Efficient and Scalable Sequence-Based XML Filtering

Mariam Salloum
University of California,
Riverside, CA 92521, USA
msalloum@cs.ucr.edu

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

ABSTRACT

The ubiquitous adoption of XML as the standard of data exchange over the web has led to increased interest in building efficient and scalable XML publish-subscribe (pub-sub) systems. The central function of an XML-based pub-sub system is to perform *XML filtering* efficiently, i.e. identify those XPath expressions that have a match in a streaming XML document. In this paper, we propose a new sequence-based approach, which transforms both XML documents and XPath twig expressions into Node Encoded Tree Sequences (NETS). In terms of this encoding, we provide a necessary and sufficient condition for an XPath twig to represent a match in a given XML document. The proposed filtering procedure is based on a new subsequence matching algorithm devised for NETS, which identifies the set of matched queries free of false positives with a single scan of the XML document. Extensive experimental results show that the NETS method outperforms previous XML filtering approaches.

1. INTRODUCTION

With the increased adoption of XML as the de facto standard for publishing and exchanging of information over the web, XML-based pub-sub systems have emerged. In a typical XML-based pub-sub system, users (subscribers) express their interests (profiles[1]) using XML query languages (such as XPath [3]), while publishers distribute their messages encoded as XML documents. Each published message is matched against user queries so that messages are only delivered to interested users. Several approaches have been proposed to solve the XML filtering problem; the main two categories are (i) Finite State Machine-based and (ii) sequence-based approaches.

Several Finite State Automata approaches have been proposed. An early work, XFilter [1], considered simple path profiles and proposed building an FSM for each distinct profile. The FSM states are then traversed, while XML tag events are generated by the parsing of the streaming document. YFilter [4] [5] was a successor of XFilter and proposed building a Non-Deterministic Finite Automata (NFA) representation that combines all user profiles into a single machine. This approach yields better results since it exploits the commonality among path expressions. In order to implement twig filtering, FSM-based approaches typically break twig queries into their simple linear paths. This approach however, requires an expensive post-processing phase to join the results.

In the sequence-based approaches [2][7], the XML document and profile twigs are transformed into sequences. FiST [7] was the first to propose a sequence-based XML filtering system using Prufer sequence encoding. This approach was shown to be more efficient than automata-based approaches since whole twig profiles are processed at once. Nevertheless, subsequence matching could create false positives and thus a post-processing phase is required to filter them. XFIS [2] is another Prufer-like sequence encoding of XML documents and profile twigs, however it does not account for tag recursion in the XML data.

In this paper, we propose a new sequence-based approach to solve the XML filtering problem. The key contributions of this paper can be summarized as follows:

- We present a new, simple and effective sequence representation for XML documents and query twig patterns, called **NETS** (**N**ode **E**ncoded **Tree S**equence).
- Using NETS, we present a necessary and sufficient condition for a twig profile to match a given XML document tree.
- We present a filtering system for ordered twig pattern matching that employs concurrent subsequence matching of query profiles. Our filtering approach is unique since it does not require a post-refinement phase. The approach guarantees that returned matches are *free* of false positives (and false negatives) with a *single* scan of the XML document.
- Experimental results show that the proposed approach outperforms previous XML filtering approaches. Performance improvements are achieved through holistic processing of twig patterns and on-the-fly detection of false matches.

We proceed with the description of the NETS encoding in Section 2. In Section 3 we provide the details of our filtering system. Section 4 presents experimental results while conclusions appear in Section 5.

2. NODE ENCODED TREE SEQUENCE

An XML document is modeled as a rooted ordered labeled tree where each node corresponds to an element tag, attribute, or value, and edges represent structural relationships between nodes. Several sequence representations for labeled trees have been proposed and utilized for XML filtering or query matching [8][10][11]. We present a simple yet efficient sequence representation for XML trees called Node Encoded Tree Sequence (NETS).

For each node in a given tree, the NETS of the tree contains two symbols referred to as 'start-symbol' and 'end-symbol.' For example, given a tree T with a node labeled x, the NETS(T) will contain the start-symbol 'Sx' and end-symbol 'Ex' for node labeled x. The Node Encoded Tree Sequence of T, NETS(T), is defined recursively as follows. An example of a XML tree T and its corresponding NETS is shown in Figure 1(a).

Definition 1 (Node Encoded Tree Sequence):

1. If tree T consists of a single node labeled by r, then the NETS(T) is **S**r **E**r.
2. Let r be the root of tree T and assume r has m children labeled from left-to-right as $r_1, r_2, \ldots, r_m$. Let sequences $S_1, S_2, \ldots, S_m$ be the NETS of the subtrees whose roots are $r_1, r_2, \ldots, r_m$, respectively. Then the NETS(T) is **S**r $S_1 S_2 \ldots S_m$ **E**r.

[1] In the paper, we shall use the terms query and profile interchangeably.

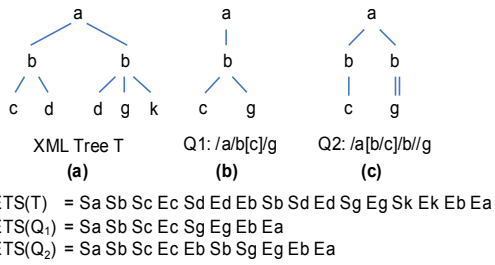

Figure 1: NETS Representation of XML Tree and Query Twigs

An XPath query expression can also be modeled as a labeled tree (referred to as a query twig) where each node represents an element or value, and edges denote parent-child ('/') or ancestor-descendant ('//') relationships (see Figure 1(b)(c)). The NETS of a query twig is generated in accordance to Definition 1. Note, when generating NETS of a twig query no distinction is made between parent-child and ancestor-descendant edges in the tree. In Section 3.1, we associate additional attributes to the NETS sequence nodes to encode '/', '//', and '*'.

We note that NETS has several properties. For a given label, x, the NETS must have an equal number of start-symbols (Sx) and end-symbols (Ex). For every two nodes in the tree with labels x and y, the corresponding segments Sx….Ex and Sy…Ey in the sequence are either disjoint or nested. The start and end symbols of the same node are called *corresponding symbols*. If a node is labeled by x, the preorder of the corresponding symbols Sx and Ex are defined as preorder(Sx)=preorder(Ex)=preorder(x). Hence, any two start and end symbols in the NETS are said to be corresponding symbols if they have the same preorder number.

The objective of the filtering algorithm is to determine whether a given query twig has a 'match' in a XML tree. For a query to be a match, the query twig must be a subgraph (Definition 2) of the XML tree and satisfy the 'level-consistent' property (Definition 3), which we formally define as follows:

Definition 2 (Subgraph): Let T=(V,E) be a labeled tree, where V is the set of all nodes in T and E is the set of all edges in T. For every node n in V, let *label(n)* denote the label of n and let *preorder(n)* denote the number associated with the node based on the tree preorder traversal. A labeled tree Q=(V',E') is a subgraph of T if the following two conditions hold: (1) There is a one-to-one mapping f() from V' into V such that for every node n in V' label(n)=label(f(n)), and for every edge (n_1,n_2) in E', there is an edge $(f(n_1),f(n_2))$ in E or there is a path from $f(n_1)$ to $f(n_2)$ in T. (2) For every two nodes n_1 and n_2 in Q, if $preorder(n_1) < preorder(n_2)$ then $preorder(f(n_1)) < preorder(f(n_2))$.

Definition 3 (Level-Consistent): Let Q be a query and subgraph of an XML tree T, and let *level(m)* denote the level of a node m in T. A query Q satisfies the 'level-consistent' property iff the following condition holds: For every edge (n_1,n_2) in Q, if the edge is of type '//', then $level(f(n_1))–level(f(n_2)) > 1$. Otherwise, if (n_1,n_2) is of type '/', then $level(f(n_1)) – level(f(n_2)) = 1$, where n_1 and n_2 denote nodes in Q and f is one-to-one mapping as described in Definition 2.

The filtering algorithm involves subsequence matching between the NETS sequences of the twig profile and the document, to determine if there is a match. The following

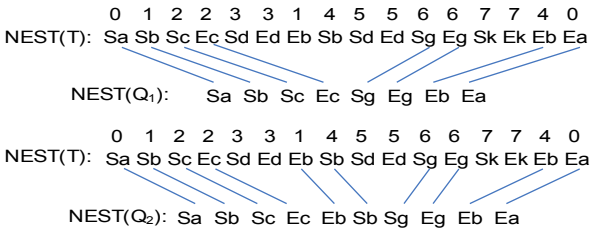

Figure 2: Q1 is a False Match and Q2 is a Match

theorem states the relation between the query twig and XML tree and their sequence representation.

Theorem 1: Given two rooted labeled trees T and Q, if Q has a match in T then the NETS(Q) is a subsequence of NETS(T).

Proof: Since Q has a match in T, Q is a subgraph and consequently there is a one-to-one mapping *f* from the nodes in V' into the nodes of V such that the label(n) = label(f(n)) for every node n in V' and satisfies other properties of Definition 2. Thus, each start-symbol and end-symbol in NETS(Q) must appear in NETS(T), but we need to prove that occurrence of these symbols appear in the same order in both NETS(Q) and NETS(T). Two cases are considered for every two nodes n_1 and n_2 in Q, whose corresponding start and end symbols are (Sx,Ex) and (Sy,Ey), respectively.

Case 1: There is a path from n_1 to n_2 in Q.

Thus, the segment Sx…Ex in NETS(Q) must contain the segment Sy…Ey such that the order of the symbols in NETS(Q) is Sx..Sy..Ey..Ex. Since $f(n_1)$ and $f(n_2)$ have the same symbols as that of n_1 and n_2 and there is a path from $f(n_1)$ and $f(n_2)$ in T, then the order of the start and end labels of $f(n_1)$ and $f(n_2)$ in NETS(T) is also Sx…Sy…Ey..Ex.

Case 2: No path exists between n_1 and n_2.

Since there is no path between nodes n_1 and n_2, then the segment Sx…Ex and Sy…Ey in NETS(Q) must be disjoint. Thus, the order of the symbols in NETS(Q) is either Sx…Ex…Sy…Ey or Sy…Sy…Sx…Ex depending on the preorder number of nodes n_1 and n_2. Since Q is a subgraph of T, the $preorder(n_1) < preorder(n_2)$ iff $preorder(f(n)) < preorder(f(n_2))$, consequently, the symbols in NETS(T) must appear in the same order.

Given trees Q and T, if we enumerate all possible subsequences of NETS(T) that match NETS(Q), then we are guaranteed to report all matches with no false dismissals. However, we note that the result may contain false positives. Consider the XML tree T and query twigs Q_1 and Q_2 shown in Figure 1. Figure 2 shows the NETS of T, Q_1 and Q_2. The $NETS(Q_1)$ and $NETS(Q_2)$ are both subsequences of the NETS(T). However, we note that Q_1 is a false match because it is not a subgraph of tree T. As we proceed, we shall prove a *necessary* and *sufficient* condition for a query to have a match in a given XML document.

Definition 4 (Tree Subsequence): Let T be a NETS labeled tree. A subsequence S of NETS(T) is called a tree subsequence if the following two conditions hold:

1. If Sx(Ex) is in S, then the corresponding symbol Ex(Sx) is also in S, i.e. if Sx(Ex) is in S then the Ex(Sx) with the same preorder number is also in S.
2. **S = Sr … Er,** where Sr and Er are corresponding symbols**.**

Reconsider the XML tree and query twigs shown in Figure 1.

The XML tree nodes are numbered with a preorder traversal of the tree so that each node has a unique number (See Figure 2). When NETS(Q_1) is matched with NETS(T), we note the matched subsequence is not a tree subsequence and hence Q_1 is not reported as a match. The matched subsequence for Q_2 is a tree subsequence and hence Q_2 is reported as a match.

Definition 5 (Level of Start and End Symbols): Given two rooted labeled trees Q and T, let T be a tree and x be a node in T. The Level(Sx) = Level(Ex) = Level(x), where Sx and Ex are the corresponding start and end labels of x.

Theorem 2: Given two rooted labeled trees Q and T, Q has a match in T iff there is a tree subsequence S of NETS(T) such that:

1. S = NETS(Q)
2. For every edge (n_1,n_2) in Q the following property holds:
   If (n_1,n_2) is of type '/', then the level(Sy) - level(Sx) = 1,
   If (n_1,n_2) is of type '//', then the level(Sy) - level(Sx) ≥ 1,
   where n_1 and n_2 are labeled tree nodes in Q and Sx and Sy are the corresponding start symbols in S.

Proof: *Necessary Condition* - Assume that query Q has a match in XML document T. By Theorem 1, NETS(Q) is a subsequence of NETS(T). Let S be a subsequence of NETS(T) that corresponds to NETS(Q). Since there is a one-to-one mapping f from the nodes of Q into the nodes of T that satisfies the properties stated in the definition of subgraph, then S satisfies the properties of tree subsequence. Also, since query Q has a match in T, then Q satisfies the level-consistent property. It is easy to show that the second condition of Theorem 2 follows from the level-consistent property.

Sufficient Condition - Let Q be a query such that there is a tree subsequence S of NETS(T) that satisfies the two conditions of Theorem 2. To prove that Q has a match in T, we first prove that Q is a subgraph of T, i.e. we define a 1-1 mapping f from the nodes of Q into the nodes of T, which satisfies the properties stated in the subgraph definition. Let n be a node of Q with label x. The mapping f(n) is defined as follows:

Since S=NETS(Q), the start and end symbols of node n must appear in S. As we progress, we will show that these two symbols in S are the start and symbols of the same node in T (i.e. they are not symbols of different nodes in T). This node of T will be denoted as f(n). Two cases need to be considered:

Case 1: There is only node of Q with label x.

Thus, tree subsequence S contains only one Sx and one Ex and they must be the start and end symbols of the same node in T, because they have the same preorder number.

Case 2: There is more than one node in Q with label x. Without loss of generality, we assume that there are only two nodes (n and m) of Q with label x and preorder(n) < preorder(m). Thus, the symbols of n and m in NETS(Q) must appear in the following order:

n n m m or n m m n

Sx … Ex … Sx … Ex or Sx … Ex … Sx … Ex

Since S=NETS(Q), these symbols also appear in S. Since S is a tree subsequence, then there are only two Sx and two Ex in S and they must appear in the same order as above. These symbols correspond to the two nodes in T denoted by T_m and T_n. We will prove by contradiction that the Sx and Ex of node n in Q correspond to either the Sx and Ex of T_m or the Sx and Ex of T_n. Suppose that Sx and Ex of n corresponds to the Sx of T_n and Ex of T_m, respectively as shown below:

T_n T_m T_m T_n or T_n T_m T_n T_m

Sx … Ex … Sx … Ex or Sx … Ex … Sx … Ex

The first case can be excluded because the sequence is not a valid NETS, since the Ex proceeds its corresponding Sx in the NETS. The second case can also be excluded because the property of NETS requires that the segment Sx….Ex and Sx…Ex of nodes T_m and T_n must either be disjoint or nested. In this case, the two segments overlap and thus it's not a possible NETS. Thus, Sx and Ex of n in Q correspond to the Sx and Ex of the same node f(n) in T. It is easy to verify that f(n) satisfies the two properties stated in the subgraph definition. Thus, Q is a subgraph of T. The second condition of Theorem 2 can be used to verify that Q satisfies the level consistency property; as a result, Q is a match of T.

3. FILTERING SYSTEM

In this section, we first describe how to encode wildcards, '/' and '//' for the query twig NETS. We proceed to describe the core data structures utilized and the filtering algorithm.

3.1 NETS Query Encoding and Data Structures

The NETS representation of query profiles is generated by encoding a start and end symbol for each node in the tree as presented in Definition 1. For each *start-symbol* in the NETS, an additional attribute referred to as 'relationship' is encoded to specify the relationship between a node and its parent node. For example, if the relation is a parent-child '/' then the attribute shall be '=1', specifying that two nodes must be one level apart. If the relation is an ancestor-descendant '//' then the attribute shall be '≥1', specifying that two nodes must be at least one level apart. Wildcards '*' are handled differently depending on the occurrence of the '*' in the query. If the wildcard operator appears as a branch node in the twig (See Q_1 in Figure 3), then the '*' is encoded as a regular node. Otherwise, if it is a non-branch node (See Q_2 in Figure 3), then the next non-wildcard node is encoded in the NETS, and the occurrence of the wildcard is reflected by the 'relationship' attribute. The encoded NETS of a query twig is referred to as query sequence.

Example 1: Consider the query twigs shown in Figure 3. Q_1 contains a branch wildcard node, thus the * is encoded as a regular node in the NETS(Q_1). Q_2 contains a non-branch wildcard node, thus, the * is not encoded as part of NETS(Q_2) and the next non-wildcard node is associated with relation'=3.'

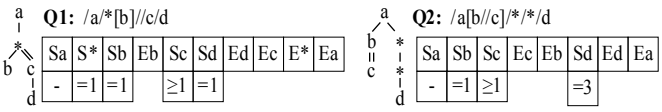

Figure 3: NETS Encoding of Query Twigs

The filtering algorithm utilizes several structures for subsequence matching, as illustrated by the example in Figure 4. A runtime global stack is maintained by the filtering algorithm where each entry in the stack is a tuple that contains a start-symbol of the XML sequence and its preorder and level. The tuples are pushed to the stack as start-symbols are generated. At the start of the filtering algorithm, concurrent subsequence searches are initiated over the query sequences. For non-recursive XML documents, only one subsequence search is needed for each query, however, for recursive XML, more than one subsequence search over the same query

sequence may be initiated. Each subsequence search maintains *queryPos* and *queryStack*. The *queryPos* is an integer that denotes the current position in the query sequence. The *queryStack* is a stack where each entry is an ordered pair. The first component in the stack is the index of the matched XML start-symbol in the global stack. The second component is the position of the matched start-symbol in the query sequence. The ordered pairs are pushed and popped to and from the queryStack as start and end symbols in the XML sequence are matched to the query sequence. The filtering algorithm also utilizes a dynamic hashtable, called *sequenceIndex*, to facilitate concurrent processing of query twigs. The *sequenceIndex* uses the start and end symbol assigned to each XML tag as a key into the hashtable. For each key (start or end symbol) it maintains a list of queries to be matched specified by their *queryIds*, the query's unique identification.

3.2 Filtering Algorithm

At the start of filtering, the first node of each query sequence is inserted into the *sequenceIndex*. The streaming XML document is parsed by the SAX parser; the *ProcessStartSymbol(.)* function is called when an open tag is generated and the *ProcessEndSymbol* is called when an end tag is generated. Note that the SAX methods have been slightly altered to maintain level and preorder information for each XML tag (see Algorithm 1).

Algorithm 1: XML SAX Parser – Filtering Algorithm

```
1   int level = -1
2   int preorder = -1
3   Stack globalStack;

    /* at the start of each new document, initialize sequenceIndex and
    queryPos */
4   procedure startDocument()
5     foreach query twig q do
      /* let nextSymbol denote the initial symbol in the query sequence */
6           sequenceIndex[nextSymbol].insert (q's queryId)
7           set queryPos to 0
8   end
    /* generate start-symbol, preorder and level of XML tag*/
9   procedure startElement (tag)
10    startSymbol = S + tag
11    level + = 1
12    preorder  + = 1
13    globalStack.push(startSymbol, preorder, level)
14    ProcessStartSymbol (startSymbol, preorder, level)
15  end
    /* generate end-symbol, preorder and level of XML tag */
16  procedure endElement (tag)
17    endSymbol = E + tag
18    preorder = pop global stack to get preorder of node
19    ProcessEndSymbol (endSymbol, preorder, level)
20    level - = 1
21  end
```

The *ProcessStartSymbol(.)* function is described by Algorithm 2a. This function receives a start-symbol and its preorder number and level as input. The filtering algorithm probes the sequenceIndex for a list of queries that match the startSymbol. For each query in the currentList, the algorithm verifies that the level-consistent property holds for the current query node (lines 4 – 5). The top entry of the queryStack is retrieved and the first component, referred to as *indexParAnc*, is used to retrieve the *xmlParAncNode* from the globalStack. The difference between the current startSymbol's level and the xmlParAncNode's level is calculated to determine whether the query's relationship attribute is satisfied. If the property is satisfied, then the following steps are performed. First, the index of the matched startSymbol (its index in the globalStack) and the queryPos are inserted into the queryStack. Second, the queryPos is incremented by one. Lastly, the next symbol in the query sequence is added to the sequenceIndex.

Algorithm 2a: Filtering Algorithm - Start Symbol Handing

```
1   procedure ProcessStartSymbol( startSymbol, preorder, level )

      /* probe sequenceIndex for matching queries */
2     currentList = sequenceIndex [ startSymbol ]
3     foreach q in currentList do

        /* Let 'indexParAnc' denote the first component of the top
        element in the queryStack' */
4       xmlParAncNode = globalStack . get ( indexParAnc )

        /* verify level-consistent property holds */
5       If xmlParAncNode.level–level satisfies query's
            relationship attribute then
          /* let 'nextSymbol' denote the next symbol in query sequence */
6         sequenceIndex[ nextSymbol ] . insert (q's queryId)
7
          /* let 'index' denote the index of the parameter startSymbol in the
          globalStack */
8         push index & queryPos onto queryStack
9         queryPos + = 1 / * advance query sequence position */
10    endfor
11  end
```

Algorithm 2b: Filtering Algorithm – End Symbol Handling

```
1   procedure ProcessEndSymbol( endSymbol, preorder, level)

      /* probe sequenceIndex for matching queries */
2     currentList = sequenceIndex [ endSymbol ]
3     foreach q in currentList do
        /* Let 'indexParAnc' denote the first component of the top element
        of the queryStack */
4     xmlParAncNode = globalStack . get(indexParAnc)

        /* verify preorder match s */
5     if xmlParAncNode . preorder = preorder then
          /* remove end and start symbol from sequenceIndex */
          /* let 'startSymbol' denote the symbol obtained by replacing 'E'
          with 'S' in the parameter endSymbol */
6     sequenceIndex[endSymbol].remove(q's queryId)
7     sequenceIndex[startSymbol].remove(q's queryId)
8     queryStack.pop()    /* pop the top element off the queryStack */
9     queryPos + = 1
          /* let 'nextSymbol' denote the next symbol in query sequence*/

10        if nextSymbol is null then /* end of query sequence */
11          report query as a match
12        else
13          sequenceIndex[nextSymbol].insert(q's queryId)
14    endfor
15
      /* handle query backtracking */
16    currentList = sequenceIndex[startSymbol]
17    foreach q in currentList do
        /* Let 'indexParAnc' denote the first component of the top element of
        the queryStack */
18      xmlParAncNode = globalStack . get(indexParAnc)

19      if xmlParAncNode.preorder = preorder then
          /* must backtrack query sequence position */
20        queryStack.pop() /*pop top element off the queryStack */
21        delete last inserted queryId in sequenceIndex
22        update queryPos
23    endfor
24  end
```

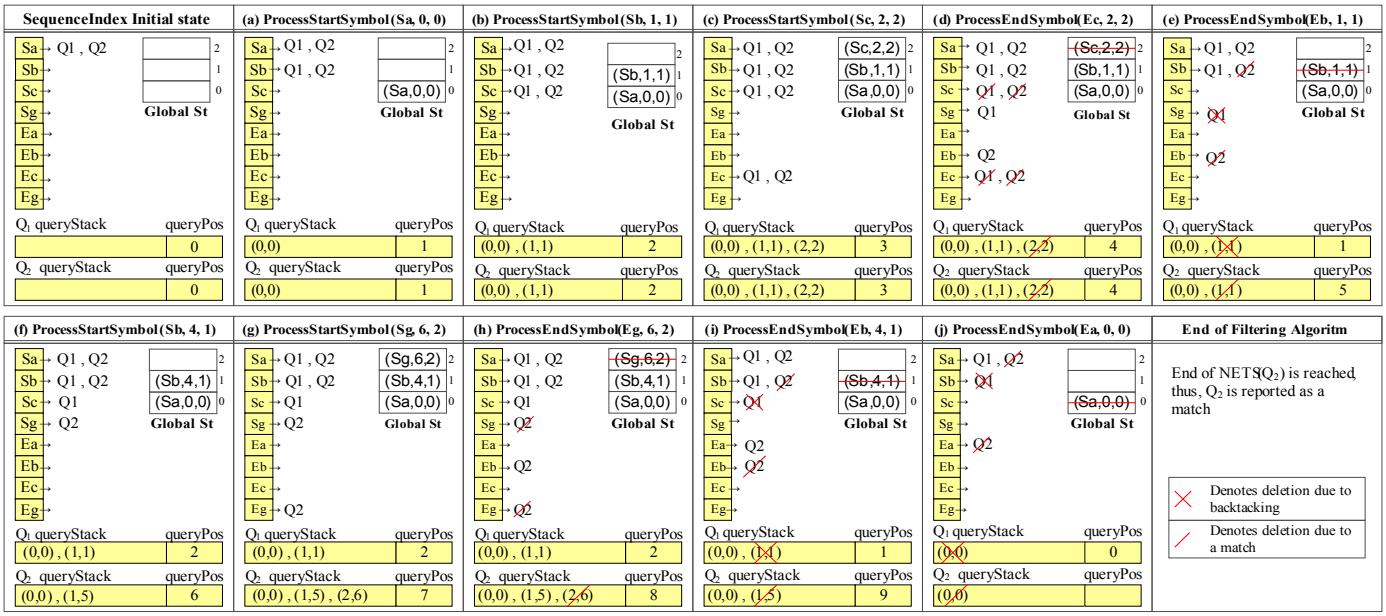

Figure 4: Filtering Algorithm Example

The ProcessEndSymbol(.) function is described by Algorithm 2b. For each query in the currentList, the filtering algorithm first verifies that the current endSymbol's preorder number equals that of the xmlParAncNode. If there is a match, then the queryId is deleted from the start-symbol's and end-symbol's lists in the sequenceIndex. If the end of the query sequence is reached, then the query is reported as a match. Otherwise, the next symbol is added to the sequenceIndex. At times, backtracking to a previous query sequence position is required due to a false match (lines 16-23). The sequenceIndex is probed for the start-symbol and a list of queryIds is retrieved. For each query in currentList, the top entry of the queryStack is retrieved and the first component, referred to as indexParAnc, is used to retrieve the xmlParAncNode from the globalStack. The xmlParAncNode's preorder is compared to the current endSymbol's preorder. If there is a preorder match, then backtracked is performed by the following steps. First, the top entry is popped off the queryStack. Second, the last inserted queryId is deleted from the sequenceIndex, and lastly the queryPos is updated to indicate the new position in the query sequence.

Below, we illustrate the execution of our filtering algorithm with the XML tree T and twig patterns Q1 and Q2 shown in Figure 1.

Example 2: In Figure 4(a), the ProcessStartSymbol(.) is invoked for 'Sa' and the sequenceIndex contains Q_1 and Q_2 for key 'Sa'. The 'level-consistent' property is automatically satisfied since the queryPos points to the first symbol in the query sequence. Thus, the subsequence search for both Q_1 and Q_2 is advanced. First, the next symbol in the query sequence ('Sb' for both Q_1 and Q_2) is retrieved and the queryIds are inserted into the sequenceIndex for that key. Second, the index of the startSymbol in the global stack (the index is '0') and the queryPos (the queryPos is '0') are inserted into the queryStack, thus the tuple (0,0) is pushed to the Q_1 and Q_2 queryStack. Lastly, the queryPos of Q_1 and Q_2 is incremented by 1. The algorithm proceeds to process XML nodes (Sb,1,1) and (Sc,2,2) in Figures 4(b) and 4(c).

In Figure 4(d), the ProcessEndSymbol(.) is invoked for 'Ec,' and the sequenceIndex contains Q_1 and Q_2 for key 'Ec'. The top ordered pair in the Q_1 and Q_2 queryStack is (2,2). The first component of the pair is used to retrieve the xmlParAncNode in the global stack, thus, (Sc, 2,2) is retrieved. The preorder of the current endSymbol and xmlParAncNode match, thus, the search for both Q_1 and Q_2 is advanced. The queryIds, Q_1 and Q_2, are deleted from the list of 'Sc' and 'Ec' in the sequenceIndex. The next symbol in the sequence of Q_1 and Q_2 is 'Sg' and 'Eb', respectively. The queryIds are inserted into the sequenceIndex for their corresponding keys. For both Q_1 and Q_2, the top entry is popped off the queryStack and the queryPos is incremented by 1.

In Figure 4(e), the ProcessEndSymbol(.) is invoked with end-label 'Eb'. Q_2 is retrieved and the preorder check is verified, thus Q_2's search is advanced. First, the top element is popped off Q_2's queryStack. Second, Q_2's queryPos is incremented by 1. Lastly, Q_2 is deleted from the sequenceIndex list for keys 'Sc' and 'Ec'. Note that backtracking of Q_1 occurs as well. After processing Q_2, the sequenceIndex is probed for the corresponding start-symbol 'Sb', and Q_1 is retrieved. The preorder check returns a match, thus, indicating that Q_1 should be backtracked. Thus, the Q_1 queryStack is popped, queryId Q_1 is deleted from the sequenceIndex list for key 'Sg,' and the Q_1's queryPos is backtracked to 1. In Figure 4 (f – i) the algorithm proceeds as described by Algorithm 2. In Figure 4(j), the last query sequence symbol of Q_2 is matched and thus Q_2 is reported as a match. Q_1, however, is not reported as match because the end of the query sequence is not reached. Please note (Sd,3,2), (Ed,3,2), (Sk,7,2) and (Ek,7,2) are not shown in the Figure 4 since the state of the structures does not change.

Example 2 shows the execution of the basic filtering algorithm for non-recursive XML document. If recursion occurs in the XML documents, multiple subsequence 'searches' may be initiated over each query sequence. The data structures and

the algorithm steps were slightly modified in the final implementation of the algorithm to process multiple searches for each query sequence. For each query search initiated, a queryStack and queryPos is maintained as before. The entries inserted into the sequenceIndex are extended to include the search ID, thus, for each key the sequenceIndex will contain a list of ordered pairs that contains the queryId and searchId. The ProcessStartSymbol(.) was slightly modified to initiate a new search when a recursive start-symbol is encountered. The filtering algorithm will first verify that the level-consistent property holds. If the level-consistent property is satisfied, then a new search is initiated and its corresponding queryStack and queryPos are updated to denote the current position of the search in the query sequence. The other operations of the filtering algorithm are maintained with the exception that multiple searches must be initiated as recursive elements are encountered.

4. EXPERIMENTAL RESULTS

In our experiments, we compared the performance of our system to that of YFilter[5] and FiST[7]. We utilized the YFilter Java implementation provided by the authors. We implemented FiST and NETS in Java as well. All experiments were performed on a Quad Core 2.66GHz processor with 8GB of memory running Linux Red Hat. We used the synthetic Sigmod [6] dataset for our experiments. We also generated twig patterns using the XPath generator available in the YFilter package. The element names were chosen from uniform distribution and the max depth of a twig pattern was fixed at 7. The number of branches in the twig patterns was fixed to 3, and the probability of '*' and '//' was fixed to 50 percent.

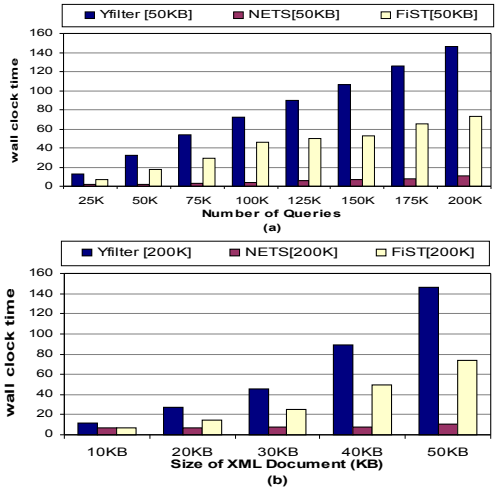

Figure 5: Filtering Time Experimental Results

Figure 5 compares the performance of our filtering system with that of YFilter and FiST. In Figure 5(a), the document size was fixed to 50KB and the numbers of twig queries were varied from 25,000 to 200,000 in steps of 25K. For 25K queries, all three methods perform comparably well, however, as the number of twig patterns were increased, the filtering time for FiST and YFilter increase dramatically. In Figure 5(b), the number of queries was fixed to 200,000 and the document size was varied from 10KB to 50KB. For 10KB case, all three methods yielded comparable results. The performance improvement of NETS was noticed as the document size was increased to 50KB.

The performance improvement achieved by the NETS approach is due to two reasons. First, our filtering approach employs holistic filtering of the query sequences, thus a match or a dismissal of a particular query twig can be made earlier in the filtering process. Second, our filtering approach filters false matches on-the-fly, thus avoiding an expensive post-processing phase. We conclude that our approach provides an efficient filtering algorithm that is scalable in terms of the number of user profiles and the size of the XML document.

5. CONCLUSION

We presented an XML-based filtering system that uses a new sequence encoding (NETS) for both the streaming documents and query profiles. Using NETS we provided a necessary and sufficient condition for a query profile to have a match in a given document. An important property of our system is that false positives and false dismissals are eliminated on-the-fly with a single scan of the streaming document. Experimental evaluation showed that NETS filtering provided performance improvements in comparison to state-of-the-art filtering systems. We plan to explore NETS encoding for the full-fledged case that reports the positions of all query matches in a streaming XML document. Moreover, we will explore the benefits of this encoding for traditional structural XML query processing.

6. REFERENCES

[1] Altinel, M., and Franklin M., Efficient Filtering of XML Documents for Selective Dissemination of Information. *In VLDB Journal*, pages 53-64, September 2000.

[2] Antonellis, P., and Makris, C., XFIS: An XML Filtering System based on String Representation and Matching. *International Journal on Web Engineering and Technology (IJWET)*. v.4, n.1, pages 70–94, 2008.

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

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

[5] Diao, Y., Altinel, M., Franklin, M.J., Zhang, H., and Fischer, P. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Trans. Database Syst.,* v.28, n.4, pages 467–516, 2003.

[6] Kwon, J., Rao, P., Moon, B., Lee, S. Value-based Predicate Filteirng of XML Documents. *Data and Knowledge Engineering*, v.67, n.1, pages 51-73, 2008.

[7] Kwon, J., Rao, P., Moon, B., and Lee, S. 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.

[8] Rao, P., Moon, B., Sequencing XML Data and Query Twigs for Fast Pattern Matching, *ACM Transactions on Database Systems (TODS),* v.31 n.1, pages 299-345, 2006.

[9] University of Washington XML Repository, 2002, http://www.cs.washington.edu/research/xmldatasets/

[10] Wang, H., and Meng, X., On the Sequencing of Tree Structures for XML Indexing. In *Proceedings of ICDE*, 2005.

[11] Wang, H., Park, S., Fan, W., And Yu, P., ViST: A dynamic index method for querying xml data by tree structures*, In Proceedings of the SIGMOD Conference*, pages 110–121, 2003.