FiST: Scalable XML Document Filtering by Sequencing Twig Patterns [*]

Joonho Kwon[*]   Praveen Rao[†]   Bongki Moon[†]   Sukho Lee[*]

[*]School of Electrical Engineering and Computer Science
Seoul National University
Seoul 151-742, Korea
joonho@db.snu.ac.kr   shlee@snu.ac.kr

[†]Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{rpraveen, bkmoon}@cs.arizona.edu

Abstract

In recent years, publish-subscribe (pub-sub) systems based on XML document filtering have received much attention. In a typical pub-sub system, subscribed users specify their interest in profiles expressed in the XPath language, and each new content is matched against the user profiles so that the content is delivered to only the interested subscribers. As the number of subscribed users and their profiles can grow very large, the scalability of the system is critical to the success of pub-sub services. In this paper, we propose a novel scalable filtering system called **FiST** (**Fi**ltering by **S**equencing **T**wigs) that transforms twig patterns expressed in XPath and XML documents into sequences using Prüfer's method. As a consequence, instead of matching linear paths of twig patterns individually and merging the matches during post-processing, FiST performs *holistic matching* of twig patterns with incoming documents. FiST organizes the sequences into a dynamic hash based index for efficient filtering. We demonstrate that our *holistic matching* approach yields lower filtering cost and good scalability under various situations.

1   Introduction

The publish-subscribe (pub-sub) systems play an important role in e-commerce and Internet applications by enabling selective dissemination of information. In a typical pub-sub system, whenever new content is produced, it is selectively delivered to interested subscribers. They have enabled new services such as alerting and notification services for users interested in knowing about the latest products in the market, current affairs, stock price changes etc. on a variety of devices like mobile phones, PDAs and desktops. Such services necessitate the development of software systems that enable scalable and efficient matching of a large number of content against millions of user subscriptions.

Today we come across e-commerce sites such as `Priceline.com` and `Hotwire.com` that provide email notifications to subscribers about price changes and hot deals. A recent service by `Google.com` called *Google Alerts* provides email updates to users regarding relevant Google results. Users can choose to receive notifications by selecting a topic and providing a list of search keywords. Another interesting example is the stock quote tracking service provided by `Yahoo.com`. There is a growing use and demand for large-scale information dissemination systems.

The popularity of extensible markup language XML as a standard for information exchange has triggered several research efforts to build scalable XML filtering systems for information dissemination. In such a system, user profiles are typically expressed in the XPath language [4]. In this paper, we consider user profiles that can be represented as *twig patterns*. These twig patterns contain the `child` and `descendant` XPath axes. For example, a path expression given in XPath syntax

```
book[author//name="John"]/title
```

qualifies XML documents by specifying a twig pattern composed of four elements `book`, `author`, `name` and `title` in an XML document, and a value-based selection predicate `name="John"`. In the filtering system, each incoming XML document is examined against user profiles represented by XPath expressions. The XML document is sent to users whose profiles are matched. One of the key challenges in building such a system is to effectively

organize a large number of profiles in order to minimize the filtering cost and achieve good scalability.

It should be noted that the XML filtering problem is different from the problem of finding all occurrences of a twig pattern in an XML document. This is due to the reversal in the roles of twig patterns and XML documents. Essentially, the filtering problem that we address in this paper is stated as follows.

> *Given a set of XPath expressions, identify those XPath expressions that appear in a given XML document.*

XFilter [1] was one of the early work in XML filtering. XFilter handles simple XPath expressions by transforming each path expression into a single finite state machine. Subsequently, the YFilter system [19] was proposed that focused on shared path matching to improve the scalability of the filtering system. YFilter constructs a single non-deterministic finite automaton (NFA) for all the XPath expressions. YFilter supports twig patterns by first matching individual linear paths from root-to-leaf and then by performing post-processing to identify matching twig patterns. Consider a nested XPath expression `book[author//name]/title`. YFilter splits the pattern and indexes two linear path expressions in its NFA, namely `book/title` and `book/author//name`. The individual linear path matches are used during post-processing to identify twig pattern matches.

In this paper, we propose a novel filtering system called **FiST** (**Fi**ltering by **S**equencing **T**wigs) that performs *holistic matching* of twig patterns with each incoming XML document. The matching is *holistic* since FiST does not break a twig pattern into root-to-leaf paths. Rather the twig pattern is matched as a whole due to sequence transformation. Our system focuses on *ordered twig pattern matching*, which is essential for applications where the nodes in a twig pattern follow the document order in XML. For example, an XML data model was proposed by Bow *et al.* for representing interlinear text for linguistic applications, which is used to demonstrate various linguistic principles in different languages [7]. Bow's XML model provides a four-level hierarchical representation for the interlinear text, namely, text level, phrase level, word level and morpheme level. For the purpose of linguistic analysis, it is essential to preserve linear order between the words in the text [18]. Thus, there is a compelling need for ordered twig pattern matching. In addition to interlinear text, language treebanks have been widely used in computational linguistics. Treebanks capture syntactic structure of text data and provide a hierarchical representation of text by breaking them into syntactic units such as noun clauses, verbs, adjectives and so on. A recent paper by Müller *et al.* used ordered pattern matching over treebanks for question answering systems [15].

Our FiST system matches twig patterns *holistically* using the idea of encoding XML documents and twig patterns into Prüfer sequences [17]. The Prüfer's method constructs a one-to-one correspondence between labeled trees and sequences. (Refer to Section 3). It was shown in the PRIX system [17] that the above encoding supports ordered twig pattern matching efficiently. A collection of sequences for twig patterns are organized into a

```
<A>
  <B>
    <D> v₁ </D>
    <E> v₂ </E>
  </B>
  <E>
    <G> v₃ </G>
    <F> v₄ </F>
    <F> v₅ </F>
  </E>
</A>
```

Tree representation:

$(A,13)$
$(B,5)$ $(E,12)$
$(D,2)$ $(E,4)$ $(G,7)$ $(F,9)$ $(F,11)$
$(v_1,1)$ $(v_2,3)$ $(v_3,6)$ $(v_4,8)$ $(v_5,10)$

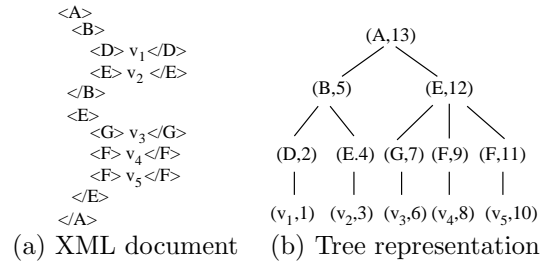(a) XML document   (b) Tree representation

Figure 1: A sample XML document

dynamic hash based index for efficient filtering. Our filtering algorithm involves two phases: *Progressive Subsequence Matching* and *Refinement for Branch Node Verification*. While the first phase identifies a superset of twig patterns that potentially match an incoming document, the second phase discards false matches by performing post-processing for branch nodes in the twig patterns. Our extensive experimental study shows that the holistic matching approach enables FiST to outperform the state-of-the-art YFilter system by achieving better scalability under various situations.

The key contributions of our work are summarized as follows.

- We have developed a new filtering system called **FiST** that supports the holistic matching of twig patterns against incoming XML documents. The matching is holistic since the twig pattern is matched as a whole rather than matching individual linear paths from root-to-leaf first.
- Our filtering algorithm involves a novel *progressive subsequence matching* phase followed by a *refinement phase* for branch node verification. This two-phase processing guarantees that matching by FiST is free of false positives and false negatives.
- By using a single runtime stack, the subsequence matching phase is optimized by avoiding redundant accesses to the hash index. The stack enables the testing of *parent-child* and *ancestor-descendant* relationships and limits the search space during the subsequence matching.
- The FiST system provides ordered twig matching for applications that require the nodes in a twig pattern to follow document order in XML.

The remainder of this paper is organized as follows. Section 2 provides the background and motivations of our work. We present the overview of the FiST system in Section 3. In Section 4 we describe the core algorithms and optimizations in FiST. Section 5 discusses our experimental results. A survey of related work is presented in Section 6. We conclude our work in Section 7.

2 Background and Motivations

XML documents can be modeled as ordered labeled trees. For example, the XML document in Figure 1(a) can be represented as an ordered labeled tree as shown in Figure 1(b). Each node in a tree corresponds to an element or a value. Values are represented by character data (`CDATA`, `PCDATA`) and appear at the leaf nodes. The tree edges represent a relationship between two elements or between an element and a value. Each element can have a list of (attribute, value) pairs associated with it. In this paper, attributes are treated the same way

as elements. Hence, no special distinction will be made between elements and attributes in the subsequent discussions. In the following sections, we will overview the key ideas of XFilter and YFilter, and provide motivations for our work.

XFilter and YFilter

The XFilter system maps each location path expression into a finite state machine (FSM). The collection of FSMs are indexed to support efficient filtering. YFilter system builds on XFilter and relies on shared path matching to improve the scalability of the filtering system. YFilter constructs a single non-deterministic finite automaton (NFA) for all the path expressions. YFilter supports incremental construction and maintenance. In addition, YFilter supports value-based predicates in the path expressions. YFilter handles twig patterns by decomposing them into individual linear paths and then performing post-processing over linear path matches. Path sharing is also exploited for twig queries.

Our Motivations

None of the previous work on XML filtering supports *holistic matching* of twig patterns against incoming XML document. In addition, no previous work has addressed *ordered matching* of twig patterns, which is needed in applications where the order of the nodes in the twig patterns should follow the document order in XML. These shortcomings have motivated us to build a filtering system that supports holistic matching of twig patterns with inherent support for ordered pattern matching.

3 The FiST System

In this section, we present the key ideas of the FiST system. We first formulate the filtering problem that we address in this paper. We then provide an architectural overview of the FiST system and briefly describe its core components. Finally, we explain the construction of Prüfer sequences for labeled trees.

3.1 XML Document Filtering

The filtering problem is different from the task of finding all occurrences of a twig pattern in an XML database. In traditional XML indexing and query processing, XML documents are indexed to quickly find all occurrences of a twig pattern (*e.g.*, XISS [13], TwigStack [6], PRIX [17]). However, in XML filtering, the role of twig patterns and documents are reversed. It is the twig patterns that are indexed in order to quickly determine whether those twigs appear in the input document to be filtered. Formally the problem of XML document filtering can be stated as follows.

> Given a set Q of twig patterns and an XML document D, find the subset $Q' \subseteq Q$ such that every $q \in Q'$ has a match in D.

In this paper, we focus on ordered matches, where the ordering of twig pattern nodes should match the document order.
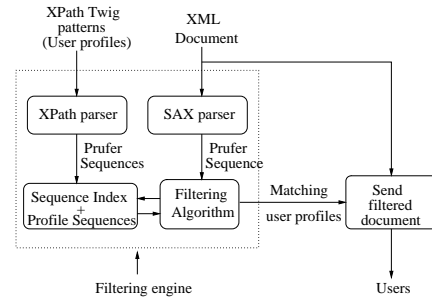

Figure 2: Architecture Overview

3.2 Architectural Overview

In this section, we shall describe the core components of the FiST system. Figure 2 shows an architectural overview. The core filtering engine is shown in a dotted box.

User profiles expressed in XPath are parsed using an XPath parser and converted into Prüfer sequences. We defer the description of Prüfer sequence construction to Section 3.3. User profiles can be updated during the execution of the filtering engine. The collection of sequences are stored in a hash based dynamic index called *Sequence Index*. For each sequence an auxiliary list called *Profile Sequence* is maintained. The *Sequence Index* and a collection of *Profile Sequences* make up the core data structures of our filtering engine. We will delve into the details pertaining to the index construction and maintenance in Section 4.

Incoming XML documents that need to be filtered are first parsed using a SAX parser [14]. The SAX parser generates a *start tag* event for each opening tag of an element and an *end tag* event for each closing tag of an element. The filtering engine progressively constructs the Prüfer sequence representation of the document and performs certain operations on these events. With this high level overview of the FiST system, we shall move on to explain the Prüfer sequence construction.

3.3 Prüfer Sequences for Labeled Trees

Prüfer (1918) proposed a method that constructed a one-to-one correspondence between a labeled tree and a sequence by removing nodes from the tree one at a time [12]. The algorithm to construct a sequence from tree T_n with n nodes labeled from 1 to n works as follows. From T_n, delete a leaf with the smallest label to form a smaller tree T_{n-1}. Let a_1 denote the label of the node that was the parent of the deleted node. Repeat this process on T_{n-1} to determine a_2 (the parent of the next node to be deleted), and continue until only two nodes joined by an edge are left. The sequence $(a_1, a_2, a_3, ..., a_{n-2})$ is called the Prüfer sequence of tree T_n. From the sequence $(a_1, a_2, a_3, ..., a_{n-2})$, the original tree T_n can be reconstructed. The length of the Prüfer sequence of tree T_n is $n-2$. Similar to the PRIX system, we construct a Prüfer sequence of length $n-1$ for T_n by continuing the deletion of nodes till only one node is left.

The labeled Prüfer sequence (LPS) of an XML document tree is obtained by replacing the node numbers in the sequence with XML tags [17]. Extended Prüfer sequences can be constructed by extending leaf nodes of the document tree with dummy child nodes. As result, the leaf node labels of the original tree appear in
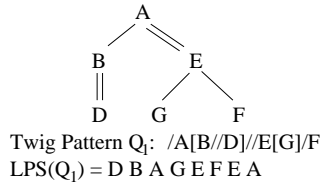
Twig Pattern Q_1: /A[B//D]//E[G]/F
LPS(Q_1) = D B A G E F E A

Figure 3: Twig Pattern to Sequence Conversion

the LPS. The following example illustrates the sequence representation for an XML document tree.

Example 1 *Consider the XML document tree in Figure 1(b). The nodes of the tree are labeled in postorder. The Prüfer sequence of the tree using the node numbers is 2 5 4 5 13 7 12 9 12 11 12 13. The LPS of this tree is D B E B A G E F E F E A. By extending the leaf nodes of the document tree with dummy child nodes v_1 through v_5, the extended LPS can be constructed and is v_1 D B v_2 E B A v_3 G E v_4 F E v_5 F E A.*

4 Index Structure and Filtering Algorithm

With a high level overview of our system, we shall now describe the index structure and the filtering algorithm of FiST. Subsequently, we will present a few optimizations to speed-up the filtering process. In the following sections, we will use the terms *"user profiles"* and *"twig patterns"* interchangeably.

4.1 Transforming User Profiles into Sequences

In the FiST system, user profiles expressed as XPath expressions are transformed into Prüfer sequences. The twig patterns we deal with have either parent-child relationship ('/') or ancestor-descendant ('//') relationship between two nodes. In this section, we shall describe the method to map twig patterns into sequences. For simplicity let us first consider patterns without wildcard '*'. Later in Section 4.4, we will describe how '*' can be handled.

A twig pattern is first mapped to a tree structure by treating the nodes in the pattern as tree nodes. Both '/' and '//' in the pattern are treated as regular tree edges. Note that in this paper, we focus on ordered twig pattern matching. For example, consider the tree representation of the pattern Q_1 in Figure 3. The extended LPS of Q_1 is D B A G E F E A. For each user profile, in addition to the LPS, additional information like the relationships between the nodes (parent-child or ancestor-descendant) and branch node information are stored. This auxiliary list is called the *Profile Sequence*. Each profile sequence is represented by an ordered list of nodes. For a parent-child (or ancestor-descendant) pair of nodes in the twig pattern, this relationship information is stored in the profile sequence node corresponding to the child (or descendant). Each node in the profile sequence has four attributes namely **Label**, **Qid**, **Pos**, and **Sym**. These attributes are summarized in Table 1. The attribute **Label** stores the Prüfer sequence label, **Qid** contains an unique identifier, **Pos** denotes the position of the node in the profile sequence, and **Sym** stores a combination of values listed in Table 2. Given a node q in the profile sequence, the four attributes are denoted by q_{Label}, q_{Qid}, q_{Pos} and q_{Sym} respectively.

Attribute	Description
Label	A Prüfer sequence label
Qid	An unique sequence identifier
Pos	An integer that describes the location of this sequence node in the profile sequence
Sym	A set of values that describe the type of sequence node

Table 1: Node Information

Value	Description
'/'	A parent-child relationship
'//'	An ancestor-descendant relationship
'$'	A branch node
'#'	The root node in the profile sequence

Table 2: Symbol Values

Example 2 *Figure 4(a) shows two profile sequences for the twig patterns Q_1 and Q_2 where LPS's are D B A G E F E A and E B C B respectively. So there are eight nodes in the profile sequence of Q_1 and four nodes in that Q_2. The relationships are stored in the **Sym** attribute of each node in a profile sequence. For example in the profile sequence of Q_1, the **Sym** attribute of node D has the value '//' because the first node D and the second node B have an ancestor-descendant relationship in Q_1. Q_1 has two branch nodes A and B which have two child nodes each. Hence the third, fifth, seventh and eighth nodes in the profile sequence of Q_1 have $ in their **Sym** attribute. Note that some branch nodes have two symbol values at the same time. For example, in the profile sequence of Q_1, the seventh node with **Label** E has and ancestor-descendant relationship with the eighth node representing node A in Q_1. Hence its **Sym** attribute has values '$' and '//'. The last node in the profile sequence always corresponds to the root of the twig pattern. We call this node as the root node of the profile sequence. The **Sym** attribute of this node has value '#'.*

4.2 Indexing User Profiles

Given a large number of user profiles that need to be matched against incoming XML documents, it is natural that a good indexing strategy should be developed for efficient filtering. In this section, we propose an index structure to store the profile sequences.

Conceptually, the first phase of the filtering algorithm in FiST involves subsequence matching between the profiles sequences and the input document sequence in order to compute the superset of the twig patterns that match the input document. The following theorem states the relationship between the sequence representation of a twig pattern and an XML document.

Theorem 1 ([17]) *If tree Q is a subgraph of tree T, then $LPS(Q)$ is a subsequence of $LPS(T)$.*

The nodes in a profile sequence can be mapped to a state machine. (See Figure 4(c).) This figure is a simplified illustration since the actual state machine has more transitions and is not shown here for the purpose of clarity. As new tags in the input document are parsed, the state machine undergoes appropriate transitions. If the state machine reaches the final state, the profile sequence

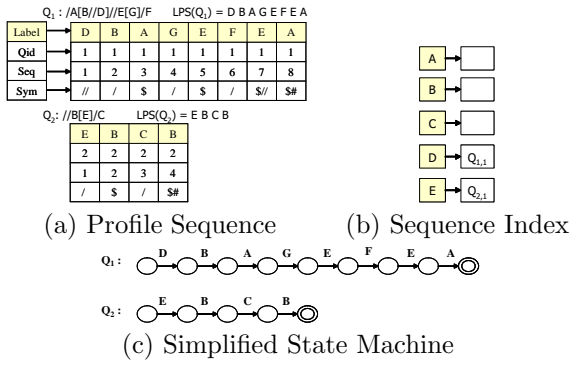(a) Profile Sequence   (b) Sequence Index

(c) Simplified State Machine

Figure 4: Profile Sequence, Sequence Index and State Machine

has a subsequence match in the document sequence. For efficient filtering, however, it is desired to perform subsequence matching on all the profile sequences simultaneously. To do so, we maintain a dynamic hash-based index called the *Sequence Index*.

Example 3 *A Sequence Index is shown in Figure 4(b). The XML tags are used as keys in the hash table. For each key, the value is a list of nodes from the profile sequences. At the start of filtering, the first nodes of the profile sequences are added to the hash index. As new Prüfer sequence labels of the input XML documents are obtained, state transitions take place and the nodes are added to the hash table. It can be observed that the Sequence Index contains the first node of profile sequences for patterns Q_1 and Q_2 for hash keys D and E respectively.*

4.3 Filtering Algorithm

In this section we will describe our filtering algorithm that involves *progressive subsequence matching* followed by a refinement phase for *branch node verification*. Theorem 1 is a necessary but not a sufficient condition. In the subsequence matching phase, our filtering algorithm performs additional tests to eliminate most false matches. For a given a profile sequence node q, the nature of the test depends on the value of q_{Sym}. To facilitate these tests, a *runtime global stack* is maintained by our filtering algorithm that stores the tags along the path from the current tag being processed to the root of the document. The elements are pushed to and popped from the global stack in document traversal order. Note that the maximum depth of the stack is no more than the maximum height of the incoming documents.

4.3.1 Progressive Subsequence Matching

It is essential that we find the subsequence matches simultaneously for all the twig sequences in a scalable manner. We call the subsequence matching *progressive*, because we generate the sequence representation of the document incrementally and find those profile sequences that are subsequences in steps. The LPS of an XML document is constructed incrementally by examining the run time global stack as the document is being parsed. In the FiST system, a SAX parser is used to parse input XML documents. A few modifications need to be made to the `StartTagHandler` and `EndTagHandler` procedures of the SAX parser to accommodate our filtering algorithm. Algorithm 1 shows the
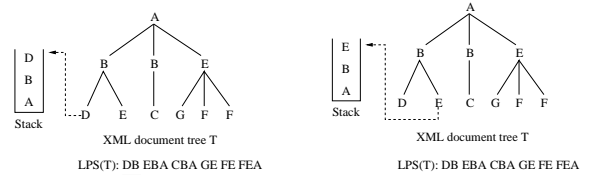

(a) EndTagHandler of D   (b) EndTagHandler of E

Figure 5: Generation of LPS(T)

`StartTagHandler` and the `EndTagHandler` procedures. When the `StartTagHandler` is invoked with a tag name, the tag name is pushed onto the stack as shown in Line 1. When the `EndTagHandler` is invoked, the element tag is checked if it is a leaf node in the document in order to generate an extended LPS (Line 2). If the tag is a leaf node, the top element of the stack is used as the next Prüfer sequence label and the filtering procedure (`FindSubsequence(·)`) is invoked (Line 3). Whether the tag is a leaf or not, the top element is popped from the stack and the new top element is used as the next Prüfer sequence label (Line 4). The filtering procedure is again invoked (Line 5).

Algorithm 1: SAX Handlers

---

stack S;   /* a runtime global stack */
procedure `StartTagHandler`(tag)
1: S.push(tag)
end

procedure `EndTagHandler`(tag)
              /* for extended Prüfer sequence */
2: if *tag is a leaf node* **then**
3:   FindSubsequence (S.top());
   end
4: S.pop();
5: FindSubsequence (S.top());
end

Example 4 *We illustrate the construction of the LPS of the XML document tree T as shown in Figure 5. For this document LPS(T) is DBEBACBAGEFEFEA. When the StartTagHandler(·) of an element is invoked, the element is pushed onto the stack. When the EndTagHandler(·) of D is invoked, the state of the stack is shown in Figure 5(a). Element D is a leaf node in T. So the top element in the stack represents the 1st label of LPS(T). The top element D is then popped from the stack. As a result, the new top element B in the stack represents the 2nd label of LPS(T). Note that element B is still kept in the stack after it is used. When the EndTagHandler(·) of E (child of B) is called, the state of the stack is shown in Figure 5(b). E is also a leaf node in T. Hence the top element E in the stack represents the the 3rd label of LPS(T). Then, the top element of the stack is popped. After this, the new top element B in the stack represent the 4th label of LPS(T). Subsequently when the EndTagHandler of B is invoked, B and A are the two elements in the stack. Since element B is not a leaf node in the XML document tree T, the top element in the stack is popped. The new top element A in the stack represents the 5th label of LPS(T). The above process is repeated till the EndTagHandler of the root element (i.e., A) is invoked.*

Each time the `EndTagHandler`(·) is invoked, the top element of the stack indicates the ith element of the LPS

of the document. Our filtering algorithm uses the *Sequence Index* to find matching subsequence simultaneously for all the profile sequences. As a result, the document and its LPS are scanned only once. Hence it is suffice to generate the LPS incrementally by accessing the global stack without actually storing the entire sequence.

Algorithm 2: Progressive Subsequence Matching

Input: {L} - L is a Prüfer sequence label;
procedure FindSubsequence(L)
1: $CurrentList \leftarrow SequenceIndex[L]$;
2: **foreach** *SequenceNode q in CurrentList* **do**
3:    $test \leftarrow false$;
4:    **foreach** *value v in* q_{Sym} **do**
5:       **switch** v **do**
         /* Parent-Child or Ancestor-Descendant relationship */
6:          **case** '/' *or* '//':
7:           **if** *doSimpleStackTest* $(q, v) = true$ **then** $test \leftarrow$ true;
         /* Branch node */
8:          **case** '$': doBranch (q);
         /* Root node of twig pattern */
9:          **case** '#':
           BranchNodeVerification (q_{Qid});
      end
   end
10:    **if** $((q_{Sym} = '/' \text{ or } q_{Sym} = '//') \text{ and } (test = true))$ *or* $(q_{Sym} = '$')$ **then**
11:       $q' \leftarrow$ NextNode(q);
12:       copy q' to Sequence Index using key q'_{Label};
   end
 end
end

During subsequence matching, FiST performs additional tests to eliminate most false matches by using a runtime stack. In essence, transitions occur in a state machine (*e.g.*, Figure 4(c)) when both the tag name is matched and the stack test succeeds. The runtime stack has three main benefits during filtering namely (a) for testing parent-child and ancestor-descendant relationships, (b) for avoiding frequent node copying into the hash index, and (c) for pruning subsequences by limiting the range of subsequence matching. The core operations during filtering are shown in Algorithm 2. The procedure FindSubsequence($\cdot$) is invoked from the EndTagHandler($\cdot$). Using the label L as the key, the Sequence Index is searched to obtain the list of nodes to be tested (Line 1). For each node q in the list, an appropriate action is taken depending on the values in q_{Sym} (Lines 6-9). Note that since q_{Sym} is a list of values, we iterate through each value in Line 4.

Processing Parent-Child and Ancestor-Descendant Relationships

The runtime stack will be used for testing parent-child and ancestor-descendant relationships between nodes in the input document that match the nodes in the profile sequences. Let us refer to these two tests as TestPC($\cdot$) (parent-child) and TestAD($\cdot$) (ancestor-descendant). These tests will differ in the extent to which the stack is checked. Consider two nodes q and

$q' = $ NextNode(q) in a profile sequence. TestPC(q, q') is successful, if q'_{Label} is immediately below q_{Label} in the stack. On the other hand, TestAD(q) is successful, if q'_{Label} occurs somewhere below q_{Label} in the stack. Whenever such a test is successful, the state machine moves to the next state.

For example, in Figure 6, when the EndTagHandler of E is called, the state of the stack is shown on the left. The jth element in the profile sequence for Q_i matches the top of the stack. Since the symbol value of the jth element is '//', we apply TestAD$(Q_{i,j}, Q_{i,j+1})$. Since element C is two elements below E in the stack, TestAD$(\cdot)$ is successful. This means that the ancestor-descendant relationship between nodes $Q_{i,j+1}$ and $Q_{i,j}$ in the twig pattern in Figure 6 is satisfied in the document T. Next we attempt to match $Q_{i,j+1}$. Element C in the stack matches $Q_{i,j+1}$. Besides, element B is one element below C in the stack. This means that the parent-child relationship between nodes $Q_{i,j+2}$ and $Q_{i,j+1}$ is satisfied in the document T, that is, TestPC$(\cdot)$ is successful. Note the procedure doSimpleStackTest$(\cdot)$ is called from FindSubsequence$(\cdot)$ to perform the above operations (Line 6).
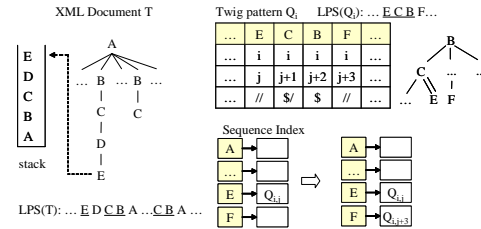

Figure 6: Benefits of Runtime Global Stack

Algorithm 3: Simple Stack Checking

Input: q is a node in the profile sequence;
       v is a value in the **Sym** attribute of q
procedure doSimpleStackTest(q, v)
1: $q' \leftarrow$ NextNode(q);
2: **if** $(v = '/' \text{ and } $ TestPC(q, q') *is successful*) OR $(v = '//' \text{ and } $ TestAD(q, q') *is successful*) **then**
3:    **return** true;
4: **else return** false;

Note that the twig pattern in Figure 6 can match anywhere in the incoming document. However if node B was required to match the root of the document ('/'), then the runtime stack is checked to determine if B is the only element in the stack. If so, then this implies that node B in the twig pattern matches the document root.

Avoiding Frequent Node Copy to Sequence Index

Let us again consider the example in Figure 6. Based on Algorithm 1, FindSubsequence($\cdot$) is invoked each time the EndTagHandler is called. In the example, when the EndTagHandler for leaf E is called, the set of elements in the stack represent a segment of the LPS(T), *i.e.*, E D C B A. Note that the node A is a branch node. A naive way is to invoke FindSubsequence($\cdot$) once for each of E D C B A in order using Algorithm 1. This requires

copying the next node of a profile sequence to the Sequence Index each time `FindSubsequence`(·) matches a node in the profile sequence. However, since the runtime stack stores a segment of the LPS till the branch node A, we can use it as a look-ahead buffer. Thus instead of performing `doSimpleStackTest`(·) for each tag, a recursive stack check can be performed thereby eliminating the copying of nodes in the profile sequence up to the branch node. This process is shown in Algorithm 4.

A modification to Algorithm 2 is done by replacing the procedure `doSimpleStackTest` with `doRecursiveStackTest` on Line 6 and by omitting Lines 10 through 12. In Algorithm 4, on a successful stack test (Line 2), the next node of q in the profile sequence, *i.e.*, q' is used for subsequence matching by performing tests similar to Algorithm 2. Thus our algorithm tries aggressively to find subsequence matches up to the branch node. On success, the next node of the branch node is copied to the Sequence Index. In essence, we have effectively skipped copying nodes up to the branch node in the twig pattern. Note that in Algorithm 4, if a branch node in the profile sequence has a '/' or '//' relationship with the next node, then the subsequence matching continues by invoking `doRecursiveStackTest`(·).

Algorithm 4: Recursive Stack Checking to Avoid Node Copying

Input: q is a node in the profile sequence;
v is a value in the **Sym** attribute of q

procedure doRecursiveStackTest(q, v)

1: $q' \leftarrow$ NextNode(q);
2: **if** (v = '/' and *TestPC(q, q')* is successful) OR
      (v = '//' and *TestAD(q, q')* is successful) **then**
3:    **foreach** value v' in q'_{Sym} **do**
4:       **switch** v' **do**
5:          **case** '/' or '//':
               doRecursiveStackTest (q', v');
6:          **case** '\$': doBranch (q');
7:          **case** '#':
               BranchNodeVerification (q'_{Qid});
            end
         end
8:    **if** q_{Sym} = '\$' **then**
9:       copy q' into Sequence Index using key q'_{Label};
      end
   end

Limiting the Range of Subsequence Matching

Another important benefit of the runtime stack is that we can limit the range of the document sequence for subsequence matching. Consider an XML document T and a twig pattern Q_i in Figure 6. For document T, LPS(T) = ... E B C B A ... C B A ... For twig pattern Q_i, LPS(Q_i) = ... E C B ...LPS(T) has two subsequence instances that match 'E C B'. (They are underlined in Figure 6.) In the XML document, the second element C (leaf node in T) and its parent, *i.e.*, B, do not have any relationship with element E. When the Prüfer sequence label E of LPS(T) is generated, there is only one C and B in a global stack. Thus our filtering algorithm finds only one

instance of subsequence 'E C B' using the elements in the stack. As a result, the stack provides pruning capability by avoiding the computation of matching subsequences that do not represent true matches.

Below we illustrate the execution of our filtering algorithm with an XML document T in Figure 7(a) and twig patterns Q_1, Q_2 and Q_3 in Figure 7(b). The nodes $Q_{1,1}$, $Q_{2,1}$, and $Q_{3,1}$ are initially stored in the Sequence Index shown in Figure 7(c). This figure also shows the changes to the Sequence Index during the filtering process. In this example, we will illustrate the use of stack tests for parent-child and ancestor-descendant relationships. The branch detection techniques are deferred until Section 4.3.2.

Example 5 *When* `FindSubsequence`*(D) is invoked, the state of the runtime stack is shown in Figure 7(a). The node list in the Sequence Index for key D is first processed. Currently two nodes $Q_{1,1}$ and $Q_{2,1}$ qualify. First, let us consider $Q_{1,1}$. The next node for $Q_{1,1}$ is $Q_{1,2}$. The progressive subsequence matching phase succeeds since $Q_{1,2_{Label}} = B$, $Q_{1,1_{Sym}} = $ '//', and B is one element below D, the stack test is successful (i.e.,* `TestPC`*(·)). Now* `doRecursiveStackTest`*($Q_{1,2}$, '/') is invoked (Line 5 in Algorithm 4). The next node for $Q_{1,2}$ is $Q_{1,3}$. Because the label A is one element below the label B in the stack, the stack test is successful again. $Q_{1,3}$ is a branch node, so the next node $Q_{1,4}$ is added to the Sequence Index using hash key G. Thus we have matched nodes $Q_{1,1}$ through $Q_{1,3}$. Here we can skip copying the $Q_{1,2}$ and $Q_{1,3}$ into a Sequence Index. Next, let us consider $Q_{2,1}$. The next node for $Q_{2,1}$ is $Q_{2,2}$. A stack test for $Q_{2,1}$ is successful and $Q_{2,2}$ is matched. Then* `doRecursiveStackTest`*($Q_{2,2}$, '/') is called but a stack test for $Q_{2,2}$ is unsuccessful since C is not present in the stack. In this case, no node copying is done, and the $Q_{2,1}$ remains in the Sequence Index as shown in Figure 7(c).*

Next, when `FindSubsequence`*(B) is invoked, there are no nodes in the Sequence Index for hash key B. Hence nothing is done. Then,* `FindSubsequence`*(E) is invoked, $Q_{3,1}$ passes the stack check and $Q_{3,2}$ is matched. Because $Q_{3,2}$ is a branch node, the next node $Q_{3,3}$ is copied to the Sequence Index using hash key C. Next* `FindSubsequence`*(B) is invoked again, there are no nodes in the Sequence Index for hash key B. Hence nothing is done. When* `FindSubsequence`*(C) is invoked, the global stack at this instant has elements C, B, and A in it. Only node $Q_{3,3}$ is active in the Sequence Index for hash key C. Since $Q_{3,3_{Sym}} = $ '/' and label B is below the label C in the runtime stack, we have matched the next node $Q_{3,4}$ which a branch node and a root node.*

4.3.2 Branch Node Processing

It was shown that filtering by subsequence matching alone can lead to false matches [17]. To eliminate such false matches in FiST, we have developed refinement techniques to test the connectedness property for branch nodes in twig patterns. In this section, we begin with an example to motivate the need for special branch node processing to support the refinement phase. The refinement phase will be described in Section 4.3.3. For
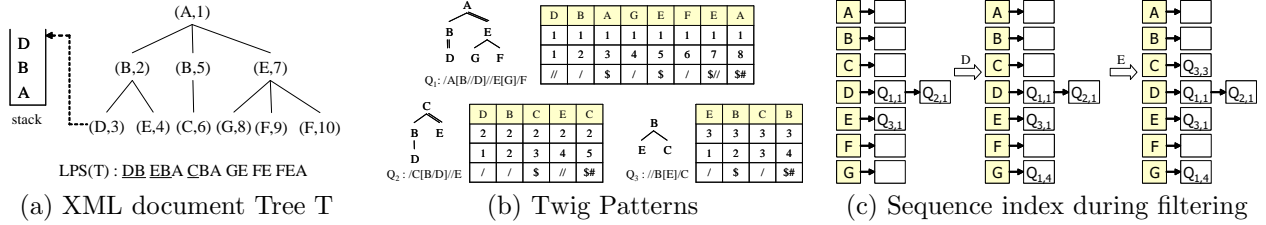
(a) XML document Tree T   (b) Twig Patterns   (c) Sequence index during filtering

Figure 7: Progressive Subsequence Matching

convenience, assume that the nodes in each XML document tree are numbered in preorder. [1] Annotating the elements in an XML document with the preorder numbers is straightforward during SAX parsing. A counter can be maintained and incremented on every call to the `StartTagHandler`. The counter value is then assigned to the tag being processed.

Example 6 *Consider the example in Figure 7. The XML document tree T is numbered in preorder (see Figure 7(a)). Figure 7(b) shows two twig patterns Q_1 and Q_3. For document tree T, LPS(T) = D B E B A C B A G E F E F E A. For twig pattern Q_1, $LPS(Q_1) = D B A G E F E A$ and for Q_3, $LPS(Q_3) = E B C B$. $LPS(Q_1)$ is a subsequence of LPS(T) and $LPS(Q_3)$ is also a subsequence of LPS(T). Both twig patterns Q_1 and Q_3 are candidates that could be possible matches in T. However, Q_3 is not a true match since there is no node B in T that has both E and C as child nodes. Thus Q_3 matches two different B nodes in T, i.e., (B,2) and (B,5). In order to eliminate such false matches, it is essential to ensure that the B nodes that were matched during subsequence matching represent one and the same node in T.*

On the other hand, two E nodes of $LPS(Q_1)$ matched two E nodes in LPS(T) that represent one and the same node in T, i.e., (E,7). And two A nodes of $LPS(Q_1)$ matched two A nodes in LPS(T) that represent one and the same node in T, i.e., (A,1). Note that Q_1 has a match in T.

FiST performs special processing for branch nodes in the profile sequences in order to facilitate the refinement phase to discard false matches. Thus the main role of a branch node processing is to store the information of matched tag in the input XML document. This information is used in the refinement phase to check the connectedness property. A data structure called the *BranchID Set* (*i.e.*, a list) is maintained for each occurrence of a branch node in the profile sequence to keep track of the preorder number of the node in the document that matches the sequence node.

During the subsequence matching phase, when the profile sequence node say q is a branch node, then `doBranch`(q) stores the preorder number of the node in the document with label q_{Label} (that matches q) in the *BranchID set* for q. In a Prüfer sequence, the number of times a given node appears in it depends on the number of its child nodes [17]. Thus a profile sequence has two types of branch nodes for any given branch node in a twig pattern. For example, in Figure 7(b), the

profile sequence nodes of Q_1 *i.e.*, $Q_{1,5}$ and $Q_{1,7}$ correspond to the branch node E in the twig pattern. $Q_{1,5}$ does not correspond to the last occurrence of E. We refer to such nodes as *internal branch nodes*. Note that the internal branch nodes in a profile sequence do not have any relationship with the next node in its profile sequence. Hence no stack checks are necessary and the filtering algorithm can proceed to test the next node for subsequence matching. However the last occurrence of the branch node has either a parent-child or ancestor-descendant relationship with its next node. For example, node $Q_{1,7}$ in Figure 7(b) has an ancestor-descendant relationship with $Q_{1,8}$. We refer to such nodes in a profile sequence as *final branch nodes*. For a final branch node, in addition to storing the preorder number of the document node in its BranchID set, the stack test is required. Only on successful stack test, the filtering algorithm examines the next node for subsequence matching. Note that the recursion in `doRecursiveStackTest`(·) terminates when an internal branch node is processed.

In Figure 7(b), two BranchID sets are maintained for $Q_{1,3}$ and $Q_{1,8}$ corresponding to node A. Similarly, two BranchID sets are maintained for $Q_{1,5}$ and $Q_{1,7}$ corresponding to node E.

4.3.3 Root Node Processing

In this section, we present the *Refinement by Branch Node Verification* phase to determine twig patterns that appear in the incoming document. The subsequence matching phase computes a superset of twig patterns that are candidates. False matches are eliminated by verifying the connectedness property at the branch nodes in the twig patterns from this candidate set.

For each candidate profile sequence, its BranchID Sets that are constructed during the subsequence matching phase are examined. Algorithm 5 shows the steps involved. The procedure `BranchNodeVerification`(·) is invoked from Algorithms 2 and 4 when the root node of a profile sequence is processed. For each branch node l in the candidate twig pattern, the algorithm computes the intersection of the BranchID sets for each occurrence of the branch node in its profile sequence (Lines 3 through 6). A non-empty result set implies that this branch node l in the twig patterns matches a branch node in the input document since there exists at least one matching subsequence where all matching occurrences of this branch node in the profile sequence represent one and the same node in the document. If every branch node in the twig pattern matches at least one branch node in the document, then it is reported as a match (Line 7).

For example, consider the twig pattern Q_1 in Figure 7(b). The intersection of the BranchID sets for node

---

[1]Other numbering schemes like postorder can also be used. However preorder numbering seems to be a natural choice since the tags in the document are parsed in document traversal order.

Algorithm 5: Branch Node Verification

Input: $\{q_{Qid}\}$: q_{Qid} is a profile sequence identifier;
Output: Report a match;

procedure `BranchNodeVerification` (q_{Qid})

1: $test \leftarrow true$;
2: $L_B \leftarrow$ list of labels of the branch nodes in the twig pattern with id Qid;
3: **foreach** l in L_B **do**
4:    $n \leftarrow$ number of BranchID sets for l in the profile sequence;
5:    let $B_{l1}, B_{l2}, ..., B_{ln}$ denote the n BranchID sets for l
6:    **if** $\bigcap_{i=0}^{n} B_{li} = \varnothing$ **then** $test \leftarrow false$;
   end
7: **if** $test = true$ **then** report q_{Qid} as a match;

A is $\{1\}$. Similarly, the intersection of the BranchID sets for node E is $\{7\}$. As a result, Algorithm 5 reports Q_1 as a match, since Q has a true match in T.

4.4 Wildcard Processing

If the wildcard '*' appears in a non-branch node in the twig pattern, it is refered to as a regular wildcard node. If a branch node has a wildcard in the twig pattern, then it is refered to as a branch wildcard node.

In the case of a regular wildcard node in a twig pattern, the sequence nodes for the wildcards are not generated in the profile sequence. Instead, the wildcard information is added to a sequence node which is the next label of a wildcard in the LPS. Thus in our filtering algorithm, when we process a node which contains wildcard information during a stack test, its wildcard information is also checked.

In the case of a branch wildcard node, we generate sequence nodes for the wildcard. When the filtering algorithm processes a branch wildcard node, we store preorder numbers of all the matched tags that passed the stack test. Finally the intersection of the BranchID sets is computed as before.
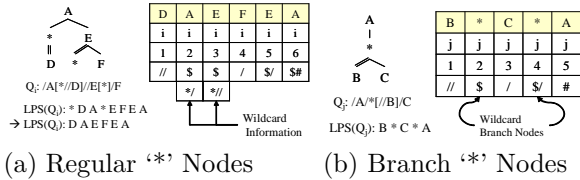

(a) Regular '*' Nodes   (b) Branch '*' Nodes

Figure 8: Processing Wildcard '*'

Example 7 *Consider the example in Figure 8. The twig pattern Q_i in Figure 8(a) has two regular branch wildcard nodes. The 2nd node and the 3rd node in the profile sequence have extra wildcard information. When we process the 1st node and 2nd node, the wildcard information will be checked. If there are some labels between label D and label A in the runtime stack, the stack check will succeed. Then, the next node (3rd node) of the branch node is copied to the Sequence Index. When the 3rd node is processed, we check whether there are some nodes above label E in the stack or not based on the ancestor-descendant relationship of the wildcard node.*

A twig pattern Q_j in Figure 8(b) shows two branch wildcard nodes in its profile sequence, which are the 2nd and the 4th nodes respectively. Assume there are four elements B, D, E and A (top to bottom) in a global

stack when we process the 1st node and the 2nd node. During the stack test, all the elements below the label B are eligible to be a match for a branch wildcard node because of the ancestor-descendant relationship. Thus the BranchID set stores the preorder numbers of all these candidate tags. Again assume that there are three elements C, E and A (top to bottom) in the stack when we process the 3rd node and the 4th node. In this case, only the preorder number of E is stored because of the parent-child relationship. The branch node verification phase computes the intersection of the BranchID sets for the two wildcard nodes.

5 Experiments

In our experiments we compared the FiST with the YFilter system [19]. FiST was implemented in C++ using Xerces XML Parser version 2.5.0 [2]. The YFilter package was implemented in Java and was obtained from the University of California at Berkeley. We measured the performance of YFilter and FiST for a variety of XML document sizes and user profiles (twig patterns). For the FiST, we observed a strong correlation between the number of matching twig patterns and the total filtering time. The filtering time of FiST decreased as the the number of matching user profiles was reduced. On the other hand, the filtering time for YFilter grew quickly as the size of the twig patterns increased. The results from our experiments show that FiST scales better than YFilter with an increasing number of twig patterns and documents of growing size due to holistic processing of twig patterns.

5.1 Experimental Setup

We ran all our experiments on a 2.4 GHz Pentium IV machine with 512 MB memory running Linux. Our code was compiled using GNU g++ compiler version 3.3.2. The YFilter code was run using Eclipse 3.0.1 with Java virtual machine version 1.4.2.

5.2 Datasets and Twig Patterns

We used synthetic Treebank data for our experiments. These data were generated by an XML Generator from IBM [9]. We generated 1000 documents of different sizes with a maximum document depth of 36 using the Treebank DTD. The Treebank data had deep recursion of elements. The generated documents were categorized into four datasets based on the document sizes in bytes. The four datasets had sizes in the range $[1KB, 10KB)$, $[10KB, 20KB)$, $[20KB, 30KB)$, and $[30KB, 123KB)$. In subsequent discussions, we will refer to these four datasets as "1k", "10k", "20k", and "30k" respectively.

We also generated user profile sets of two different distributions using the XPath generator available in the YFilter package. In one set, the element names were chosen from uniform distribution. In the other set, the element names are chosen from a skewed distribution with Zipfian skew parameter of $z = 0.9$. The maximum depth of a twig pattern was fixed at 10. For each uniform and skewed distribution, the number of branches in the twig patterns were varied from 3 to 7. We varied the number of twig patterns that were indexed by the filtering system from 50,000 to 150,000 in steps of 25,000.

5.3 Different Aspects of Scalability and Evaluation Metrics

We have evaluated the FiST system in three different aspects of scalability. We compared the scalability of FiST and YFilter (1) by varying the total number of twig patterns, (2) by varying the number of branches in the twig patterns and (3) by varying the size of input documents. Note that FiST supports *ordered twig pattern* matches and YFilter supports *unordered twig pattern* matches. A simple post-processing step can be added to YFilter to support ordered matches. For YFilter, we measured the total time without the post-processing stage.

We compared YFilter and FiST by observing the trend in the filtering cost in different aspects of scalability. This is because YFilter was implemented in Java and FiST was implemented in C++. In order to make fair comparisons between the two systems implemented in different languages and tested on different execution environments, we measured the performance of YFilter and FiST in terms of *scaleup* as well as *wall clock time.* When we measured the wall clock time, the average filtering time per document was computed for each dataset for a given twig set. Note that the filtering time is the sum of the document parsing time and the time taken by the filtering algorithm. The parsing time was very small compared to the filtering time. For example, the average parsing time consumed by YFilter for the dataset 30k was just 30 ms.

To measure the scaleup performance, we used the following formula.

$$scaleup = \frac{tAvg - tAvg_{base}}{tAvg_{base}} \qquad (1)$$

where $tAvg$ is the filtering time measured for the case under observation and $tAvg_{base}$ is the filtering time measured for the base case. Depending on the type of scalability being measured, the $tAvg_{base}$ can be the filtering time for the smallest number of twig patterns, the smallest number of branches, or the smallest size of input document. For example, if scalability is evaluated by varying the number of twig patterns that are indexed, then $tAvg_{base}$ is the filtering time measured for a twig set of 50,000 user profiles. Thus, a positive (or negative) measurement of scaleup indicates that the filtering cost increases (or decreases), as the scale of test cases grows. We observed throughout the experiments that FiST scales better (*i.e.*, the filtering time grows more slowly) than YFilter under various situations.

5.4 Performance Analysis

In this section, we analyze the performance of FiST and YFilter in terms of scaleup and wall clock time. Due to the space limitations, we present the scalability trend only for a few representative cases of document sizes, twig set sizes and the number of branches.

5.4.1 Varying Number of Twig Patterns

Figure 9 summarizes the wall clock time for FiST and YFilter for uniform and skewed twig sets with 6 or 7 branches per a twig. The number of twig patterns indexed by FiST and YFilter was varied from 50,000 to 150,000 in steps of 25,000. The results for dataset 1k are omitted since their trend was similar to that of 10k for both FiST and YFilter.

Let us analyze the results shown in Figure 9(a). The number of branches in the twig patterns was six. The filtering time for both YFilter and FiST grew as the number of twig patterns increased. For datasets 20k and 30k, FiST was significantly faster than YFilter. For example, for 150,000 twig patterns, FiST was 34% faster than YFilter for 20k dataset. Similarly for 150,000 twig patterns with 7 branches (see Figure 9(b)), FiST was 43% faster than YFilter for 20k. For dataset 10k (and 1k) FiST and YFilter yielded comparable performance. Similar trend was observed for the skewed twig set and the results are shown in Figures 9(c) and (d). The above results demonstrate that FiST scales better than YFilter as the size of the input documents increases.

Another observation that can be made from Figure 9 is that the filtering time increased for both FiST and YFilter as the document sizes were increased. This is consistent with the expected trend. See Section 5.4.3 for more about the scalability with respect to the document sizes.

5.4.2 Varying Number of Branches in Twig Patterns

In this section, we compare the scalability of FiST and YFilter with respect to the varying number of branches in the twig patterns. The results are shown in Figure 10. Figures 10(a) and (b) show the *scaleup* for dataset 20k using the uniform twig set for YFilter and FiST, respectively. The filtering cost of YFilter increased as the the number of branches in the twig patterns increased from 3 to 7. This trend was observed for all the twig set sizes that we used. On the contrary, the filtering time for FiST decreased with increase in the number of branches. This is shown by the negative scaleup in Figure 10(b). The reason why the filtering time for FiST decreased with increase in the number of branches was due to the reduction in the number of matching twig patterns. This downward trend was consistent across all twig set sizes. For example, the dataset 30k had an average of 185.3, 16.9, 3.9, 2.4, and 1.9 matching twigs per document for the twig sets with 3, 4, 5, 6 and 7 branches respectively. On the other hand, YFilter's performance degraded with increase in the number of branches despite the decrease in the number of matching twigs. Similar performance trend was observed in other cases shown in Figures 10(c) through (h). These results demonstrate that the *holistic matching* of twig patterns by FiST yields better scalability than YFilter.

Figures 11(a) and (b) show the wall clock time for FiST and YFilter for datasets 20k and 30k. FiST consistently outperformed YFilter for twig sets with 4, 5, 6, and 7 branches. As described above, we observed an increasing trend in the filtering cost for YFilter and a decreasing trend in the filtering cost for FiST. We also observed that the filtering time increased as the document sizes increased.

5.4.3 Varying XML Document Sizes

In this section, we analyze the performance of FiST and YFilter by comparing their scaleup factor, as the size of
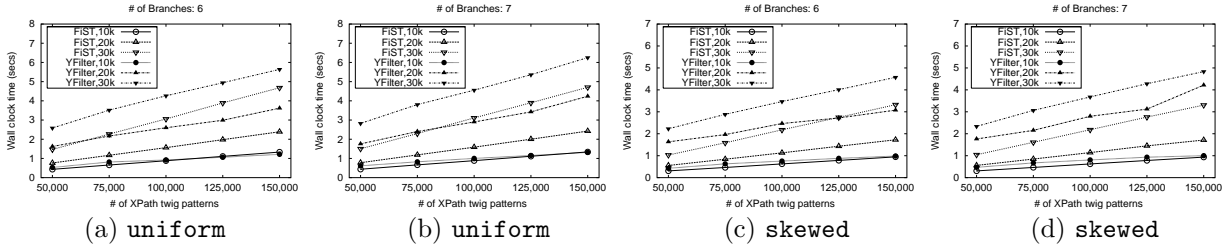
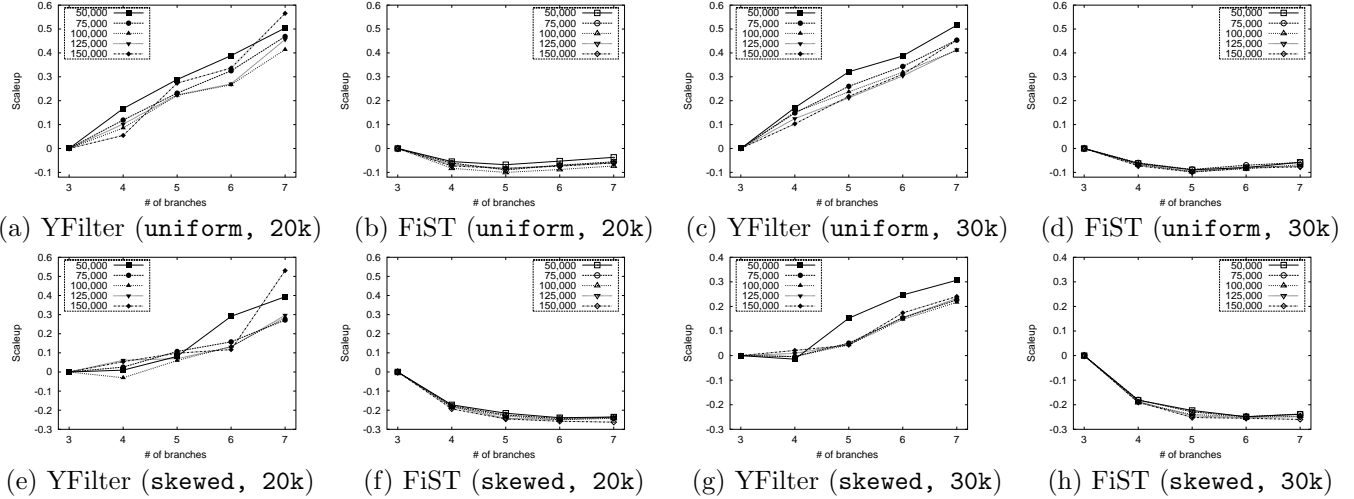Figure 9: Varying Number of Twig Patterns (User Profiles)

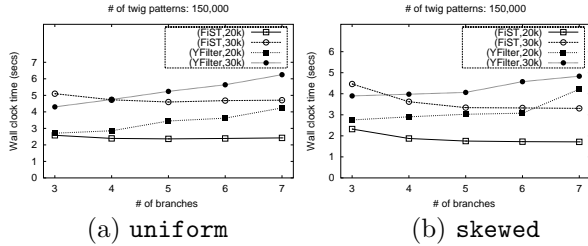Figure 10: Varying Number of Branches

Figure 11: Filtering Time

the documents increases.

The results are summarized in Figure 12. For each of the plots, results for twig sets with 4 and 6 branches were omitted since they showed trend similar to twig sets with 3, 5, and 7 branches for both YFilter and FiST. Figure 12(a) shows *scaleup* for YFilter and FiST for a uniform twig set of 125,000 twig patterns. Along the x-axis, we show the increase in the document sizes by using the datasets 1k, 10k, 20k and 30k. The scaleup of YFilter grew quicker than that of FiST indicating that YFilter's filtering cost increased much faster than FiST. Note that the lower set of lines correspond to FiST. We observed that the gap in the scaleup between YFilter and FiST was widened, as the size of the documents increased. Similar trend was observed in other scenarios shown in Figures 12(b) through (d). These results demonstrate that FiST scales better than YFilter, as the document sizes increase.

From the graphs shown in Figures 12(a) through (d), it can be seen that the performance trend for FiST almost overlap each other for twig sets with 3, 5 and 7 branches indicating that the trend is similar despite the increase in the number of branches. This is consistent with the performance trend shown in Figure 10 for varying the number of branches.

6 Related Work

The popularity of extensible markup language XML as a standard for information exchange has triggered several research efforts to build scalable XML filtering systems. Most of the previous approaches have been based on constructing automaton representations for the user profiles.

XFilter [1] was one the early work in XML filtering. XFilter handles simple XPath expressions by transforming each expression them into a single finite state machine. YFilter [19] is a continual work of XFilter and uses a non-deterministic finite automata (NFA) based approach for shared processing of XPath expressions. A trie-based data structure, called XTrie [8], was proposed to support filtering of complex twigs. YFilter and XTrie decompose twig patterns into linear paths and match them individually. Twig patterns are matched by post-processing linear path matches. Our system FiST, supports *holistic matching* of twig patterns by transforming twig patterns and incoming documents into Prüfer sequences with inherent support for ordered pattern matching. Note that both FiST and YFilter use a runtime stack. FiST uses the stack to store the tags along the path from the current tag being processed to the root of the document. The size of the stack is bound by the depth of the document. On the other hand, YFilter uses the stack to track the active and previously processed states during the execution of the NFA. Note that many states in the NFA can be active simultaneously.

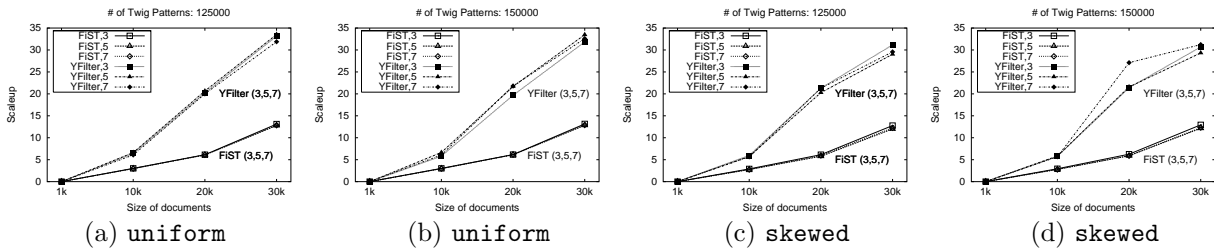There has been work on filtering using automata with

Figure 12: Varying XML Document Sizes

buffers. XSM [5] adopted a transducer based approach and used a subset of XQuery as a query language. To handle a subset of XQuery properly, the authors introduced the use of internal buffers. XPush [3] proposed the use of a modified deterministic pushdown automaton to simulate the execution of XPath filters and can handle predicates. XSQ [10] system handles multiple predicates, closures, and aggregations by using a hierarchical network of pushdown transducers augmented with buffers. Bruno *et al.* studied index-based and navigation-based XML multi-query processing [16] and showed both techniques have their own advantages. More recently, Tian *et al.* proposed the use of a relational database system for XML-based publish/subscribe system [11]. The XML filtering problem is turned into a join query that evaluates both the value predicate part and the tree structure part of the pattern which are both stored in relational tables.

7 Conclusion

In this paper, we have presented a novel XML filtering system called FiST. FiST performs *holistic matching* of twig patterns with incoming XML documents unlike the previous systems that rely on matching linear paths and merging the results of linear path matches. For this purpose, XML twig patterns (or user profiles) and XML documents are transformed into Prüfer sequences. Our filtering algorithm involves a *progressive subsequence matching* phase followed by a *refinement phase* to discard false matches. Furthermore, FiST provides ordered twig pattern matching for applications that require the nodes in the twig patterns to follow the document order in XML. Our experimental studies showed that FiST outperformed the state-of-the-art YFilter system by yielding better scalability under various situations.

Acknowledgments

References

[1] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th VLDB Conference*, pages 53–64, Cairo, Egypt, September 2000.

[2] Apache. Apache Xerces C++ Parser. http://xml.apache.org/xerces-c/.

[3] Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the 2003 ACM-SIGMOD Conference*, pages 419–430, San Diego, CA, June 2003. ACM Press.

[4] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jrme Simon. XML path language (XPath) 2.0 W3C working draft 16. Technical Report WD-xpath20-20020816, World Wide Web Consortium, August 2002.

[5] Bertram Ludscher, Pratik Mukhopadhyay and Yannis Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of the 28th VLDB Conference*, pages 227–238, Hong Kong, China, August 2002.

[6] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM-SIGMOD Conference*, Madison, Wisconsin, June 2002.

[7] Baden Hughes Catherine Bow and Steven Bird. Towards a General Model of Interlinear Text. In *Proceedings of EMELD Workshop*, Lansing, MI, July 2003.

[8] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis and Rajeev Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the 18th IEEE International Conference on Data Engineering*, pages 235–244, San Jose, CA, February 2002.

[9] Angel Luis Diaz and Douglas Lovell. XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator, September 1999.

[10] Feng Peng and Sudarshan S. Chawathe. XPath queries on streaming data. In *Proceedings of the 2003 ACM-SIGMOD Conference*, pages 431–442, San Diego, CA, June 2003. ACM Press.

[11] Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr and Jussi Myllymaki. Implementing a Scalable XML Publish/Subscribe System Using a Relational Database System. In *Proceedings of the 2004 ACM-SIGMOD Conference*, pages 479–490, Paris, France, June 2004.

[12] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv für Mathematik und Physik*, 27:142–144, 1918.

[13] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th VLDB Conference*, pages 361–370, Rome, Italy, September 2001.

[14] David Megginson. Simple API for XML. http://sax.sourceforge.net/.

[15] Karim Müller. Semi-Automatic Construction of a Question Treebank. In *Proceedings of the 4th International Conference on Language Resources and Evaluation*, Lisbon, Portugal, 2004.

[16] Nicolas Bruno, Luis Gravano, Nick Koudas and Divesh Srivastava. Navigation- vs. Index-Based XML Multi-Query Processing. In *Proceedings of the 19th IEEE International Conference on Data Engineering*, pages 139–150, Bangalore, India, March 2003.

[17] Praveen R. Rao and Bongki Moon. PRIX: Indexing and Querying XML Using Prüfer Sequences. In *Proceedings of the 20th IEEE International Conference on Data Engineering*, pages 288–299, Boston, MA, March 2004.

[18] William D. Lewis. Personal communications. http://zimmer.csufresno.edu/~wlewis/.

[19] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.