates) also. Sequences are grouped into a dynamic hash-based index for concurrent processing. This paper describes these structures and an event-based PFilter algorithm. It is found by experiments that the algorithm performs significantly better when compared to the state-of-the-art YFilter [3] system. PFilter avoids the need of post-filtering phase as it scans the XML document once to locate the desired path location. Filtering experiments show that PFilter performs well for any XML document that has linear or branched XPath expressions and value-based predicates as well.

The remainder of this paper is organized as follows. Section 2 provides the background and motivations of this work. Section 3 describes the proposed algorithm. Section 4 discusses the experimental results followed by conclusion in Section 5.

II. BACKGROUND AND MOTIVATIONS

The popularity of XML as a language for data exchange has opened several research possibilities in the area of efficient XML filtering systems. There are several approaches suggested in past for preparing such filtering systems. All these approaches solve the querying problem in which large amount of XPath expressions is indexed to find a suitable match in the XML documents. They are based on various XML filtering algorithms. These algorithms can be classified into automata-based, index-based, sequence-based algorithms and other.

The XML filtering algorithm XFilter [4] used single Finite State Machine (FSM) approach to quickly locate and examine relevant profiles. Subsequently, YFilter [3], [5] developed by Yanlei Diao and Michael J. Franklin was designed to handle large numbers of queries by exploiting commonality among all the queries. YFilter used NFA-based approach to combine all user profiles into a single machine. In XFilter the nested paths are extracted from the main path expressions and processed individually. A post-processing phase is used to link matched paths to find that an entire query expression has been matched. YFilter also follows a similar approach, using the NFA/post-processing interface.

An index-based structure called XTrie[6] was designed to provide filtering on complex XPath expressions. XPush [7] proposed the use of a modified deterministic pushdown automaton to process large numbers of XPath expressions with many predicates per query, on a stream of XML data. It has not shown support to update the XPath workload directly. AFilter[8] suggested scalable and high performance path expression filtering through prefix-caching and suffix-clustering. Afilter is tested over small stream size and seems to be complex at implementation level. KiST [9], the first sequence-based XML filtering system that represented structure matching of twig (tree structured XPath Expressions) patterns with incoming XML documents. Fist-
P [10], a sequence-based XML filtering system that processes the structure matching and also handles value-based predicates in the twig patterns. They both require two phases, first identifies a super set of twig patterns that potentially match an incoming document and second phase discards the false matches by performing post-processing.

HFilter [11] gave solution to process deep and recursive XML data. SFiIter [12] indexed the queries compactly using query guide and showed time and space scalable system. An efficient and scalable sequence-Based XML Filtering [13] is presented that suggests encoding for structure matching. Also, the results are reported for small stream size. Bloom Filters [14] introduced an approximate method for XML data filtering using Bloom filters for representing path queries. It takes less time to build routing table but the performance of Bloom filter-based filter system is not good when the depth of input XML packets is large. Most of the methods to solve the problem of matching an XPath query expression either require two parse of the XML document or requires implementing several data structures making algorithm complex at implementation level. The two times scanning of document makes the system slower and also unsuitable for streaming document applications.

A need was felt to build a filtering algorithm for XML streaming documents that is simple and can retrieve the matched node and content in a single parse of the document. The proposed algorithm, PFfilter is described in the next section. The objective of this work is to propose an extension in existing XML filtering algorithms possibly leading to faster response time and scalable to support increasing number of users seeking information of their interest. The section also describes the sequential construction needed to process an XML document as well as query expression.

III. PROPOSED ALGORITHM: PFILTER

XML document has a hierarchical structure. It is represented using tree structures with element nodes and relationships between the nodes. Consider the following XML document fragment.

```
<a>
  <b>
    <d> .. </d>
    <e> .. </e>
  </b>
  <c>
    <d> .. </d>
    <f> .. </f>
  </c>
</a>
```

Fig. 2 illustrates the tree representation of this fragment.

Fig. 2. Tree Representation of XML document fragment

It is observed that an expression written in XPath language also has a hierarchical structure. Fig. 3 shows tree representations of two XPath expressions /a/b/c and /a[b]/c.

```
XPath Expression : /a/b/c  XPath Expression : /a[b]/c
```

Fig. 3. Tree Representation of two XPath Expressions
The filtering process of an XPath expression with an XML document can thus be exploited using the hierarchical property of both the format. XPath expressions can be treated as a subtree mapped in any XML document. The expression /a/b/c has no match in the tree represented in Fig. 2 since b and c nodes are siblings (two nodes that are children of same node) and not the parent and child. The expression /a[b]/c is matched since a is parent of b and c which are siblings. The matching guide of this expression is shown in Fig. 4. The dashed lines represent the matching outline.

![Fig. 4. XML Document and Expression/a[b]/c: Matching Guide](image)

The two basic inputs of an XML filtering system are the user profiles (user queries in the form of XPath expressions) and the streaming XML documents.

PFilter is a sequence-based XML filtering algorithm. It filters XML documents over the XPath expressions. It generates the matched data in output. It handles thousand or more of user profiles in form of XPath expressions. Also, the construction of sequences is deployed that represents the queries for further processing. It implements the idea of encoding XPath query expressions into sequences termed as value-based sequences (VBS). A QueryParser (described in Section 3.1) module is written that converts the XPath expressions into VBS sequences. The incoming XML document is processed using event-based SAX parser [15] that handles events like the open-tag and the close-tag for the start and end of an XML node respectively.

A. Construction of Sequence: Converting user profiles into query sequence

To generate the VBS sequence for the XPath query expressions, they are treated as ordered tree. Pre-order traversing of this tree is done to obtain a query sequence of the form open-tag (On), value-based predicates (Vn). When the leaf node of the expression is processed, reverse the sequence so as to name close-tag (Tn). In this way prepare a sequence with On, Vn and Tn symbols. Consider for example a query expression /a/b/[c=5].

It can be represented as tree shown in Fig. 5. The nodes of the tree are traversed in preorder to obtain Oa Ob Oc V5 and the process is reversed to obtain Tc Tb Ta. Thus the VBS sequence will be Oa Ob Oc V5 Tb Ta.

![Fig. 5. Tree representation of query sequence /a/b/[c=5]](image)

The wild cards are also handled similarly. The child operator ('/') and descendant operator ('//') in the expression are treated as regular tree edges. This process continues till all query expressions are processed and converted into specified form.

Each VBS sequence is uniquely identified by its query sequence number (query Id) and each element of the VBS sequence is termed as query sequence node or node in short. Thus Oa and Ta are query sequence nodes. Also document nodes can be represented as ‘Oa Vb Ta’ for ‘<a> b </a>’ code statement. The preorder number (initial value taken as zero) of each query sequence node is also maintained along with node. These sequences are stored in an input file (text file) that serves as input to the PFilter. The next sub-section describes the method of query structure matching.
B. Structure Matching

The incoming XML document is represented as a sequence of open, close and value tags similar to query sequence construction explained in the above section. The ‘[’ and ‘]’ symbols are used to represent value-based predicates as well as document branch. The VBS sequence (T) of the XML fragment shown in Fig. 2 will be:

\[ \text{Oa Ob Od Td Oe Tb Oc Od Td Of Tf Tc Ta} \]

Now, consider an XPath expression Q1 - /a/b[d]/f (Fig. 6), its VBS Sequence will be (S1):

\[ \text{Oa Ob Od Td Of Tf Tb Ta} \]

While matching VBS (S1) with T, it is found that the close-tag (b) in T occurred before open-tag (f) and in S1 their positions are opposite, leading to unsuccessful match.

While matching VBS (S1) with T, it is found that the close-tag (b) in T occurred before open-tag (f) and in S1 their positions are opposite, leading to unsuccessful match.

Consider another XPath expression (Q2) - /a[b/c]/d/f (Fig. 7). Now construct its VBS Sequence (S2): \( \text{Oa Ob Od Td Tb Oc Of Tf Tc Ta} \). While matching VBS (S2), it is reported as successful match. The Fig. 8 shows S2 sequence match using dashed line. It also represents that an XPath expression is embedded in a document as a subtree.

C. Content Matching

The handling of value-based predicates in XPath expression is done by treating it as an independent sequence node. Fig. 9 shows a query expression Q3 - /a[b[c="M1"][d="5"]]. Its VBS sequence will be (S3): \( \text{Oa Ob Oc VM1 Tc Od V5 Td Tb Ta} \). Sequence node referred as ‘value node’ is created for the value of the predicate in each query.
Fig. 10 shows a sample XML document and its tree representation. Its VBS sequence is (T) : *Oa Ob Oc Tb Od Oe V5 Te Tf Td Ta*. For an XPath expression Q4 = /a/b[c], VBS Sequence (S4) = *Oa Ob Tb Oe Tc Ta* is obtained. It shows that the open-tag (c) in T occurs before close-tag (b), while in S4 the open-tag (c) occurs after close-tag (b), leading to unsuccessful match.

For an XPath expression Q5 = /a/d[e=5] VBS Sequence (S5) = *Oa Od Oe V5 Te Ta* is obtained. Matching S5 leads to a successful match. Fig. 11 shows the XML tree and matching for two XPath expressions Q4 = /a/b[c] and Q5 = /a/d[e=5].

The value delivered in the *characters* procedure is checked against the ‘value node’ of the sequence Index in open-tag and close-tag handler. For this a predicate flag is maintained for each query. The query matching process will continue only if its respective predicate flag is found true. Next sub-section describes the data structures used and filtering algorithm.

### D. Data structures and Algorithms

PFilter maintains several data structures. A runtime global stack is defined for handling each XML node along with its pre-order number. The entries are pushed in this stack when the open-tag event occurs. A dynamic hash-table known as sequence index is maintained to index the query sequence nodes during the matching process. It uses the unique open-tag, value-tag and close-tag of all the query sequences as keys and for each key it maintains a list of queries to be matched. For each query sequence a unique stack is also maintained that stores the index of the matched open-tag in global stack.

The incoming XML document is parsed by a SAX parser. It generates event for start and end of each node. A variable preNum (initialized to 0) is used to maintain pre-order number of the document node. The open-tag handler pushes the node as well as its current preNum into the global stack and increments preNum. The *characters* event fetches the value present between an open-tag and close-tag. This value is matched with value-based predicate of each query maintained in sequence index where key is its value-tag. A query is discarded where this match fails. Otherwise, the matched nodes are also pushed onto their corresponding query stack. The sequence index is advanced for next node. The current position of the corresponding query is also incremented. The close-tag handler pops a node off the global stack.

**PFilter Algorithm (Using XML SAX Parser)**

```
/* Dynamic Hashtable where processed query expressions are stored concurrently (as sequences) */
Hashtable sequenceIndex;
Stack globalStack;  /* A runtime global stack */
int preNum = 0       /* Preorder Number for document nodes */
```
ArrayList queryStack      /* Dynamic Array of each query */
/* At the start, initialize sequenceIndex */
begin procedure init()
  read queries
  assign queryId’s to each query
  for each query sequence q
    insert each unique sequence node as key in sequenceIndex
    insert q's queryId in sequenceIndex[first node]
    set Current Position as zero for all queries
  end for
end Procedure

/* Read the value (=predicates) from the XML document */
Begin characters(value)
  strPredicate = value
end Procedure

/* Open Tag Handler */
begin procedure open-TagHandler (tag)
  Open-Tag = 'O' + tag
  /* push every node in global stack */
  globalStack.push(Open-Tag, preNum)

  /* Match Content Value-based Predicates */
  set vList = sequenceIndex [valuePredicate]
  for each q vList do /* to fetch every value node */
    if !strPredicate matches with value node
      continue to read next tag    /* Unsuccessful Predicate Match */
  end for

  /* scan sequenceIndex for matching queries */
  set processList = sequenceIndex [Open-Tag]

  for each q in processList do
    X: insert q's queryId in sequenceIndex[next node]
    if next node is value node
      repeat X
      push inx & currQueryPos in q’s queryStack

    /* inx represents the index of the tag in the
     globalStack */
    increment q's current position
  end for
  increment preNum
end Procedure

At the close-tag handler the preNum of the document tag and the top element of the corresponding query
stack are matched. Also match content value with value-based predicate of each query maintained in
sequence index where key is its value-tag.

/* Close Tag Handler */
begin procedure Close-tagHandler (tag)
Close-Tag = 'T' + tag
/* Match Content Value-based Predicates */
set vList = sequenceIndex [valuePredicate]

for each q vList do /* fetch every value node */
    if !strPredicate matches with value
        continue to read next tag /* Unsuccessful Predicate Match */
    end for

pop global stack to get preorder Number of the node to get preNum
/* Pop from global stack */
set processList = sequenceIndex [Close-Tag]
for each q in processList do
    get the Node from globalStack that matches the top element of q’s queryStack
    if Node. preNum = preNum then /* Match preorder Number */
        delete queryId of q from Close-Tag & Open-Tag
        pop the top element from q's queryStack
        increment q's current position
        read next node of q
        if next node is null then /*end of query sequence */
            query is found matched.
            else
                X: insert q's queryId in sequenceIndex[next node]
                if next node is value node
                    repeat X
                end for
        end if
    end if
end for

set processList = sequenceIndex[Open-Tag]
for each q in processList do
    get the Node from globalStack that matches the top element of q’s queryStack
    if Node.preNum = preNum then
        pop top element from q's queryStack
        set currQueryPos to previous matched Node Position
        remove all elements of vList
    end if
end for
end Procedure

The queries are discarded where the match fails. This ensures the matching of document-node with query-node. Sequence index is then updated by deleting the corresponding open-tag and close-tag entries. The subsequent query index is also popped. After this, the reading of the next node of the query sequence is performed. The next node if found as null, implies the query as a match. For unmatched nodes of the current processing queries, the backtracking is done in close-tag handler itself. This avoids the need of any post-processing phase.

IV. EXPERIMENTAL RESULTS

PFilter is implemented in Java. The experiments are conducted on 3.10 GHz Intel Core i5-2400 Processor with 2GB RAM running Windows 7. Java Virtual Machine (JVM) version 1.7.0 is used for conducting experiments. The algorithm is compared with YFilter[4]. YFilter package was implemented in Java and is available in the public domain. YFilter code is executed using Netbeans IDE 7.2 with JVM 1.7.0. The experiments are conducted using NITF (News Industry Text Format) DTD available with YFilter. The XPath query expressions are generated using the XPath generator of YFilter. A Query Parser module is prepared in Java to convert the XPath queries into required format.
Description of parameters used in the experiments is given in Table 1. In first experiment scalability is evaluated. The algorithm is compared varying the number of queries from 25k to 125k in steps of 25k with \(d = 6, \ p(*) = 0.1, \ p(//) = 0.1, \ p(PV) = 0.03\) and \(S = 1\ MB\). The result is shown in Fig. 12.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>Number of Queries</td>
</tr>
<tr>
<td>(S)</td>
<td>XML stream size</td>
</tr>
<tr>
<td>(d)</td>
<td>Maximum Query Depth</td>
</tr>
<tr>
<td>(p(*))</td>
<td>Probability of Wildcards</td>
</tr>
<tr>
<td>(p(//))</td>
<td>Probability of Descendant Operator</td>
</tr>
<tr>
<td>(p(PV))</td>
<td>Probability of Value-based Predicates</td>
</tr>
</tbody>
</table>

The XML document stream of 1 MB (500 document each of ~2KB) are taken for filtering. The filtering time grows consistently as the number of query expressions increases for both the filters. PFilter is significantly faster for 100K and 125K. For 125K query expressions PFilter is 18\% faster than YFilter for 1 MB of streaming data. This signifies that PFilter scales better than YFilter when the number of query expressions increases.

The experiments are also carried out for stream of ~2 MB (500 document each of 1 to 10 kB) and ~10MB (2500 document each of 1 to 10 kB). In these sets of experiments probability of wildcard operator and probability of descendant operator are also varied. The query depth is set to 6. Similar performance improvement in filtering time is seen for large set of streaming documents as well. It suggests that PFilter performs substantially better for streaming documents with different number of XPath operators and increasing number of queries. It is also observed that the filtering time difference is more for large number of queries (>100K).

For the next experiment (Fig. 13), the XML stream size is varied from 0.2 to 1.0 MB in steps of 0.2 with \(d = 6, \ p(*) = 0.1, \ p(//) = 0.1, \ p(PV) = 0.03\) and \(N = 50k\). Here, 100 to 500 XML documents (each of size ~2 KB) are used to carry out filtering.

The increase of YFilter’s filtering time depicts that the filtering cost of PFilter is less (Fig. 13). The gap is widened for stream size more than 0.8 MB. It represents the better performance of PFilter in terms of filtering cost for large streaming documents.

The experiments are also carried out for stream size varied from 2 to 10 MB in steps of 2MB. Here, 500 to 2500 XML documents (each of size between 1 kB to 10 kB) in steps of 500 are used.

The query depth is set to 6, probability of wildcard characters and probability of descendant operator both are taken as 0.1. The experimental results show performance improvement over YFilter for the large stream size as well. PFilter is comparable with the algorithms [10,12] suggesting scalability with stream size.

The results show that the algorithm filters faster than YFilter in both the cases when number of queries is increased or the stream size is more.
V. CONCLUSION

The filtering problem of XML streaming documents that filters the queries written in XPath language is investigated in this paper. A new sequence-based algorithm, PFilter is proposed for XML filtering. It converts the query expressions into sequences of nodes to provide efficient and fast search in the streaming XML document. The algorithm scans the XML document once to locate the desired path location. The algorithm process the structure matching as well as value-based predicates in the XPath expressions. Filtering experiment shows that PFilter performs well for any XML document that has linear or branched XPath expressions. PFilter is scalable with the increase in number of user queries. It has also shown improvement in filtering time for large XML stream size.

REFERENCES