Fast XML Document Filtering by Sequencing Twig Patterns

JOONHO KWON
Advanced Institutes of Convergence Technology
PRAVEEN RAO
University of Missouri-Kansas City
BONGKI MOON
University of Arizona
and
SUKHO LEE
Seoul National University

XML-enabled publish-subscribe (pub-sub) systems have emerged as an increasingly important tool for e-commerce and Internet applications. In a typical pub-sub system, subscribed users specify their interests in a profile expressed in the XPath language. Each new data content is then matched against the user profiles so that the content is delivered only to the interested subscribers. As the number of subscribed users and their profiles can grow very large, the scalability of the service is critical to the success of pub-sub systems. In this article, we propose a novel scalable filtering system called iFiST that transforms user profiles of a twig pattern expressed in XPath into sequences using the Prüfer's method. Consequently, instead of breaking a twig pattern into multiple linear paths and matching them separately, iFiST performs *holistic matching* of twig patterns with each incoming document in a *bottom-up* fashion. iFiST organizes the sequences into a dynamic hash-based index for efficient filtering, and exploits the commonality among user profiles to enable shared processing during the filtering phase. We demonstrate that the holistic matching approach reduces filtering cost and memory consumption, thereby improving the scalability of iFiST.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information filtering*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness); user profiles and alert services*; I.7.0 [**Document and Text Processing**]: General

Authors' addresses: J. Kwon, Advanced Institutes of Convergence Technology, Gyeonggi-do 443-270, Korea; email: joonhokwon@gmail.com; P. Rao, Department of Computer Science Electrical Engineering, University of Missouri-Kansas City, Kansas City, MO 64110; email: raopr@umkc.edu; B. Moon, Department of Computer Science, University of Arizona, AZ, 85721; email: bkmoon@cs.arizona.edu; S. Lee, School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea; email: shlee@snu.ac.kr.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1533-5399/2009/09-ART13 $10.00
DOI 10.1145/1592446.1592447 http://doi.acm.org/10.1145/1592446.1592447

General Terms: Algorithms, Performance

Additional Key Words and Phrases: XML filtering, selective dissemination of information, twig pattern, Prüfer sequences

ACM Reference Format:

Kwon, J., Rao, P., Moon, B., and Lee, S. 2009. Fast XML document filtering by sequencing twig patterns. ACM Trans. Internet Technol., 9, 4, Article 13 (September 2009), 51 pages.
DOI = 10.1145/1592446.1592447 http://doi.acm.org/10.1145/1592446.1592447

1. INTRODUCTION

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. Pub-sub systems 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 items to deliver against a potentially large number of subscribed users.

Nowadays, we come across many e-commerce sites that provide email notifications to subscribers about price changes and hot deals. For example, a recent service by `Google`, called *Google Alerts*, provides email updates of the latest news based on topics of choice to subscribed users. 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`. Evidently, there is a growing use and demand for large-scale systems for selective information dissemination.

The popularity of the XML (eXtensible Markup Language) as a standard for information exchange has triggered several research efforts to build scalable XML filtering systems, where subscribers' interests are stored in their profiles typically expressed in the XPath language [Berglund et al.]. For example, a path expression given in the XPath syntax `book[author//name="John"]/title` qualifies XML documents for delivery by checking the occurrence of a pattern composed of the four elements `book`, `author`, `name`, and `title`, and by checking a value-based selection predicate `name="John"` in an XML document. Depending on the presence of predicates in it (for example, `[author//name="John"]`), an XPath expression can be considered a linear path or a structure of a twig pattern.

Note that the problem of filtering XML documents is fundamentally 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. Informally, the filtering problem that we address in this article is stated as follows. *Given a set of XPath expressions, identify such XPath expressions that match each of input XML documents to deliver.* In an XML-based pub-sub system, each incoming XML document is examined against user

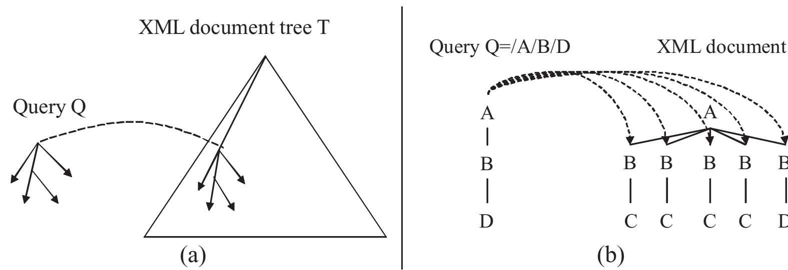


Fig. 1. (a) Top-down processing and (b) an example.

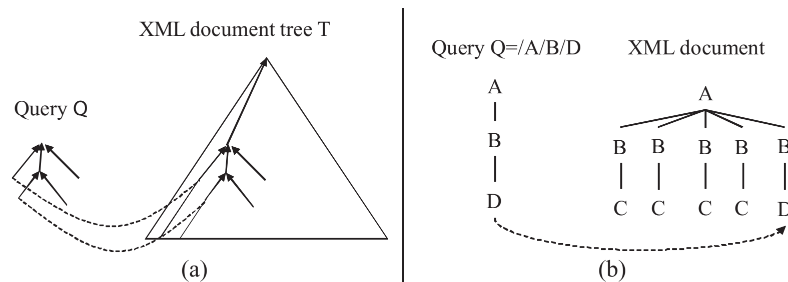


Fig. 2. (a) Bottom-up processing and (b) an example.

profiles represented by XPath expressions. The XML document is then delivered to users whose profiles were 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. For this purpose, various XML filtering systems have been proposed [Altinel and Franklin 2000; Diao et al. 2003; Ludäscher et al. 2002; Peng and Chawathe 2003; Gupta and Suciu 2003; Chan et al. 2002a; Bruno et al. 2003]. Typically, XPath queries are processed in a top-down fashion. XML publish-subscribe systems using a top-down approach evaluate XPath queries over XML documents from the root to a leaf, as shown in Figure 1(a). Figure 1(b) shows a scenario where the top-down approach is problematic. For simplicity, the XML document has five linear root-to-leaf paths with A as the root, and the XPath query contains only the parent-child relationships. The query is considered to have a match in the document after processing five linear root-to-leaf paths in the XML document.

A different perspective is to evaluate XPath queries in a bottom-up fashion. They are evaluated from a leaf to the root, as shown in Figure 2(a). The bottom-up approach benefits from the fact that selectivities of leaf nodes tend to be high. As shown in Figure 2(b), only the last leaf-to-root path in the document is processed during the filtering.

In this article, we propose a novel filtering system called *iFiST* (improved FiST [Kwon et al. 2005]) that performs *holistic matching* of *twig patterns* with each incoming XML document in a *bottom-up* manner. We do not address the problem of efficiently delivering XML documents once the interested users are identified.

The idea of holistic processing of twig patterns is not new. Bruno et al. [2002] proposed a holistic twig join algorithm called *TwigStack* for finding all occurrences of a query pattern in an XML document. TwigStack is a holistic approach because a query twig pattern is not decomposed into multiple root-to-leaf linear paths and it processes each path individually to evaluate the results. PRIX [Rao and Moon 2004] also adopts a holistic approach by sequencing XML documents and query patterns. Holistic processing avoids the generation of unnecessary intermediate results and improves performance. In an XML filtering system, the notion of holistic processing can be defined similarly.

Definition 1. A filtering system supports holistic processing of user profiles/twig patterns if: (1) twig patterns are not decomposed into multiple root-to-leaf paths and processed individually, and (2) there is no merging of intermediate results (generated by matching paths individually) to compute the final answers.

Our filtering system iFiST matches user profiles in a holistic fashion by matching a twig pattern as a whole unit during filtering rather than decomposing it into linear paths. Our system adopts the idea of encoding XML documents and user profiles into Prüfer sequences. Prüfer's method provides a one-to-one correspondence between labeled trees and sequences [Prüfer 1918]. It has been shown in the PRIX system [Rao and Moon 2004, 2006] that the tree-to-sequence encoding supports efficient twig pattern matching. A collection of sequences from user profiles are organized into a dynamic hash-based index, so that XML documents are filtered against the user profiles efficiently during the two basic phases: subsequence matching followed by refinement. In the subsequence matching phase, a superset of user profiles are identified that potentially match an incoming document. In the refinement phase, false matches are discarded by performing postprocessing for branch nodes in the twig patterns. Note that the encoding of user profiles and the processing of XML documents are done in a bottom-up manner (from leaf elements to the root element).

To further improve the efficiency and scalability, iFiST identifies user profiles with similar interests and optimizes the filtering process by enabling shared processing of user profiles with common patterns. As a result, both the memory requirements and filtering time of iFiST can be reduced. Our extensive experimental study shows that the holistic matching approach enables iFiST to outperform the state-of-the-art YFilter system by achieving superior scalability, particularly when user profiles contain complex XPath expressions and XML documents are heavily recursive and deep.

The iFiST system focuses on *ordered* twig pattern matching, which is essential for applications that are sensitive to the order between elements in XML documents. XML documents are typically modeled as ordered labeled trees. (Common XPath expressions do not specify the order between sibling nodes, although the XPath standard provides a facility to do so.) Furthermore, each input XML document is processed by the SAX Parser [Megginson] element-by-element in document order.

Consider an XML pub-sub system for music objects [Chiu and Hsu 2006] based on the MusicXML [MusicXML]. A user submits an XPath query which

expresses a sequence of notes such as "D F# A D A F# D" that he/she is interested in. In this case, his/her profile must be matched to only those MusicXML documents containing the exact order of note sequences specified by the user. Another example for ordered XML processing can be drawn from the linguistics domain. In a linguistic data model proposed by Bow et al. [2003], interlinear texts are represented in the XML format to analyze various linguistic principles in different languages. Due to the requirements of linguistic analysis, it is essential to preserve order between the words in the text [Lewis, personal communication], and there is a compelling need for ordered twig pattern matching. Besides, language treebanks have been widely used in computational linguistics because 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 [2004] used ordered pattern matching over treebanks for question answering systems. In this work, we focus on efficient and scalable filtering of ordered twig patterns. Unordered twig pattern matching will be investigated in the future.

Our iFiST system is an extension of FiST [Kwon et al. 2005] and aims to further improve the filtering performance by exploiting commonality in user profiles. Thus iFiST performs shared processing of user profiles to reduce the filtering time and memory consumption. Similar to FiST, iFiST has two phases, namely, progressive subsequence matching followed by a refinement phase for branch node verification. Like FiST, iFiST is free from false positives or false dismissals. Our iFiST system requires additional data structures to support shared processing. Further, it supports insertion and deletion of user profiles from the system.

The key contributions of this work are summarized as follows.

— iFiST exploits commonality among user profiles, and supports shared processing during filtering. As a result, both the storage cost for indexing user profiles and the filtering time can be reduced.
— Since iFiST is an extension of FiST, it also supports holistic matching of twig patterns against incoming XML documents in a bottom-up fashion and provides ordered twig matching for applications that require the nodes in a twig pattern to follow the document order in XML.
— We provide a comprehensive solution for processing "*" wildcard. Leaf and regular wildcards are processed by adding the wildcard information to our data structures and checking this information during subsequence matching. Branch wildcards are processed by modifying the stack comparison operation during subsequence matching.
— We report results from a comprehensive set of experiments carried out to evaluate the scalability of iFiST in comparison with YFilter with datasets having different characteristics. The effectiveness of shared processing of iFiST is evaluated in terms of filtering cost and memory consumption with and without shared processing. We also compared the memory consumption of iFiST with YFilter.

The remainder of this article is organized as follows. A survey of related work is presented in Section 2. We present the overview of the iFiST system in Section 3. In Section 4, we describe the basic filtering algorithm in iFiST. In Section 5, we present optimizations to our filtering algorithm. Section 6 discusses the processing of wildcards in user profiles. Section 7 discusses our experimental results. We conclude our work in Section 8.

2. RELATED WORK

In this section, we first review previous research on publish-subscribe systems and then describe prior work on XML filtering systems.

2.1 Publish-Subscribe Systems

A variety of publish-subscribe systems have been developed. They can be broadly classified into three categories: (1) topic-based systems, (2) predicate-based systems, and (3) XML-based systems.

A number of topic-based systems have been proposed [Castro et al. 2002; Ramasubramanian et al. 2006; Milo et al. 2007]. In these systems, publishers and subscribers are connected together by a predefined topic. Scribe [Castro et al. 2002] is a standard topic-based system for managing topics, topic-clusters, and user subscriptions. Corona [Ramasubramanian et al. 2006] is a topic-based system for detecting and disseminating Web page updates. The Tamara [Milo et al. 2007] system minimizes the maintenance overhead for topics during the dissemination of events based on a distributed clustering algorithm. Our system is more flexible because users are allowed to specify their own interests as opposed to being restricted by predefined topics.

A number of predicate-based pub-sub systems have been developed. In the content-based network [Carzaniga et al. 2004], an input document is structured as a set of attribute/value pairs and a user profile is expressed as *a disjunction of conjunctions* of constraints on attributes. As compared to an XML-based system, this system allows a limited form of expressing user interests. Chandramouli et al. [2007] proposed techniques for scalable processing and dissemination of a large number of subscriptions with value-based notification conditions. However, the subscriptions track the value of the same data item over time and their system does not deal with XML.

Much research has been done in XML-based publish-subscribe systems [Diao et al. 2004; Chan and Ni 2007; Hong et al. 2007]. The ONYX [Diao et al. 2004] system is a large-scale dissemination system that delivers XML messages based on user profiles. A piggyback optimization [Chan and Ni 2007] was proposed for optimizing the performance of content-based dissemination of XML data. However, only a subset of XPath expressions that contain parent-child ("/") and ancestor-descendant ("//") axes are supported. The Massive Multi-Query Join Processing (MMQJP) technique [Hong et al. 2007] was proposed for dealing with a large number of interdocument queries. Interdocument queries join different XML documents based on the values in their nodes, either attributes or text. They proposed the XML Stream Conjunctive Language (XSCL) which consists of three clauses, namely, SELECT, FROM, and PUBLISH.

Delivering a matched document to a group of interested users is an important problem in a publish-subscribe system. Much research has been conducted in this regard [Carzaniga et al. 2004; Shah et al. 2004; Fenner and Srivastava 2005; Rao et al. 2007; Li et al. 2008; Miliaraki et al. 2008]. In the content-based network [Carzaniga et al. 2004], packets are forwarded based on their data contents rather than IP addresses. Multicast allows a source to send the same content to multiple receivers. CBM [Shah et al. 2004] is a content-based multicast scheme to reduce network bandwidth usage and delivery delay by considering semantics of the information in the multicast process. Overlay networks provide application-layer routing. ONYX [Diao et al. 2004] is based on an overlay network and uses a source-based tree for publishing new data. XTreeNet [Fenner and Srivastava 2005] is also based on the overlay network and similar to ONYX. The main difference is that intermediate routers in XTreeNet only forward data items. NetX [Rao et al. 2007] adopts a novel XML indexing scheme on top of a distributed hash table to implement a content-based matching. The advertisement-based routing algorithm for optimizing content-based routing for XML contents was recently proposed [Li et al. 2008]. However, only linear XPath expressions are handled in this work. Miliaraki et al. [2008] proposed a distributed implementation of YFilter [Diao et al. 2003] on top of Distributed Hash Tables (DHTs) for XML data dissemination.

The matching process is an essential step to deliver the right content to users. Thus we focus on the matching process in this work.

2.2 XML Filtering 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. The proposed approaches can be broadly classified into three categories: (1) automaton-based approaches, (2) index-based approaches, and (3) others.

Most previous approaches were based on constructing automaton representations for the user profiles. XFilter [Altinel and Franklin 2000] is one of the early works on XML filtering. XFilter takes linear XPath expressions and transforms each expression into a single Finite State Machine (FSM). The collection of FSMs are indexed to support efficient filtering. YFilter [Diao et al. 2003] is an extension of XFilter and adopts a nondeterministic Finite Automata (NFA)-based approach to improve the scalability of the filtering system by promoting shared processing of XPath expressions. YFilter handles XPath expressions of a twig pattern by decomposing them into individual linear paths, matching the linear paths individually, and then performing postprocessing over matches from the linear paths. For example, 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`. A postprocessing phase is used to check whether an entire query expression has been matched or not. YFilter uses NFA to reduce the number of automaton states, whereas the lazy DFA [Green et al. 2003, 2004] approach uses DFA, which is constructed lazily to reduce active states for deep and recursive XML

data. To improve the cache performance as well as overall performance of the XML filtering system, the cache-conscious automata model was studied [He et al. 2005, 2006]. XML documents are processed in a top-down fashion from the root element to a leaf element.

There has been work on XML filtering using automata with buffers. XSM [Ludäscher et al. 2002] adopted a transducer-based approach and used a subset of XQuery as the query language. To handle a subset of XQuery properly, the authors introduced the use of internal buffers. However, one of the disadvantages is that XSM does not support the "//" axis. XPush [Gupta and Suciu 2003] proposed the use of a modified deterministic pushdown automaton to simulate the execution of XPath filters and can handle predicates. A drawback of the approach is the difficulty in adding or deleting queries from the constructed automaton. XSQ [Peng and Chawathe 2003] handles multiple predicates, closures, and aggregations by using a hierarchical network of pushdown transducers augmented with buffers. However, XSQ evaluates only one XPath expression at a time.

A trie-based data structure, called XTrie [Chan et al. 2002b], was proposed to support filtering of complex twig patterns. XTrie consists of two components: (1) a trie constructed from a set of distinct substrings and (2) a substring-table for storing information about each substring in a twig pattern. XTrie breaks a twig into several substrings if "//" or "*" appear. XTrie's substrings contain only "/" axis. XTrie will have a large number of fragments when "//" or "*" occur frequently in the twig patterns. In addition, XTrie is not a holistic approach according to Definition 1. Bruno et al. [2003] studied index-based and navigation-based XML multiquery processing and showed both techniques have their own advantages. However, their work considers only a set of simple path expressions which do not contain predicates.

In the final category, there are several approaches. Tian et al. [2004] proposed the use of a relational database system for an XML-based publish/subscribe system. 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. The main problem with this method is that the number of joins increases with the size of query. Gong et al. [2005] conducted research on a Bloom filter-based XML filtering system. An XML query is taken as query string and all query strings are mapped into a Bloom filter by hash functions. However, their work does not consider a twig pattern query. More recently, a predicate-based filtering [Hou and Jacobsen 2006], AFilter [Candan et al. 2006], and BoXFilter [Moro et al. 2007] were proposed. The predicate-based filtering system [Hou and Jacobsen 2006] encodes XPath expressions as ordered sets of predicates such that the common parts among XPath expressions will be stored in a predicate index. AFilter [Candan et al. 2006] can exploit prefix and suffix commonalities in the set of XPath expressions. However, it does not support twig pattern queries. BoXFilter [Moro et al. 2007] is similar to our work and it also uses bottom-up approach based on sequencing the twig patterns using Prüfer sequences. It introduces the idea of early pruning by grouping sequences into envelopes.

```
Path             :=   RelLocationPath | AbsLocationPath
AbsLocationPath  :=   '/' RelLocationPath
RelLocationPath  :=   Step '/' RelLocationPath | Step
Step             :=   Axis NodeTest | Step '[' Predicate ']'
Axis             :=   '/' | '//' | '@'
NodeTest         :=   String | *
Predicate        :=   Expression | Expression '=' Expression
Expression       :=   String | Path
```

Fig. 3. Grammar of XPath subset.

Several XPath streaming engines have been proposed [Bar-Yossef et al. 2004, 2005; Chen et al. 2006b]. XSM and XSQ can be regarded as streaming engines. Bar-Yossef et al. [2004, 2005] investigated the space complexity of XPath evaluation on XML streams and provided space lower bounds of XPath evaluation without wildcards when buffering is required. TwigM [Chen et al. 2006b] uses a stack representation of data to achieve a complexity which is polynomial in the size of the data and query. However, these approaches focus on processing a single XPath expression at a time.

Our iFiST system is a holistic XML filtering system and uses a sequence representation for twig patterns. These sequence representations are indexed for efficient filtering. Further, shared regions of sequences are identified and processed together to reduce the storage cost and filtering time.

3. OVERVIEW OF THE IFIST SYSTEM

In this section, we briefly describe the XPath language and the basic data model of XML data for the iFiST system. Then we formally present the filtering problem that we address in this article. We also provide an architectural overview of the iFiST system and briefly describe its core components. We then describe the construction of Prüfer sequences for XML document trees.

3.1 XPath and Data Model

The XPath language [Berglund et al.] defines expressions for addressing parts of an XML document and is also used as a building block for XSLT [Clark 1999] and XQuery [Boag et al.] languages. The iFiST system uses a subset of XPath to express user profiles. Figure 3 shows this subset of XPath handled in this work. The subset contains elements, attributes, wildcards, and child and descendant axis. Evaluating value-based predicates in user profiles during filtering has been addressed in other papers of ours [Kwon et al. 2007, 2008].

XML documents are typically modeled as ordered labeled trees. For example, the XML document in Figure 4(a) can be represented as an ordered labeled tree as shown in Figure 4(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 article, attributes are treated the same

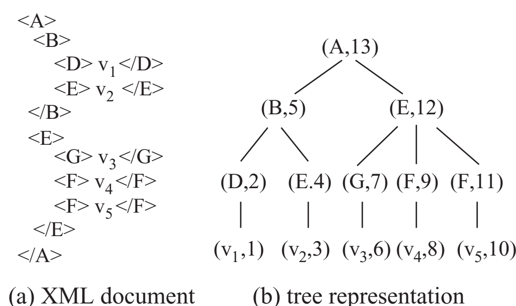


Fig. 4. A sample XML document.

way as elements. Hence, no special distinction will be made between elements and attributes in the subsequent discussions.

3.2 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 (for example, XISS [Li and Moon 2001], TwigStack [Bruno et al. 2002], PRIX [Rao and Moon 2006, 2004]). However, in XML filtering, the roles 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 an input document to be filtered. Note that the idea of reversed roles of queries and data is based on the SITF system [Yan and Garcia-Molina 1999]. Formally the problem of XML document filtering can be stated as follows.

> Given a set Q of XPath queries and an input XML document D, find a subset $Q' \subseteq Q$ such that D contains one or more matching occurrences of q for any $q \in Q'$.

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

3.3 Architectural Overview

The architectural overview of iFiST is shown in Figure 5. The core filtering engine is shown in a dotted box. User profiles expressed in XPath are parsed by an XPath parser and converted into Profile sequences based on the Prüfer's method. (See Section 3.4 for the description of the Prüfer sequence construction.) The collection of sequences are stored in a hash-based dynamic index called the *sequence index*. iFiST exploits commonality among user profiles by examining the profile sequences for similar regions. Similar regions of user profiles are identified and stored in an optimized way: The profile sequences are stored in a *segment table*. We will delve into the details pertaining to the index construction and maintenance in Section 4. User profiles can be updated during the execution of the filtering engine.

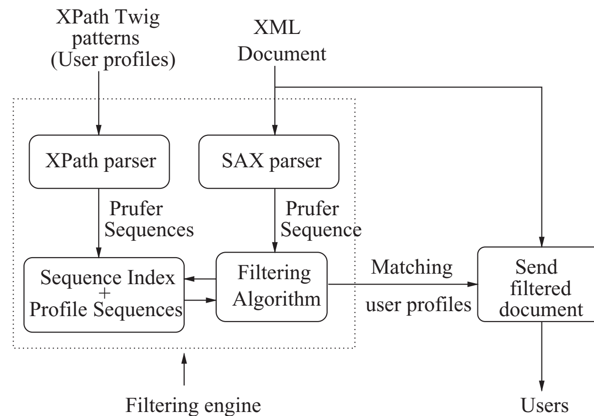


Fig. 5. Architecture overview.

Incoming XML documents that need to be filtered are first parsed using a SAX parser [Megginson]. 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 iFiST system, we shall move on to explain the Prüfer sequence construction.

3.4 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. 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 $(a_1, a_2, a_3, \ldots, a_{n-2})$ is called the Prüfer sequence of tree T_n. From the sequence $(a_1, a_2, a_3, \ldots, a_{n-2})$, 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 [Rao and Moon 2004, 2006], we construct a Prüfer sequence of length $n - 1$ for T_n by continuing the deletion of nodes until 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 [Rao and Moon 2004]. 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 the LPS. The following example illustrates the sequence representation for an XML document tree.

Example 1. Consider the XML document tree in Figure 4(b). The nodes of the tree are labeled in postorder. The Prüfer sequence of the tree using the node

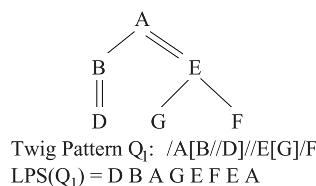


Fig. 6. Twig pattern to sequence conversion.

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.

The iFiST system adopts the idea of encoding XML documents and user profiles into Prüfer sequences. The use of Prüfer sequences for indexing and querying XML data has been studied and evaluated in the PRIX system [Rao and Moon 2004, 2006]. Prüfer's method provides both the iFiST and PRIX systems with a mechanism that transforms XML documents and queries into sequences. However, since iFiST performs filtering against queries while a document is being parsed, the sequence representation of user profiles is somewhat different from the sequence representation adopted by PRIX. (See Section 4.)

4. INDEX STRUCTURE AND FILTERING ALGORITHM

With the high-level overview of the iFiST system presented in the previous section, we now describe the index structure and the basic filtering algorithm of iFiST. We will then present a few optimizations to speed up the filtering process in the following sections. Hereinafter, we will use the terms "user profiles" and "twig patterns" interchangeably.

4.1 Transforming User Profiles into Sequences

In the iFiST system, user profiles expressed in XPath are transformed into Prüfer sequences. The twig patterns we deal with have either a parent-child relationship ("/") or ancestor-descendant ("//") relationship between two nodes. This section describes the method to map twig patterns into sequences. For now, let us consider patterns without a wildcard "*". Later, in Section 6, we will describe how the wildcard can be handled.

When a twig pattern is mapped to a sequence, both "/" and "//" axes in the twig pattern are treated as a regular tree edge with no distinction between them. For example, consider the tree representation of a twig pattern Q_1 in Figure 6. If Q_1 is mapped to a sequence, its extended LPS will be D B A G E F E A. Then, such information as the type of an axis between nodes (either parent-child or ancestor-descendant) and branch node is associated with each node in the sequence. In particular, the node relationship information is stored in child (or descendant) nodes. This sequence of nodes annotated with such information is called a *profile sequence*. (See Figure 7(a) for an illustration.)

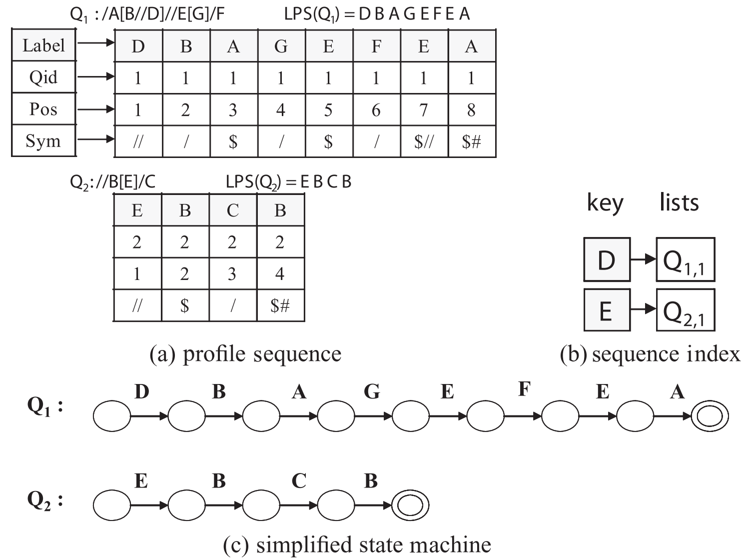


Fig. 7. Profile sequence, sequence index, and state machine.

Table I. Node Information

Attribute	Description
Label	a Prüfer sequence label (i.e., XML tag)
Qid	a unique identifier of a user profile
Pos	an ordinal number that represents the position of a node in the profile sequence
Sym	a set of values that describe the type of a sequence node: '/' or '//', '$' for a branch node, '#' for a root node

Each node in the profile sequence has four attributes, namely **Label**, **Qid**, **Pos**, and **Sym**. These attributes are summarized in Table I. 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 the table. Given a node q in the profile sequence, the four attributes are denoted by q_{Label}, q_{Qid}, q_{Pos}, and q_{Sym}, respectively.

Example 2. Figure 7(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 of 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 E 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 an 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 that the number of user profiles that need to be matched against incoming XML documents can presumably grow very large, it is critical that a good indexing strategy be developed for efficient and scalable filtering. In this section, we propose an index structure to store the profile sequences.

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

THEOREM 1 [RAO AND MOON 2004]. *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 7(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 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 *sequence index*.

The sequence index is built over profile sequences. The **Label** of a sequence node is used as a key in the hash table. With each key in the index, a list of nodes from profile sequences that need to be matched next is associated. At the parsing time, the front nodes of the profile sequences are added to the sequence index. Both the key and nodes in the lists of a key are updated dynamically during the subsequence matching phase. When a new Prüfer sequence label of an input XML document is generated during the subsequence matching phase, the nodes in the list corresponding to the label are examined to carry out necessary state transitions. On a successful state transition, the next nodes in the corresponding sequences cause the updates of the sequence index. First, we check whether the label of next node is existed in the sequence index. If the label is existed as a key, the next node is just copied to the lists in the sequence index on the label. Otherwise, the new key is generated in the sequence index using the label and the next node is the first node of the lists on the key.

In a typical pub-sub environment, it is natural for several users to share similar interests. Identifying similar profiles can help us reduce the document filtering cost and storage cost for indexing user profiles. For ease of exposition, we first describe the data structures and the basic filtering algorithm in iFiST without any sharing among user profiles. Later, in Section 5.2, we build on the

Algorithm 1. SAX Handlers

```
stack S;                                                  /* a runtime global stack */
procedure StartTagHandler(tag)
1: S.push(tag)
end
procedure EndTagHandler(tag)
2: if tag is a leaf node then
3:     FindSubsequence (S.top());              /* for extended Prüfer sequences */
   endif
4: S.pop();
5: FindSubsequence (S.top());
end
```

basic filtering algorithm and extend the data structures to support shared processing in iFiST.

Example 3. A sequence index is shown in Figure 7(b). 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 Basic Filtering Algorithm

The basic filtering algorithm carries out *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, the 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 input documents.

4.3.1 *Progressive Subsequence Matching.* It is essential that we find the subsequence matches simultaneously for all the profile sequences in a scalable manner. We call the subsequence matching *progressive* because we generate the sequence representation of the document incrementally and find matching profile sequences in steps. The LPS of an XML document is constructed incrementally by examining the runtime global stack as the document is being parsed. To parse input XML documents, iFiST uses a SAX parser with a few modifications made to the `StartTagHandler` and `EndTagHandler` procedures. Algorithm 1 shows the `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. If it is a leaf node, the top element of the stack is used as the next Prüfer sequence label and the

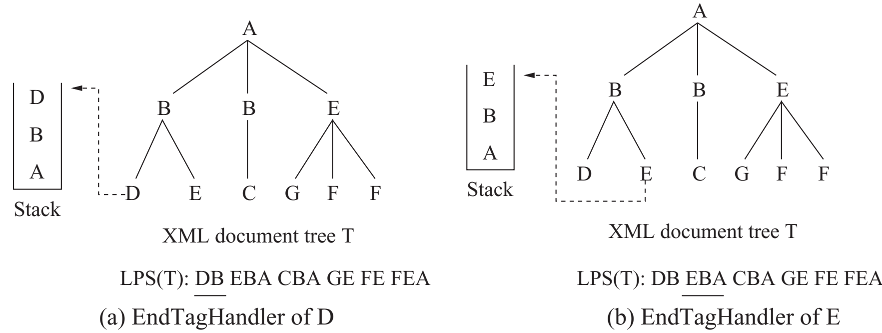


Fig. 8. Generation of LPS(T).

filtering procedure (`FindSubsequence(·)`) is invoked, because a leaf node is considered to have a dummy child node for the purpose of producing an extended LPS instead of a regular LPS (lines 2 to 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).

Example 4. We illustrate the construction of the LPS of the XML document tree T as shown in Figure 8. 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 8(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 8(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 preceding process is repeated until 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. The filtering algorithm relies on the sequence index to find all matching subsequences simultaneously for all the profile sequences. Note that the elements of an input XML document are popped out of the runtime stack by `EndTagHandler` in the same order as they appear in the LPS of the document. As a result, it is feasible to generate the LPS of an input XML document incrementally just by scanning the document only once without actually storing the entire document in memory.

Algorithm 2. Progressive Subsequence Matching

```
Input:  {L} - L is a Prüfer sequence label;
procedure FindSubsequence(L)
1:  CurrentList ← SequenceIndex[L];
2:  foreach SequenceNode q in CurrentList do
3:  |  test ← 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 ← true;
    |  |  |  /* Branch node */
8:  |  |  |  case '$': doBranch (q);
    |  |  |  /* Root node of twig pattern */
9:  |  |  |  case '#':
    |  |  |    BranchNodeVerification (q_Qid);
    |  |  endsw
    |  endfch
10: |  if ((q_Sym = '/' or q_Sym = '//') and (test = true)) or (q_Sym = '$') then
11: |  |  q' ← NextNode(q);
12: |  |   copy q' to Sequence Index using key q'_Label;
    |  endif
    endfch
end
```

During the subsequence matching phase, iFiST performs additional tests to eliminate most false matches by using the runtime stack. The runtime stack allows parent-child and ancestor-descendant relationships to be tested during this phase. (Other benefits of the stack will be presented in Section 5.) In essence, transitions occur in a state machine (e.g., Figure 7(c)) when the tag name is matched and the stack test succeeds. The core filtering operations are shown in Algorithm 2. The procedure `FindSubsequence`$(\cdot)$ is invoked from `EndTagHandler`$(\cdot)$. Using the label L as a 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 through 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 Axes. The runtime stack is used to test parent-child and ancestor-descendant relationships for a pair of document elements that match the nodes in the profile sequences.

Let `TestPC`$(\cdot)$ (parent-child) and `TestAD`$(\cdot)$ (ancestor-descendant) refer to these two tests. These tests 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 can move to the next state.

For example, in Figure 9, when `EndTagHandler` is called for E, the state of the stack is shown on the left. The jth element in the profile sequence for Q_i

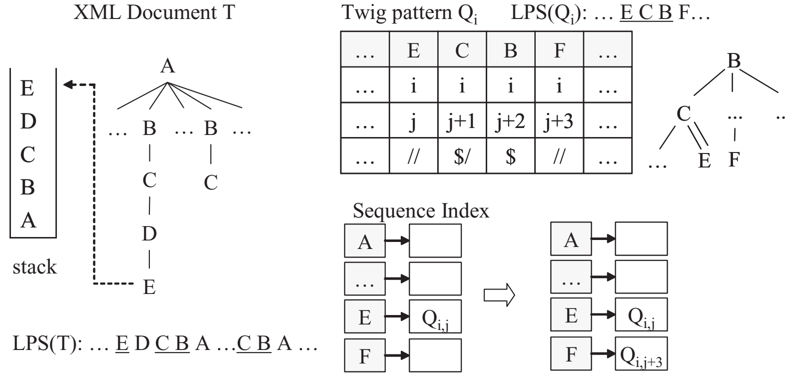


Fig. 9. Testing parent-child and ancestor-descendant relationships.

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)

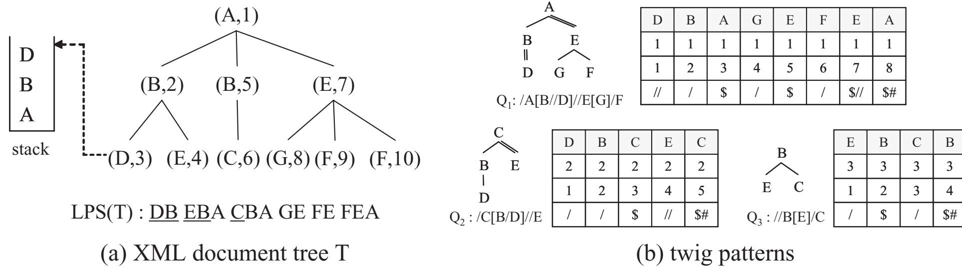
1: $q' \leftarrow$ `NextNode`(q);
2: **if** *(v = '/' and* `TestPC`*(q, q') is successful) OR*
(v = '//' and `TestAD`*(q, q') is successful)* **then**
3: **return** true;
4: **else return** false;

---

matches the top of the stack (i.e., the *Label* of $Q_{i,j}$ is E). Since the *Sym* 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 9 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 previously described operations (line 6). The state of the sequence index after matching $Q_{i,j+2}$ is shown in Figure 9.

Note that the twig pattern in Figure 9 can match anywhere in the incoming document except when the root node of a document is involved. If node B is 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 it is true, then this implies that node B in the twig pattern matches the document root.

4.3.2 *Branch Node Processing*. It has been shown that filtering by subsequence matching alone can lead to false positives [Rao and Moon 2004]. To eliminate such false matches, iFiST takes refinement steps by testing the connectedness property for branch nodes in twig patterns. We begin with an example to



D	B	A	G	E	F	E	A
1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
//	/	$	/	$	/	$//	$#

D	B	C	E	C
2	2	2	2	2
1	2	3	4	5
/	/	$	//	$#

E	B	C	B
3	3	3	3
1	2	3	4
/	$	/	$#

Fig. 10. Progressive subsequence matching.

motivate the need for special branch node processing to support the refinement phase.

For convenience, assume that the elements in each XML document tree are numbered in preorder.[1] Annotating the elements in an XML document with the preorder numbers can be done by the SAX parser in a straightforward manner by maintaining and incrementing a counter on every call to `StartTagHandler`. The counter value is then assigned to the tag being processed.

Example 5. Consider the example in Figure 10. The XML document tree T is numbered in preorder (see Figure 10(a)). Figure 10(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, namely, (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, namely, (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, namely, (A,1). Note that Q_1 has a match in T.

iFiST pays additional attention to branch nodes in the profile sequences in order to facilitate the refinement phase to discard false matches. The main task of the branch node processing is to keep track of matching element tags in an input XML document. This information about these elements is used in the refinement phase to check the connectedness property. A data structure called *BranchID set* is maintained for each occurrence of a branch node in the profile sequence to keep track of the preorder number of the element in the document that matches the sequence node.

During the subsequence matching phase, if the profile sequence node, say q, is a branch node, then `doBranch`(q) stores the preorder number of the element

[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.

in the document with label q_{Label} (that matches q) in the *BranchID set* for q. In general, the number of times a given node appears in a Prüfer sequence is determined by the number of its child nodes [Rao and Moon 2004]. Thus, a profile sequence can have two different types of branch node occurrences for any given branch node in a twig pattern: an *internal branch* and a *final branch*. For example, in Figure 10(b), the profile sequence nodes $Q_{1,5}$ 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 a node as an internal branch node. 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.

On the other hand, the last occurrence of a branch node always has either a parent-child or ancestor-descendant relationship with the following node. For example, node $Q_{1,7}$ in Figure 10(b) has an ancestor-descendant relationship with $Q_{1,8}$. We refer to such a node in a profile sequence as a final branch node. 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 does the filtering algorithm examine the next node for subsequence matching.

In Figure 10(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 *Refinement by Branch Node Verification.* 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.

When the root node of a profile sequence is processed, then the BranchID sets that are constructed during the subsequence matching phase are examined. Algorithm 4 shows the steps involved. 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 nonempty 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 10(b). The intersection of the BranchID sets for node A is {1}. Similarly, the intersection of the BranchID sets for node E is {7}. As a result, Algorithm 4 reports Q_1 as a match, since Q has a true match in T.

YFilter performs postprocessing to check whether an entire twig pattern has been matched or not, since a twig pattern is decomposed into several linear paths and matched separately first. iFiST uses the postprocessing to discard false positives since branch node testing cannot be performed during progressive subsequence matching.

Algorithm 4. Branch Node Verification

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

```
procedure BranchNodeVerification (q_Qid)
1: test ← true;
2: L_B ← list of labels of the branch nodes in the twig pattern with id Qid;
3: foreach l in L_B do
4:    n ← 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 ⋂_{i=0}^{n} B_li = ∅ then test ← false;
   endfch
7: if test = true then report q_Qid as a match;
```

5. OPTIMIZATIONS

This section presents optimization strategies to speed up the filtering process and reduce the memory requirements. The first optimization exploits the runtime stack to avoid frequent node copying, and enables early elimination of subsequences that do not yield a match. The second optimization introduces the notion of shared processing of profile sequences. By exploiting similarity among profile sequences, this technique allows to avoid redundant processing during filtering, to reduce memory consumption.

5.1 Exploiting the Runtime Stack

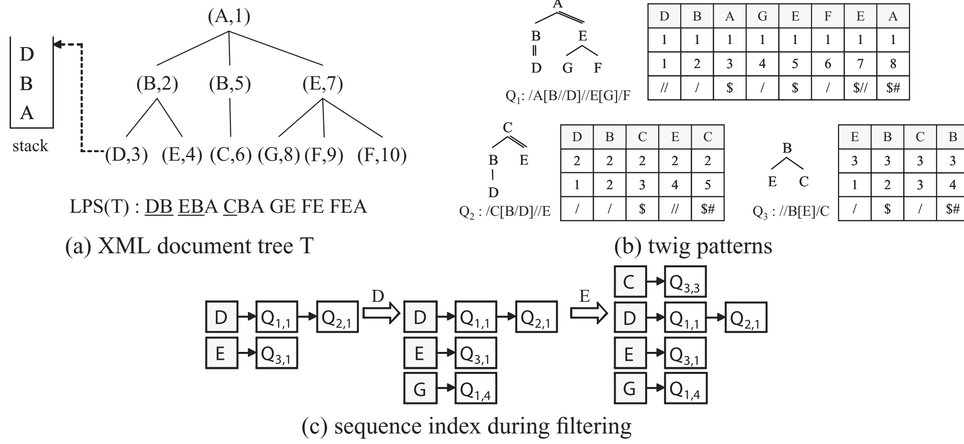
Avoiding Frequent Node Copy to Sequence Index Let us again consider the example in Figure 9. Based on Algorithm 1, `FindSubsequence(·)` is invoked each time the `EndTagHandler` is called. In the example, when the `EndTagHandler` for leaf E is called, the set of elements in the stack represent a segment of the LPS(T), namely, E D C B A. Note that the node A is a branch node. A naive way is to invoke `FindSubsequence(·)` 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 up to the branch node A, we can use it as a look-ahead buffer. Thus, instead of performing `doSimpleStackTest(·)` for each tag, an iterative stack check can be performed for a group of contiguous tags in the sequence, thereby avoiding copying nodes in the profile sequence up to the branch node. The filtering algorithm aggressively tries to find subsequence matches up to an internal branch node. When a final branch node is encountered, the stack checking resumes. The `doSimpleStackTest` is replaced by `doRecursiveStackTest` (Algorithm 5) that performs the stack checking. 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 nodes in the twig pattern.

Algorithm 5. 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' ← 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 '$': doBranch (q');
7:             case '#':
                   BranchNodeVerification (q'_Qid);
           endsw
       endfch
8:     if q_Sym = '$' then
9:         copy q' into Sequence Index using key q'_Label;
       endif
   endif
```

To incorporate our optimized stack testing, a modification to Algorithm 2 is done by replacing the procedure `doSimpleStackTest` with `doRecursive -StackTest` on line 7 and by omitting lines 10 through 12. In Algorithm 5, on a successful stack test (line 2), the next node of q in the profile sequence, that is, 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. When the iterative stack check succeeds, 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 5, if a branch node in the profile sequence has a "/" or "//" relationship with the next node, then the subsequence matching continues by invoking `doRecursiveStackTest`$(\cdot)$.

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 9. For document T, LPS(T) = $\cdots$ E B C B A $\cdots$ C B A $\cdots$. For twig pattern Q_i, LPS(Q_i) = $\cdots$ E C B $\cdots$. LPS(T) has two subsequence instances that match "E C B". (They are underlined in Figure 9.) In the XML document, the second element C (leaf node in T) and its parent, namely, 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.



(a) XML document tree T (b) twig patterns

(c) sequence index during filtering

Fig. 11. Benefits of the runtime stack.

Shortly we illustrate the execution of our filtering algorithm with an XML document T in Figure 11(a) and twig patterns Q_1, Q_2, and Q_3 in Figure 11(b). The nodes $Q_{1,1}$, $Q_{2,1}$, and $Q_{3,1}$ are initially stored in the sequence index shown in Figure 11(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. We skip the branch detection techniques for this example.

Example 6. When `FindSubsequence`(D) is invoked, the state of the runtime stack is shown in Figure 11(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 the iterative stack test from $Q_{1,2}$ is invoked. 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. Since the label of $Q_{1,4}$ is not existing in the sequence index, the hash key G is created and the $Q_{1,4}$ is the first node of lists in the 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 the iterative stack test from $Q_{2,2}$ 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 11(c).

Next, when `FindSubsequence`(B) is invoked, there is no hash key in the sequence index for the label 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 by creating the hash key C and adding it to the lists on the key. Next `FindSubsequence`(B) is

invoked again, there are is no key in the sequence index for the label 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.

Differences with Other Stack-Based Pattern Matching Techniques. For the task of XML pattern matching, several stack-based twig join algorithms have been proposed [Bruno et al. 2002; Lu et al. 2004; Chen et al. 2006a]. The main difference between these stack-based algorithms and iFiST is in the number of stacks. (In iFiST, only one stack is maintained.) In TwigStack [Bruno et al. 2002], each element node in a query has a stack. TwigStack is I/O and CPU optimal for queries with ancestor-descendant relationships between nodes. In TwigStackList [Lu et al. 2004], each element node in a query has a combination of a stack and list and queries with parent-child relationships are efficiently handled. In Twig^{2}Stack [Chen et al. 2006a], each element node in a query has a hierarchical stack which consists of an ordered sequence of stack trees. Generalized tree pattern queries are handled by this algorithm.

Recently, an index-based filtering technique called BoXFilter was proposed [Moro et al. 2007]. The BoXFilter adopts Prüfer sequence-based representation for user profiles which are then organized into a height balanced tree structure. Two stacks are used during the filtering process. However, BoXFilter does not support queries with wildcards. Like iFiST, unordered matching is not supported in BoXFilter.

Our iFiST system uses only one runtime global stack for temporarily storing elements in an input document during filtering. The size of this runtime stack is bound by the height of the input document tree.

5.2 Shared Processing of Profile Sequences

In a typical publish-subscribe environment, it is common that several users share similar interests. Thus, identifying similar profiles presents another opportunity to further reduce the filtering cost for finding matching user profiles. In this section, we present strategies to further improve the performance of iFiST by avoiding redundant processing of profile sequences during the filtering phase. This is achieved by identifying *segments* (i.e., a contiguous set of sequence nodes) of profile sequences that are common between different user profiles. As a result, these common segments can be processed just once.

5.2.1 *Shareable Segments.* It is essential to first determine the granularity of segments that can be shared so that sharing is effective and can be conveniently incorporated into iFiST. The sequence representation of user profiles and the left-to-right processing of profile sequence nodes necessitates a rather different approach from an approach such as *path sharing* that YFilter [Diao et al. 2003] uses. YFilter stores root-to-leaf paths of a twig pattern in a single nondeterministic automaton, the sharing starts from the root node of a twig query and proceeds until no more sharing is possible. During sequence

construction of user profiles by Prüfer's method, the node labels in a twig pattern from a leaf node to its immediate ancestor branch node appear contiguously in the profile sequence. Also the node labels from a branch node to its immediate ancestor branch node appear contiguously. We define *shareable* segments in a profile sequence as follows.

Definition 2. Shareable segments of a profile sequence are defined as segments of a profile sequence that can be shared and are generated progressively as follows. Starting from the leftmost sequence node, each node in the sequence is examined to determine if it marks the end of a shareable segment.[2] The next shareable segment begins from the sequence node following it. For each node in the profile sequence:

(1) if a sequence node is an internal branch node, then this node marks the end of a shareable segment, or
(2) if a sequence node is the last node (i.e., root node) of the profile sequence, then this node marks the end of a shareable segment.

In Section 4.3.2, we introduced two kinds of branch nodes in a profile sequence: an internal branch node and a final branch node. The internal branch node and the next node of internal sequence node (usually it is a leaf node) have no relationship. Thus we can mark the internal branch node as the end of a shareable segment. On the contrary, the final branch node and the next node of final branch node (it is not a leaf node) have a parent-child relationship or an ancestor-descendant relationship. Therefore, we do no mark the final branch node as the end of a shareable segment. The intuition behind choosing such shareable segments is to achieve effective sharing that is not restricted to paths that can be shared only from the root (e.g., YFilter), as well as ease of incorporating it into our existing iFiST framework.

Remark 1. Each shareable segment in a profile sequence captures a set of edges in a twig pattern. Any pair of edge sets is disjoint (Definition 2).

Remark 2. Each shareable segment can be processed by one invocation of *FindSubsequence*(·) during filtering.

Based on Remarks 1 and 2, it is evident that by identifying shareable segments, redundant processing can be avoided during filtering and only one copy of a segment needs to be stored by iFiST. As a result, both filtering cost and storage cost can be potentially reduced.

Remark 3. A shareable segment does not guarantee the finest granularity of sharing among user profiles. Rather it provides a simple scheme to incorporate shared processing in iFiST.

Example 7. Consider a profile sequence for twig pattern Q_1 shown in Figure 12. By applying the steps described in Definition 2, three shareable segments with labels "DBA", "GE", and "FEA" are obtained. The 3rd and 5th

[2]Each segment is identified by two distinct nodes that mark the beginning and the end of it.

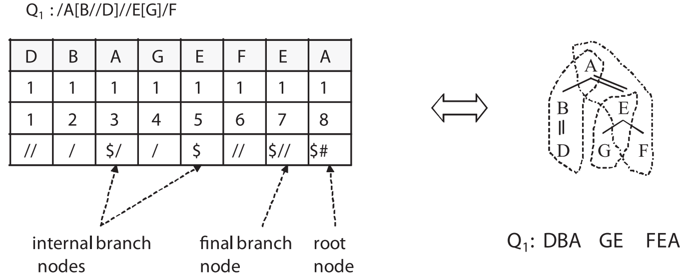


Fig. 12. Shareable segments of a twig pattern.

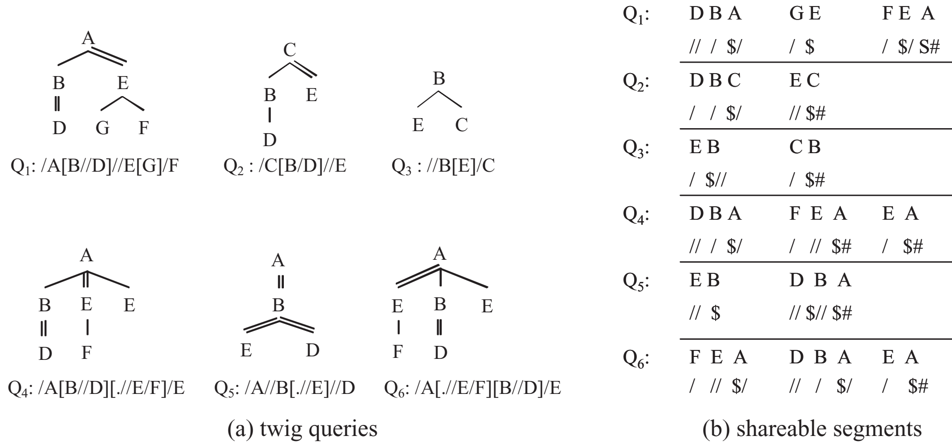


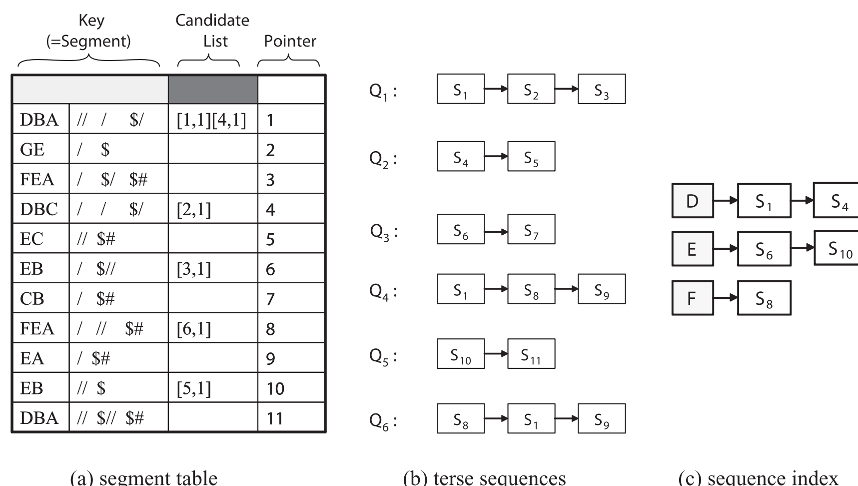
Fig. 13. Twig queries and their shareable segments.

nodes in the profile sequence are internal branch nodes and mark the end of their shareable segments, respectively. The 7th node in the profile sequence is the final branch node for "E" and it has a parent-child relationship with the 8th node, so it cannot be an end of the sharable segment. The 8th node in the profile sequence is the root node and marks the end of its shareable segment.

Each shareable segment is precisely represented by the *Label* attribute and *Sym* attribute of the profile sequence nodes. Figure 13 shows six twig patterns and their respective shareable segments.

5.2.2 *Indexing Shareable Segments.* In the following section, we shall describe data structures used to index profile sequences based on shareable segments. First, we shall describe how the profile sequences can be represented compactly and then show how the sequence index can be used to index these sequences.

The set of shareable segments are stored in a hash table segment table using a segment as the key. Each segment of a profile sequence can be uniquely identified by the profile sequence id and its position in the sequence. For each key (segment) in the segment table, the value is essentially a list of (sequence



Key (=Segment)		Candidate List	Pointer
DBA	// / $/	[1,1][4,1]	1
GE	/ $		2
FEA	/ $/ $#		3
DBC	/ / $/	[2,1]	4
EC	// $#		5
EB	/ $//	[3,1]	6
CB	/ $#		7
FEA	/ // $#	[6,1]	8
EA	/ $#		9
EB	// $	[5,1]	10
DBA	// $// $#		11

Fig. 14. Storing and indexing profile sequences compactly.

id, position) pairs. We call this list the candidate list. A candidate list is used to perform state transitions during filtering by storing current segments of twig patterns. Initially, each list in the segment table contains the (sequence id, position) pairs corresponding to the first segments of the profile sequences. Now each profile sequence is compactly represented as a terse sequence, which is a linked list storing the identifiers/pointers to the segments in the segment table. The original profile sequence can be reconstructed by scanning its terse sequence from left-to-right.

Example 8. For the set of six twig patterns in Figure 13(a), a segment table is shown in Figure 14(a). The profile sequences are represented by terse sequences which are shown in Figure 14(b). S_1 in a terse sequence refers to the segment in the first row of the segment table.[3] The candidate lists are initialized with the first segments of the profile sequences. For example, the candidate list in the first row of the segment table contains the first segments of profile sequences Q_1 and Q_4.

The process of updating the candidate lists during filtering, when a current segment is matched and its next segment becomes active, is similar to that of inserting a new profile sequence. However, in this case, the candidate lists are updated irrespective of whether the segment is the first or not.

The sequence index indexes the terse sequences. In this index, the first label of a shareable segment is used as a key and the value is a list of pointers to entries in the segment table. The sequence index is initialized as follows. For each terse sequence, we insert a key-value pair (first label of first segment, pointer to the first segment in the segment table) into the sequence index.

[3]For convenience, a subscript number is used to represent a pointer to the row of the segment table. Note that the *Pointer* column is not actually maintained in the segment table and is shown only for illustration.

Example 9. The sequence index in Figure 14(c) is initialized with the terse sequences in Figure 14(b). The first segments of Q_1 and Q_4 are common and the first label is "D". Hence for hash key "D" in the sequence index, a pointer (shown by "S_1") to the first key in the segment table is stored in the value list. In addition, the first segment of Q_2 has "D" as its first label and hence a pointer to the segment table is stored in the value list (shown by "S_4"). Similarly, the value lists for keys "E" and "F" are initialized too.

During filtering, the index contains a set of segments that need to be matched against an incoming document. The sequence index stores only one instance of a shareable segment and hence processes it only once during filtering.

Next we describe how a new profile sequence is added to the segment table. Given a profile sequence, for each segment S we do the following two steps: (1) If S is absent in the table, we insert it. (2) If S is the first segment of the profile sequence, the candidate list corresponding to S is updated with the profile sequence id and its position.

5.2.3 *Useful Observations.* The iFiST system decomposes the sequenced representation of a twig pattern into several segments. This process helps in identifying shareable parts of twigs. Once a segment is matched, the next segment is considered for matching. Thus conceptually each profile sequence is still processed from left-to-right, without breaking it into root-to-leaf paths. Thus iFiST processes twig patterns in a holistic fashion.

Remark 4. Definition 1 holds for iFiST under shared processing using segments.

When a twig pattern is added to the system, YFilter decomposes it into several root-to-leaf linear paths and inserts each of them into the automata. During filtering, all linear paths of the twig pattern are independently processed for matches. Once the entire document is parsed, a merging step is performed to obtain the final matches.

Another difference between YFilter and iFiST is the way twig patterns are shared and processed. In YFilter, the sharing of linear paths occurs from the root node in a user profile (top-down in nature). In iFiST, the sharing occurs from a leaf node to a branch node in a user profile (bottom-up in nature). For example, consider the twig pattern Q_1 and Q_4 in Figure 13. In iFiST, these patterns share the first segment "DBA", which is processed only once. In YFilter, these patterns share the path "/A/B//D", which is also processed only once. Thus, when the selectivity of element `D` is higher than element `A`, it is particularly advantageous to process bottom-up.

XTrie [Chan et al. 2002b] decomposes XPath expressions into sequences of XML element names (i.e., substrings). XTrie breaks a twig if "//" or "*" appears and can only process fragments with "/" axis as a unit. However, XTrie is not a holistic approach. XTrie will have a large number of fragments when "//" or "*" occur frequently in the twig patterns. As an example, consider the twig pattern $Q = /a[b//d]//e[g][.//f]/*/*/c//h$ containing two branch nodes (element `a` and `e`) and four root-to-leaf linear paths. XTrie decomposes Q into

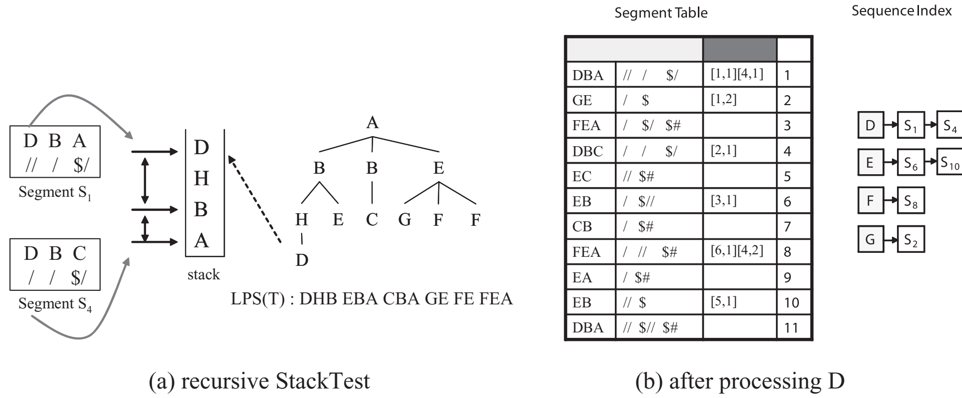


Fig. 15. When FindSubsequence(D) is invoked.

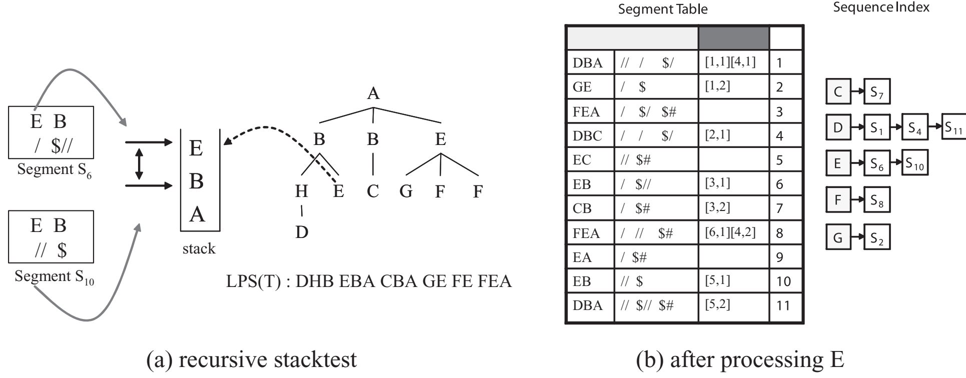
six fragments : $f_1 = /a/b$, $f_2 = //d$, $f_3 = //e/g$, $f_4 = //f$, $f_5 = /*/*/c$, and $f_6 = //h$. On the other hand, iFiST creates four segments for Q and is a holistic approach.

5.3 Optimized Shared Filtering with Stack

In this section, we describe the progressive subsequence matching phase after incorporating the two optimizations described in Section 5.1 and Section 5.2. Since each shareable segment in the segment table contains both the *Label* attribute and the *Sym* attribute of a profile sequence node, the process of checking the runtime stack can be done as before. The following is an example that illustrates the process of subsequence matching.

Example 10. Consider an XML document tree T shown in Figure 15(a) and the segment table/sequence index shown in Figure 15(b). The initial states of the segment table/sequence index are shown in Figure 14(a) and 14(c), respectively.

When `FindSubsequence(D)` is invoked (due to the end tag of D), the state of the stack is shown in Figure 15(a). Using the pointers "1" and "4" in the value list corresponding to label "D" in the sequence index, we fetch segments S_1 and S_4 from the segment table. The recursive stack test checks segment S_1 against the entries in the runtime stack. S_1 passes the stack test. The candidate list corresponding to S_1 contains pairs [1,1] and [4,1]. The next segments of Q_1 and Q_4 need to be copied into the filtering data structures. Using terse sequences, we can identify the next segments of Q_1 and Q_4, namely [1,2] and [4,2]. These are added to the candidate lists of S_2 and S_8 in the segment table. Further, the pointers to S_2 and S_8 are added into the sequence index. The new hash key "G" is created in the sequence index when inserting the pointer S_2. The inserting S_8 is skipped because it it already existed in the hash key "F". S_4 does not pass the stack test since element "C" is absent in the stack and the filtering data structures are unchanged. The states of the segment table/sequence index are shown in Figure 15(b).



(a) recursive stacktest (b) after processing E

Fig. 16. When FindSubsequence(E) is invoked.

When `FindSubsequence(E)` is invoked, the elements in the runtime stack are shown in Figure 16(a). There are two pointers (e.g., 6 and 10) in hash key "E" of the sequence index. Both segments S_6 and S_{10} pass the stack test and pairs [3,2] and [5,2] are added into the candidate list of the segment table. The pointer "7" (second segment of Q_3) is copied into the value list of hash key "C" after creating the hash key "C" in the sequence index and pointer "11" (second segment of Q_5) is copied into the value list of hash key "D". This is shown in Figure 16(b).

The core operations during filtering with shared processing are shown in Algorithm 6. The operation *doRecursiveStackTest* (line 2) is similar to lines 2 through 9 in Algorithm 2. As explained in Section 4.3.2, the branch node information is maintained in the BranchID sets during the recursive StackTest. To speed up this StackTest, the segment table, sequence index, and the terse sequence are used during subsequence matching. Because different profile sequences can share the same segment in the segment table, the recursive StackTest is invoked only once.

Algorithm 6. Progressive Subsequence Matching with Stack Optimization

Input: {L} - L is a Prüfer sequence label;

procedure **FindSubsequence**(L)
1: **for** *each Segment s in SequenceIndex[L]* **do**
2: **if** *doRecursiveStackTest(s) is true* **then**
3: **for** *each pair* $[seqId, segPos]$ *in Candidate List of s* **do**
4: let seg be the $(segPos + 1)_{th}$ segment of Q_{seqId} ;
5: let $segL$ be the first Prüfer sequence label of seg;
6: Add the pair $[seqId, segPos + 1]$ into the Candidate List of seg;
7: Copy seg into the $SequenceIndex[segL]$;
endfor
endif
endfor
end

6. WILDCARD PROCESSING

Wildcards(*) in XPath are commonly used when element names are unknown or do not matter. They are also useful as a shorthand notation to represent a set of element names. However, it can be expensive to process wildcards if they are compared with all the elements in the runtime stack during the stack test. To reduce the cost of processing wildcards in the iFiST system, we remove the wildcard nodes from the profile sequence and store them as wildcard information denoted in the nonwildcard sequence nodes. The attached wildcard information is used to check the distance of elements in the runtime stack during subsequence matching.

There are three types of wildcard nodes in twig patterns: (1) a leaf wildcard node which appears as a leaf node of a twig pattern; (2) a branch wildcard node which appears as a branch node of a twig pattern; and (3) a regular wildcard node which is not a leaf wildcard node nor a branch wildcard node. The iFiST system handles wildcard nodes differently according to their types.

6.1 Processing Leaf and Regular Wildcards

In this subsection, we explain how to process the regular wildcard nodes and leaf wildcard nodes.

Capturing Wildcard Information. Each wildcard node is denoted by q_δ and contains a 2-tuple (*op*, *extra*), where:

—*op* denotes relationships between wildcard and nonwildcard nodes.
—*extra* is the value of an additional distance for stack check.

To fully capture the information for wildcards, we maintain three values denoted by *op*, N_*, and I_R as we traverse from a leaf node to the root node in a user profile. [4] The variable *op* denotes either a parent-child or an ancestor-descendant relation. The variable *op* is initialized to "=" and it is changed to "≥" whenever we meet an ancestor-descendant relationship in the leaf-to-root path.

To compute *extra*, we use two values N_* and I_R. The value of *extra* is the sum of N_* and I_R. This is because we need to differentiate the leaf wildcard nodes from regular wildcard nodes as explained earlier. N_* is an integer that represents the number of wildcard nodes. N_* can be computed as follows: The value N_* is initialized to 0. The value N_* is increased as we meet the wildcard node, and initialized again after it is saved into q_δ. I_R is an integer that records a regular wildcard node. The value I_R is initially set to 1 and changed to 0 when we meet a regular wildcard node. It is initialized again after it is saved into q_δ.

For example, given a user profile Q_j=/A/*/B//*/*/C/*, we can compute wildcard information at nodes A, B, and C as shown in Figure 17. LPS(Q_j) is shown and the values for N_* and I_R are also depicted. The underlined values at nonwildcard nodes are used to obtain the value for *extra* in q_δ. (1) At the first

[4] The Prüfer sequence of a user profile is generated in a bottom-up manner.

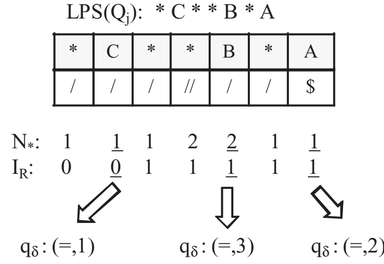


Fig. 17. Computing wildcard information.

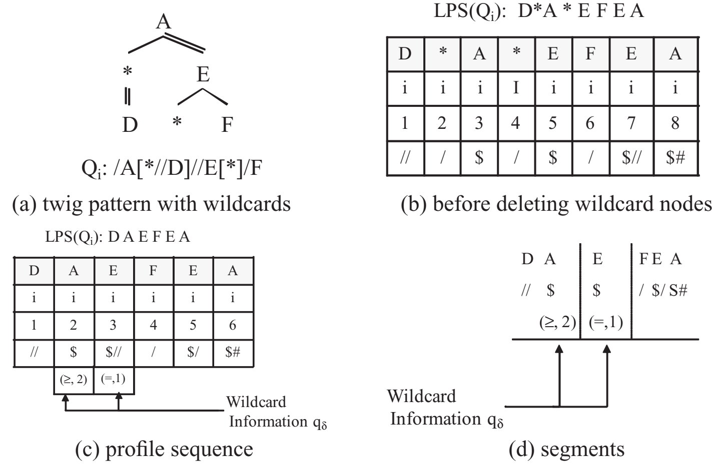
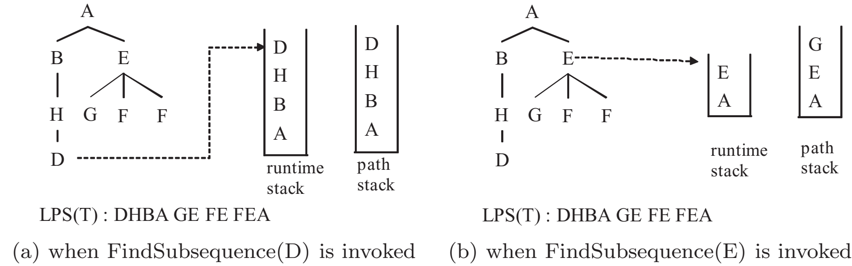


Fig. 18. Handling twig patterns with wildcard "*".

wildcard node in LPS(Q_j), N_* is increased to 1 and the value of I_R is set to 0 because it is a leaf wildcard node. Thus, wildcard information is represented by a tuple ("=", 1) at node "C" because the label "C" is not a wildcard. Then, the values of N_*, I_R, and *op* are reset to their initial values. (2) At node "B", N_* is the value of 2 because two wildcards appear after the node "C". The *op* has a value of "≥" since we meet the ancestor-descendant relationship from the node "C" to the node "B". Thus the wildcard information of the node B is represented by a tuple "(≥, 3)". Then, N_*, I_R, and *op* have their initial values. (3) At the node "A", we obtain wildcard information as "(=, 2)" since N_* is the value of 1, I_R is the value of 1, and we only meet the parent-child relationship.

Note that if there appears only parent-child relationship("/") in the root-to-leaf path, q_δ represents as an exact value (equality). Otherwise q_δ is represented as a range value (inequality). The q_δ attribute is stored with the parent node of a wildcard node in the leaf-to-root path, which is the next node of the wildcard node in the profile sequence.

Example 11. The twig pattern Q_i in Figure 18(a) has two wildcard nodes: a regular wildcard node which is an ancestor of node D and a leaf wildcard node which is a child of node E. At node "A" in the leaf-to-root path /A/*//D, δ is



(a) when FindSubsequence(D) is invoked (b) when FindSubsequence(E) is invoked

Fig. 19. Two stacks for wildcard processing.

"$(\geq, 2)$". This is because N_* has a value of 2 after resetting 1 at node "D" and *op* is "$\geq$" after meeting the ancestor-descendant relationship. Similarly, at node "E" in the leaf-to-root path /A//E/*, δ is "(=, 1)".

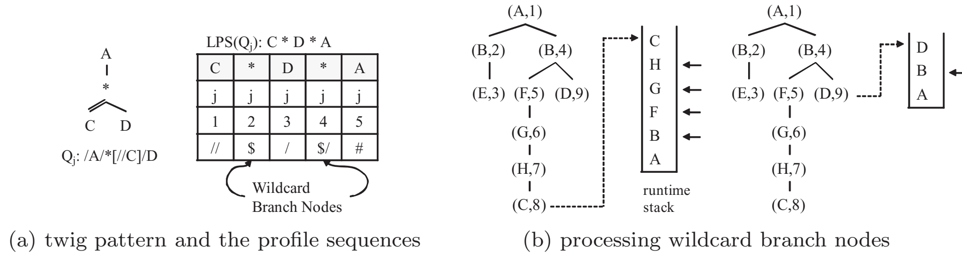
Figure 18(b) shows the profile sequence before deleting wildcard nodes. The computed values of δ are used as wildcard information in Figure 18(c). The 2nd node, which represents the parent node of a regular wildcard node, has "$(\geq, 2)$" as the δ attribute. The 3rd node, the parent of a leaf wildcard node, has "(=, 1)" as its δ attribute. Similarly, the wildcard information is added into the segments as shown in Figure 18(d). The first and second segments have this wildcard information.

Wildcards Processing during Subsequence Matching. Since a profile sequence and its segments for a user profile are constructed in a bottom-up fashion, the parent node of a wildcard node in a twig pattern stores its wildcard information in the sequence. The attached wildcard information is checked during subsequence matching. When we process sequence nodes or segments which contain wildcard information during the stack test, their wildcard information is also checked.

The overhead for processing wildcard nodes is that we need another stack to keep track of all elements on a leaf-to-root path in an XML document. We refer to this stack as path stack. The elements of the runtime stack are added and deleted according to start tag events and end tag events of the input XML document, whereas the elements of the path stack are added due to start tag events and updated only when the leaf node of the XML document has changed. The path stack is used only to check the wildcard information for leaf wildcard nodes. The following example explains how two stacks are used to check the wildcard information.

Example 12. Figure 19 shows the difference between the two stacks when we parse an XML document. When `FindSubsequence(D)` is invoked, elements of the runtime stack and the path stack are the same. But, when `FindSubsequence(E)` is invoked, the elements of two stacks are different. The runtime stack stores all ancestor nodes and the node E. The path stack stores all nodes of the leaf-to-root path which contains node E.

Assume we are given the query in Figure 18(a) and its profile sequence in Figure 18(c). In `FindSubsequence(D)`, we process the 1st node and 2nd node in Figure 18(c). The wildcard information "$(\geq, 2)$" will be checked. Since the level



(a) twig pattern and the profile sequences (b) processing wildcard branch nodes

Fig. 20. Branch wildcard nodes processing.

difference of a node A and a node D in the runtime stack is more than 2, the stack check will succeed. Then, the next node of the branch node is copied to the sequence index. In `FindSubsequence(E)`, the 3rd node is processed. The wildcard information "(=, 1)" is an exact value, which means the deleted wildcard node is a leaf wildcard node. Thus we use the path stack to check whether there are some nodes above the node E. The 3rd node is passed the check because there is a node G above the node node E. Similarly, we can process the segments with wildcard information.

6.2 Processing Branch Wildcards

Since branch nodes in a twig pattern are used to eliminate the false matches, if we eliminate the branch wildcard nodes in a twig pattern we may have false matches due to the lack of information. Thus, in the case of branch wildcard nodes we treat the wildcards in the same way of processing the typical elements in a twig pattern. In other words, the sequence nodes for branch wildcard nodes are also generated.

The difference between handling the sequence nodes for elements and the sequence nodes for branch wildcards occurs only at the branch node processing during the subsequence matching. When we compared the sequence node to the elements of runtime global stack, the sequence node for an element is matched to only one element in the stack. However, the sequence node for a branch wildcard node can be matched to several elements in the stack. For this reason, the preorder numbers of all possible matches in the stack are stored in the BranchID set for branch wildcard nodes. The steps for computing intersection of the BranchID sets remains unchanged. The overhead for processing branch wildcard nodes is the increased number of members of BranchID set.

Example 13. A twig pattern Q_j in Figure 20(a) shows two branch wildcard nodes in its profile sequence, which are the 2nd and the 4th nodes, respectively. When `FindSubsequence(C)` is invoked, which is shown the left in Figure 20(b), there are six elements in the stack. The second node is a branch wildcard node having an ancestor-descendant relationship with the first node. This makes the possible matches to be elements below the label "C" except the root label "A" in the stack. The four preorder numbers of matched elements are stored in the BranchID set. As the same way, a preorder number of the possible match is stored in the BranchID set When `FindSubsequence(D)` is invoked.

Table II. Characteristics of DTDs

	No. of Distinct Tags	Max Depth of Document
Auction	77	10
NITF	123	10
Treebank	250	20

Table III. Parameters for Synthetic User Profiles

Parameter	Description	Values
N_u	Number of user profiles	50,000 to 150,000
L	Maximum depth of a user profile	6 to 10
$p_{//}$	Probability of having a descendant axis	0.0 to 0.8
z	Skewness of element name distribution	0.0 to 0.9
N_b	Number of branches in a user profile	1 to 7

7. EXPERIMENTS

We evaluate the efficiency and scalability of iFiST and YFilter under various operational conditions, namely, by varying the number of user profiles, by varying the document sizes to filter, and by varying the number of branches in user profiles. We also evaluate the performance impact of the optimization techniques presented in Section 5 by varying the degree of duplicate user profiles. The memory consumption of iFiST is also evaluated.

7.1 Experimental Setup

In our experiments, we used three different datasets: `Auction`, `NITF`, and `Treebank`. `Auction` is a synthetic benchmark dataset from the XMark Project [XMark]. For `NITF` and `Treebank`, we generated the datasets from the NITF and Treebank DTDs using IBM's XML Generator [Diaz and Lovell 1999]. `NITF` (News Industry Text Format) is an XML-based DTD designed for the markup and delivery of news content [NITF]. `Treebank` is an XML-based encoding format for the representation of linguistic corpora [Treebank].

In each dataset, documents were grouped by their sizes in bytes. In subsequent discussions, these document groups will be referred to as "`1k`", "`5k`", "`10k`", "`20k`", and "`30k`." Each document group contained 1,000 XML documents, and all the reported experimental results were averaged over the entire set of documents.

Table II shows the number of distinct tags in the DTDs and the maximum depth of the documents. Auction and NITF are moderately deep, while Treebank is deeper and has many recursions in elements.

To generate user profiles expressed in the XPath language, we used the XPath generator from the YFilter package [Diao et al. 2003]. Table III lists the parameters and their value ranges used to generate user profiles as workload for indexing and filtering. Note that z is a Zipfian skew parameter used for determining the distribution of element names. If z is set to zero, the element names will be uniformly distributed. Otherwise, the distribution of element names will be skewed.

Evaluation Metrics

We have evaluated the iFiST system in three main aspects of scalability with respect to: (1) a varying number of user profiles, (2) a varying number of branches in the user profiles of twig pattern, and (3) a varying size of documents to filter. In Section 7.2, to demonstrate the performance benefits of iFiST over the current state-of-the-art, we compare iFiST with the YFilter system [Diao et al. 2003] by showing the trend of the filtering cost in the three aspects. In Sections 7.4 and 7.5, we present the effect of duplicate user profiles and the memory consumption.

When iFiST was compared with YFilter, their performance trends were measured in *scaleup* for fair comparison. This is because YFilter (obtained from the University of California at Berkeley) is implemented in Java, while iFiST is implemented in C++ with Xerces XML Parser version 2.5.0 [Apache].

Whenever filtering cost was measured, it was averaged over a given set of documents to filter per each set of user profiles. Note that the filtering cost was the sum of document parsing time and time taken by the filtering algorithm. The parsing time was very small compared to the filtering time. For example, the average ratio of the parsing time to the filtering time consumed by iFiST for dataset 20k was 0.006.

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

7.2 Scalability

In this section, we analyze the performance of iFiST and YFilter in terms of scaleup. Due to the space limitations, we present the scalability trend only for a few representative cases of document sizes, the number of user profiles, and the number of branches.

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

$$scaleup = \frac{tAvg - tAvg_{base}}{x - x_{base}} \tag{1}$$

where $tAvg$ is the filtering time measured for the case under observation at x and $tAvg_{base}$ is the filtering time measured for the base case x_{base}. We assume that the x-axis grows in unit steps for all aspects of scalability. Depending on the type of scalability being measured, the $tAvg_{base}$ is the filtering time for either the smallest number of user profiles, the smallest number of branches, or the smallest size of input document. More specifically, if scalability is evaluated by varying the number of twig patterns that are indexed, if 50,000 is the minimum number of indexed twig patterns, then $tAvg_{base}$ is the filtering time measured for the twig set of 50,000 user profiles. On the x-axis, although the actual number of user profiles can increase in steps of 25,000 profiles, we increase x in the aforesaid formula by one. This kind of normalizes the scaleup value across different aspects of scalability. Thus, a positive (or negative) measurement of scaleup indicates that the filtering cost increases (or decreases) as the scale of test cases grows.

Note that iFiST supports *ordered twig pattern* matches and YFilter supports *unordered twig pattern* matches. We removed twig patterns that yielded unordered matches so that YFilter and iFiST find the same number of matches and can be compared on a fair basis. YFilter needs a simple postprocessing step to support ordered matches since it is an unordered matching approach. We did not consider this postprocessing time for YFilter timings.

7.2.1 *Varying Number of User Profiles.* We first compared the scalability of iFiST and YFilter with Auction and Treebank datasets by varying number of user profiles up to 150,000. The number of twig patterns indexed by iFiST and YFilter was varied from 50,000 to 150,000 in steps of 25,000. Figure 21 summarizes the scaleup for iFiST and YFilter for uniform and skewed twig sets with 3 or 4 branches per a twig. The results for Treebank dataset `1k` are omitted since their trend was similar to that of `10k` for both iFiST and YFilter.

Let us analyze the results shown in Figure 21(a). The Auction DTD is used for uniform user profiles, and the number of branches in the user profiles was three. The average filtering time grew for both YFilter and iFiST as the number of twig patterns were increased. For Auction datasets `5k`, `10k`, and `20k`, the scaleup of YFilter was better than that of iFiST. This can be explained the following observation. Assume that three queries Q_1=/a/b, Q_2=/a//b and Q_3=/a/c are given to YFilter and iFiST. The first node "a" of three queries can be shared in YFilter's automata, whereas three queries are treated as three different segments in iFiST. YFilter has more chances for shared processing with increasing the number of user profiles. Similarly, with user profiles with 4 branches, the scaleup of YFilter was better than iFiST. A similar trend was observed in Figures 21(c) and (d) for the skewed datasets of Auction DTD.

Figures 21(e) and (f) show the *scaleup* for the uniform datasets of Treebank DTD. The average filtering time grew for both YFilter and iFiST as the number of twig patterns were increased. But, for Treebank datasets `10k`, `20k`, and `30k`, the scaleup of iFiST was better than that of YFilter. By design, YFilter's stack stores the active states in the NFA for backtracking. The number of states in YFilter's stack was greatly increased due to the deep and recursive structure of Treebank. As a result, the performance of YFilter suffered. A similar trend was observed in Figures 21(g) and (h) for the skewed datasets of Treebank DTD.

7.2.2 *Varying Number of Branches in User Profiles.* In this section, we compare the scalability of iFiST and YFilter with respect to the varying number of branches in the twig patterns. The results are shown in Figure 22.

Figures 22(a) and 22(b) show the *scaleup* for Auction dataset `10k` and dataset `20k` using the uniform twig set for YFilter and iFiST, respectively. The filtering cost of YFilter was the same or slightly decreased as the number of branches in the twig patterns increased from 1 to 5. The filtering time for iFiST decreased as the number of branches increased. iFiST had negative scaleup as shown in Figures 22(a) and 22(b). This trend was observed for all the twig set sizes that we used. The number of matched user profiles decreased when the number of

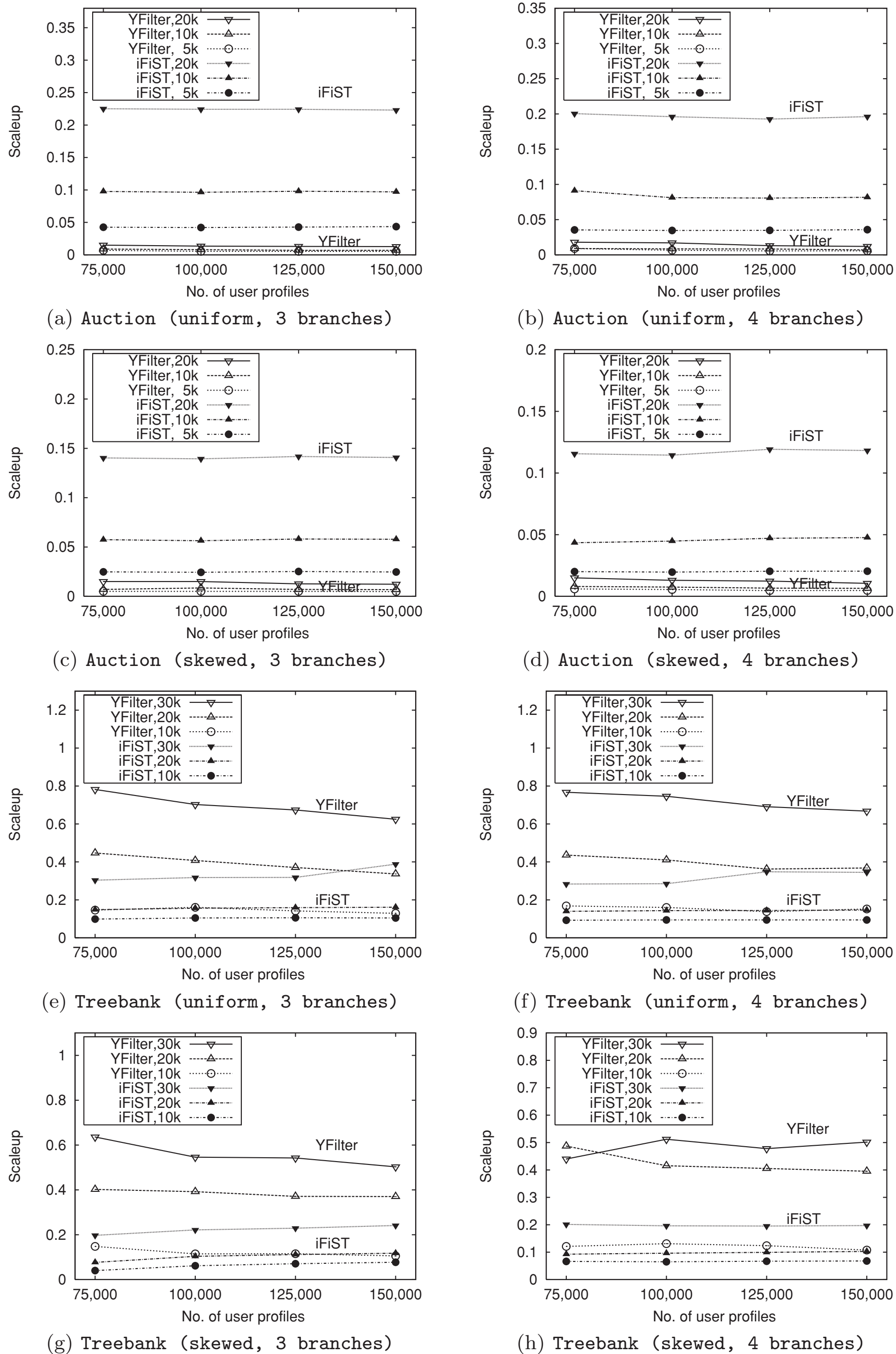
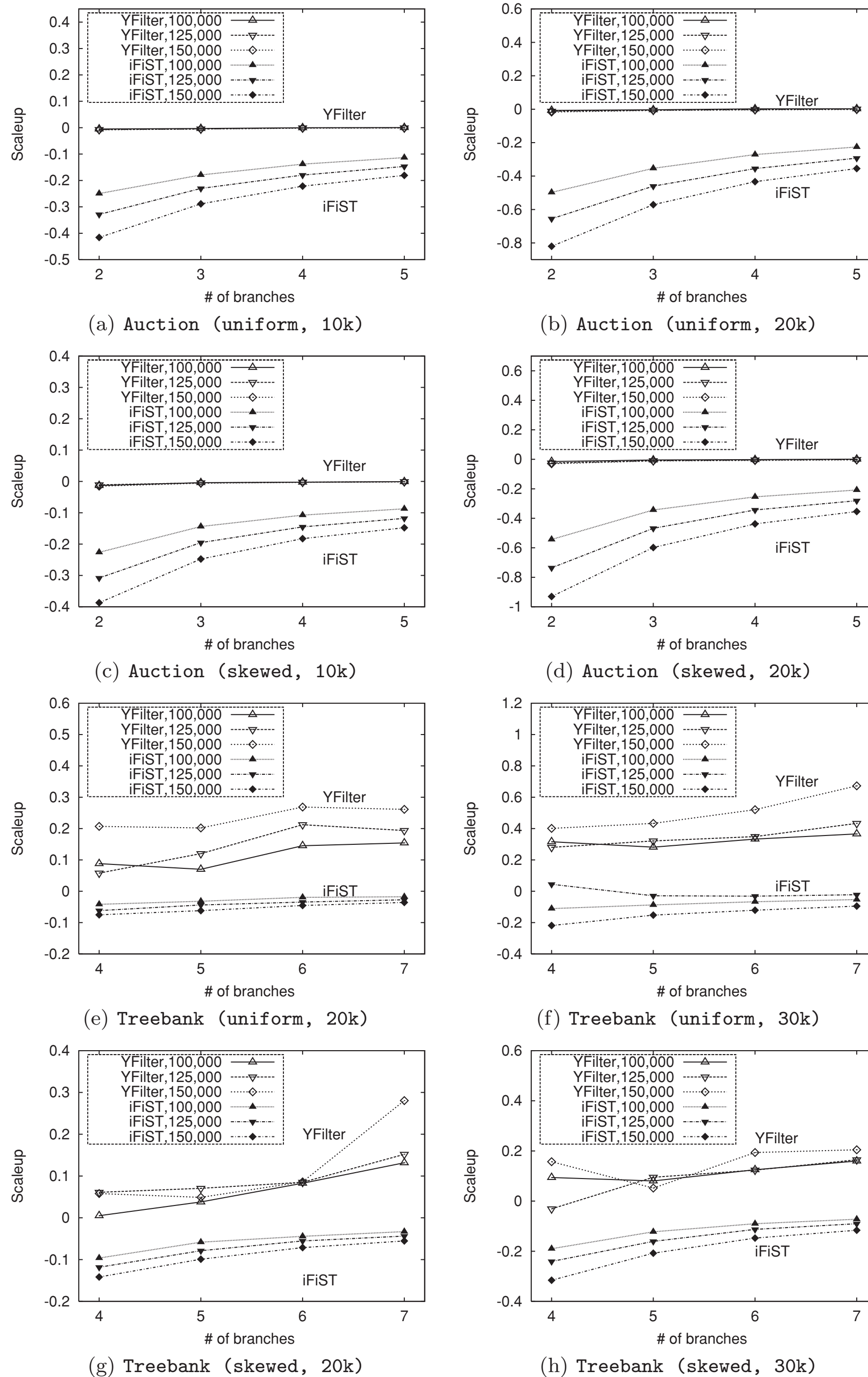


Fig. 21. Varying number of user profiles.



(a) Auction (uniform, 10k)

(b) Auction (uniform, 20k)

(c) Auction (skewed, 10k)

(d) Auction (skewed, 20k)

(e) Treebank (uniform, 20k)

(f) Treebank (uniform, 30k)

(g) Treebank (skewed, 20k)

(h) Treebank (skewed, 30k)

Fig. 22. Varying number of branches in user profiles.

branches was increased from 1 to 5. This caused a negative scaleup for YFilter and iFiST. YFilter handled twig patterns only by decomposing them into several linear paths, whereas iFiST handled twig patterns by *holistic* processing. Hence the average filtering time for iFiST reduced with increase in branch size. This difference explains why iFiST had a negative scaleup with increase in branch size. A similar performance trend was observed in other cases shown in Figures 22(c) and 22(d). Overall we observed that iFiST had scaled better than YFilter.

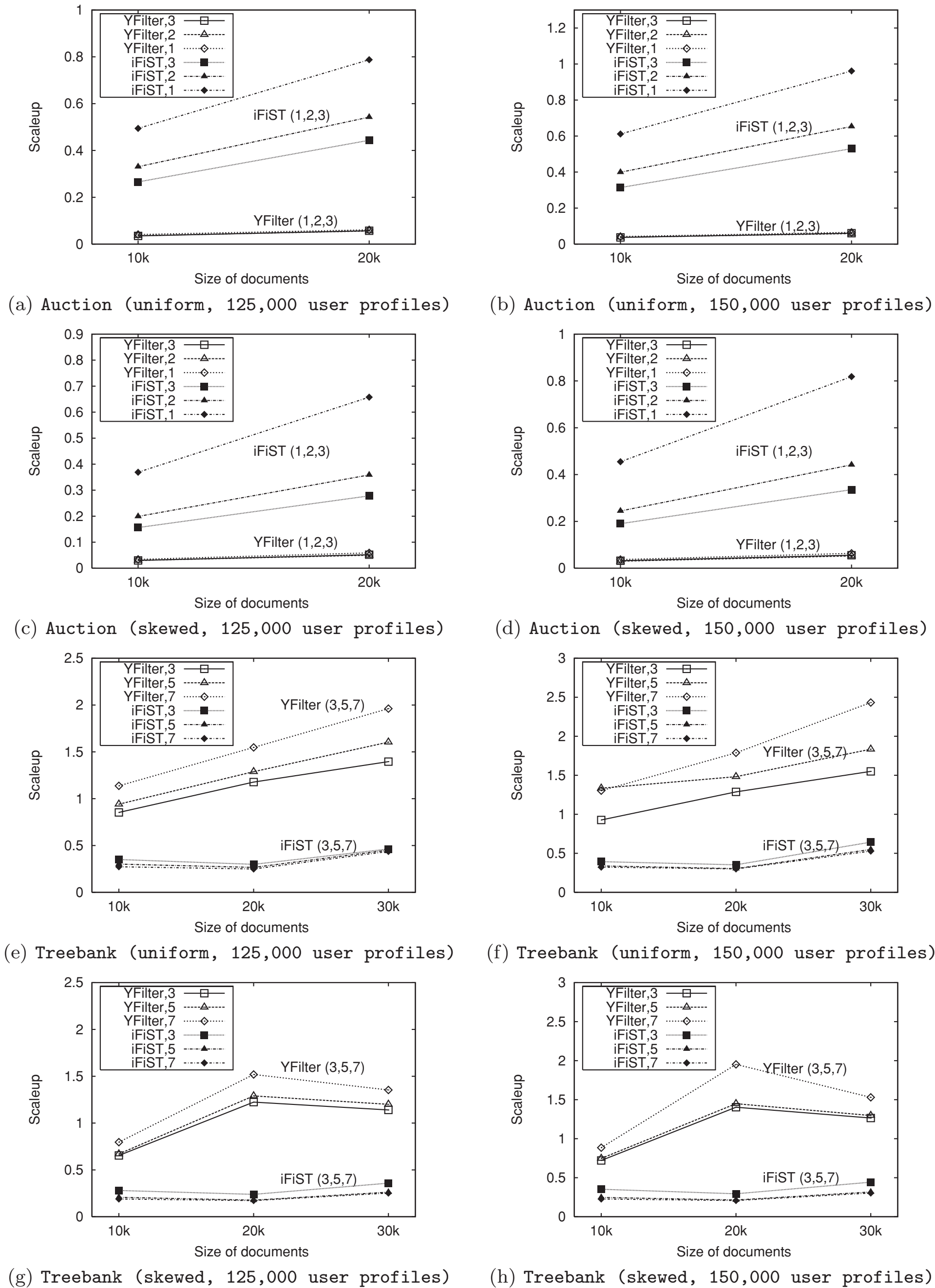
Figures 22(e) and 22(f) show the *scaleup* for Treebank dataset `20k` and dataset `30k` using the uniform twig set for YFilter and iFiST, respectively. The filtering cost of YFilter increased as the number of branches in the twig patterns increased from 3 to 7. This trend was observed for all the twig set sizes that we used. On the contrary, the filtering time for iFiST actually decreased with an increase in the number of branches. This appears as the negative scaleup in Figure 22(e) and 22(f). The reason why the filtering time for iFiST 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. Both the *holistic* processing of twig patterns and the reduction of the number of matches caused iFiST to have a negative scaleup with the number of branches. A similar performance trend was observed in other cases shown in Figures 22(g) and 22(h).

These results for Auction and Treebank datasets demonstrate that the *holistic matching* of twig patterns by iFiST yields better scalability than YFilter.

7.2.3 *Varying Size of XML Documents.* In this section, we analyze the performance of iFiST and YFilter by comparing their scaleup factor as the size of the XML documents increases.

The evaluation results for Auction and Treebank datasets are summarized in Figure 23. For each of the plots for Auction datasets, the results from user profiles with 1, 2, and 3 branches are shown. In the plots for Treebank datasets, the results from user profiles with 4 and 6 branches were omitted, because they showed trends similar to those from user profiles with 3, 5, and 7 branches for both YFilter and iFiST. The filtering time clearly increased with the sizes of documents.

Figures 23(a) through (d) show the *scaleup* performance of YFilter and iFiST measured for a set of user profiles for Auction with different numbers of user profiles and different types of tag name distributions. The x-axis shows different document sizes: `10k`, `20k`. The size `5k` serves as the base case for Auction datasets. The scaleup plots of iFiST grew more quickly than those of YFilter, and the gap in the scaleup between YFilter and iFiST was widened as the size of the documents increased. For Auction datasets, YFilter showed the better scaleup than iFiST. But we observed that the gap in the scaleup between YFilter and iFiST was narrowed as the number of branches increased.



(a) Auction (uniform, 125,000 user profiles)

(b) Auction (uniform, 150,000 user profiles)

(c) Auction (skewed, 125,000 user profiles)

(d) Auction (skewed, 150,000 user profiles)

(e) Treebank (uniform, 125,000 user profiles)

(f) Treebank (uniform, 150,000 user profiles)

(g) Treebank (skewed, 125,000 user profiles)

(h) Treebank (skewed, 150,000 user profiles)

Fig. 23. Varying size of XML documents.

Figures 23(e) through (h) shows *scaleup* for YFilter and iFiST for a set of user profiles for Treebank with different numbers of user profiles and different types of tag name distributions. The x-axis shows different document sizes: `10k`, `20k`, and `30k`. The size `1k` serves as the base case for Treebank datasets. Unlike Auction datasets, we observed that the scaleup plots of iFiST grew more slowly than those of YFilter, which indicated that YFilter's filtering cost increased much faster than iFiST as the size of documents increased. We observed that the gap in the scaleup between YFilter and iFiST was widened as the size of the documents increased in the uniform dataset (Figures 23(e) through (f)). Due to heavily recursive and deep documents, YFilter had many active states in the NFA and hence had a higher filtering cost, because its stack grew quickly. In the skewed dataset (Figures 23(g) through (h)), iFiST has better scaleup than YFilter. We observed that the gap became narrower for `30k`. In the skewed dataset, the number of distinct tags was relatively smaller than that in the uniform dataset. Still, due to heavily recursive and deep documents, YFilter had many active states in the NFA, and required more time to filter.

In this particular set of experiments, there was no clear winner between YFilter and iFiST. It appears that YFilter outperforms iFiST and vice versa for filtering documents of different characteristics. YFilter scaled better when the number of distinct tags was small and the documents to filter were lightly recursive and shallow. On the other hand, iFiST scaled better when the number of distinct tags was large and the documents to filter were heavily recursive and deep.

7.2.4 *Varying Document Selectivity.* In this experiment, we measured the effect of the document selectivity on the filtering performance. The document selectivity is computed as follows.

Document selectivity = # of documents that matched any user profile/total number of documents.

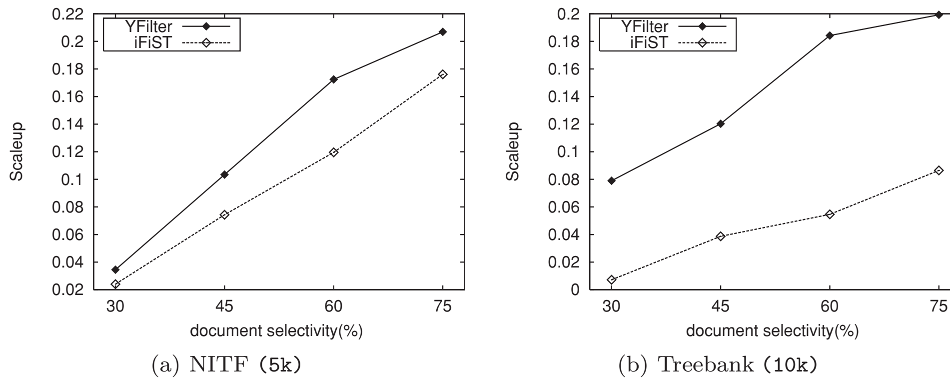
We fixed the number of user profiles to 75,000 and the number of documents to 3,000. The document selectivity was varied from 15% to 75%.

Figure 24 shows the performance results by varying document selectivities. We used NITF dataset `5k` and Treebank dataset `10k`. The filtering time increased for both YFilter and iFiST with increase in document selectivity. This was because more states were active during filtering for both YFilter and iFiST. However, iFiST showed better scaleup than YFilter.

7.3 Wildcard Processing

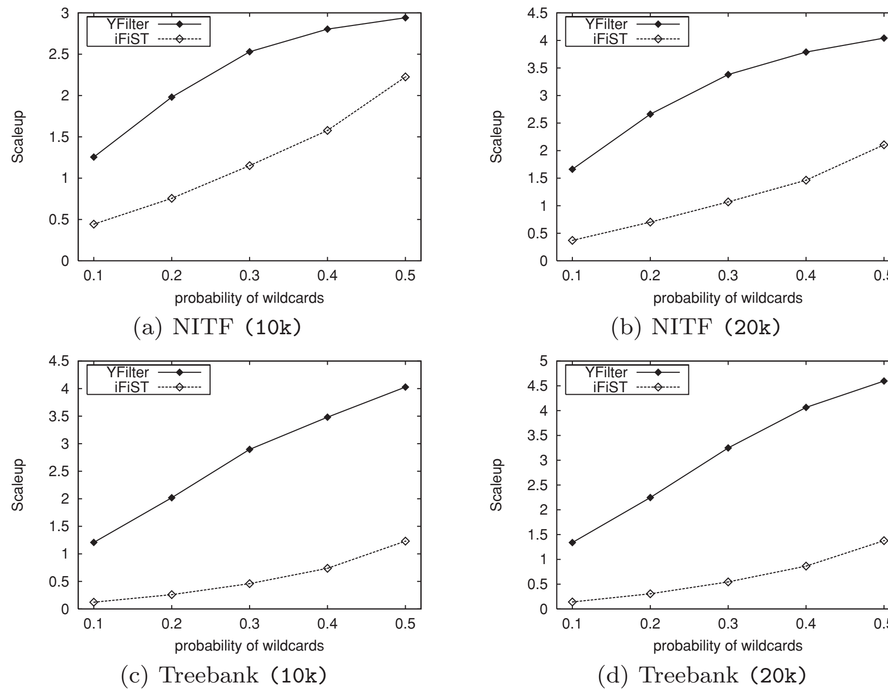
In this experiment, we compare the filtering performance of iFiST and YFilter when wildcards are present in twig patterns/user profiles.

The *scaleup* results for NITF and Treebank datasets are shown in Figure 25. The number of user profiles indexed was fixed at 150,000 and the probability of a wildcard "*" occurring at a location step was changed from 0.0 to 0.5. We set N_b (number of branches) to 3, z (skewness in element distribution) to 0.0 for each dataset. The base value for computing *scaleup* is the the filtering time when the probability of wildcards is set to 0.0.



(a) NITF (5k) (b) Treebank (10k)

Fig. 24. Varying document selectivity.

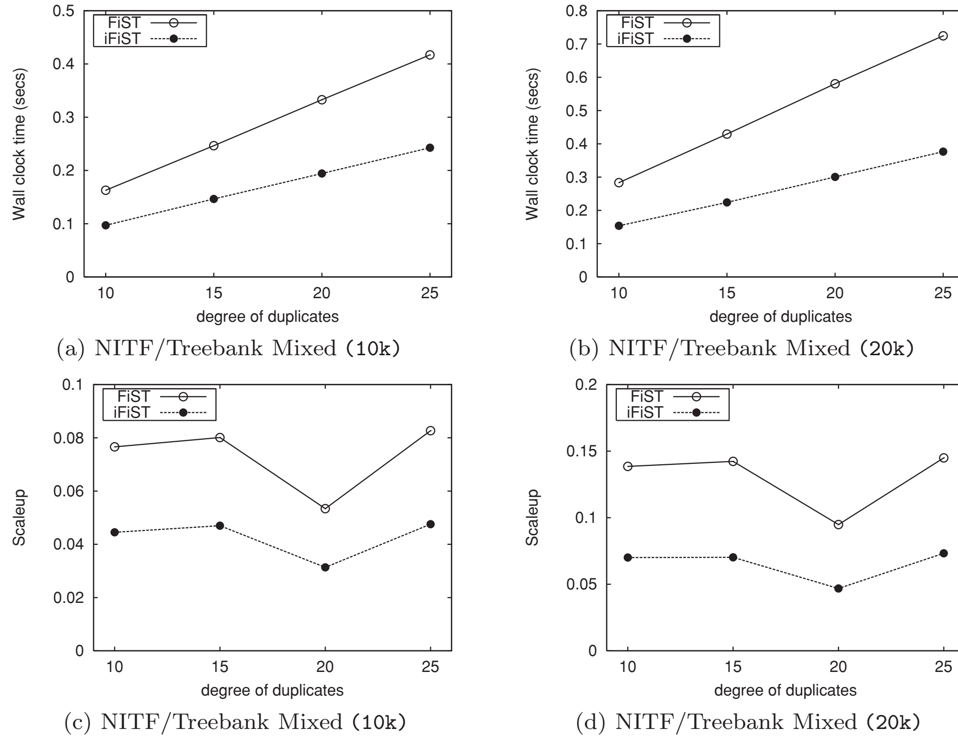


(a) NITF (10k) (b) NITF (20k)

(c) Treebank (10k) (d) Treebank (20k)

Fig. 25. Filtering performance in the presence of wildcards in twig patterns.

In both YFilter and iFiST, the filtering time increases as the probability of wildcards increases. However, iFiST's filtering time increases much slower than that of YFilter and has better *scaleup*. YFilter's processing time is affected by the number of active states in its NFA during filtering. The presence of wildcards increases this count. As described in Section 6, iFiST also pays additional cost for checking wildcard information (leaf/regular wildcards) and storing potential matches of stack elements into BranchID sets for branch wildcards.



(a) NITF/Treebank Mixed (10k) (b) NITF/Treebank Mixed (20k)

(c) NITF/Treebank Mixed (10k) (d) NITF/Treebank Mixed (20k)

Fig. 26. Effect of duplicate user profiles.

7.4 Benefits of Shared Processing of User Profiles

7.4.1 *iFiST versus FiST*. We measured the performance impact of the shared processing of common profile sequences described in Section 5. We created a light-weight version of iFiST called FiST, which does not merge common segments in the user profiles. That is, profile sharing is disabled in FiST. To compare the performance of iFiST and FiST, we measured the filtering cost in the wall-clock time as well as the scaleup.

We varied the number of duplicate user profiles to evaluate the efficacy of shared processing in iFiST. We first created a set of distinct user profiles and then replicated this set several times. The degree of duplicates is thus equal to the number of repetitions. To generate 250,000 user profiles with degree of duplicates 25, we first generated 10,000 distinct user profiles and then replicated them 25 times. For concise exposition, we carried out a set of experiments with a mixed set of user profiles created by choosing half the profiles from NITF and the other half from Treebank. The number of user profiles indexed was varied from 50,000 to 250,000 by changing the degree of duplicates from 5 to 25. We set $p_{//}$ to 0.2, N_b to 3, z to 0.8 for each DTD. We set L to 6 for NITF and 10 for Treebank. The set of input documents contained a mix of 500 NITF and 500 Treebank documents. The documents were grouped into “10k” and “20k”.

Figures 26(a) and 26(b) summarize the wall-clock time spent by FiST and iFiST. An increasing trend in the filtering time was observed while the degree

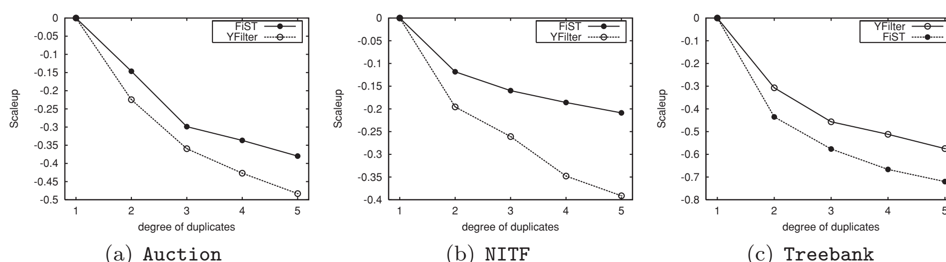


Fig. 27. Comparison between iFiST and YFilter by varying the degree of duplicates.

of duplication was increased. This was because the number of user profiles increased as the degree of duplicates increased. More importantly, iFiST consistently outperformed FiST, and the performance gap between them widened as the degree of duplicates increased. This improvement was a direct consequence of the shared processing of user profiles.

Figures 26(c) and 26(d) show the scaleup performance of FiST and iFiST. In these figures, the degree of 5 for duplicate profiles was used as the base case. The scaleup patterns were almost identical for both iFiST and FiST, but the scaleup of iFiST was consistently better than that of FiST.

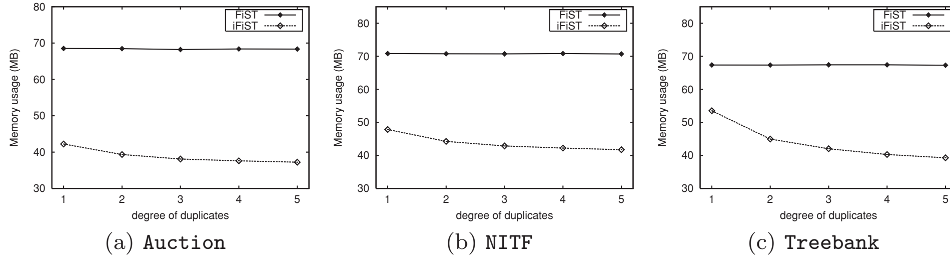
7.4.2 *iFiST versus YFilter.* We also compared iFiST with YFilter to see the impact of the shared processing. For this purpose, we fixed the number of user profiles to 75,000 and varied the degree of duplicates from 1 through 5. The input document set was `20k`.

Figure 27 shows the scaleup for different degree of duplicates. We observed negative values of scaleup for both systems, which implied that the execution time decreased with increase in the degree of duplicates. This shows that shared processing of user profiles is essential for high performance. While YFilter had better performance than iFiST for Auction and NITF datasets, iFiST had better performance than YFilter for Treebank dataset. The reason is explained in Section 5.2.3.

7.5 Memory Usage of iFiST

7.5.1 *iFiST versus FiST.* We investigated memory usage by iFiST and FiST for filtering XML documents. We measured total memory consumption by reading the statistics given in `/proc/self/statm`, which provides memory statistics of processes.

First, we measured the memory consumption by fixing the number of user profiles to 75,000 but varying the degree of duplicates from 1 to 5. The input document set was `20k`. As is shown in Figure 28, we observed that iFiST consumed approximately 20% to 40% less memory than FiST in most cases. The memory usage of FiST remained almost the same, whereas that of iFiST decreased as the degree of duplicates increased. This can be explained by the profile statistics given in Table IV. The number of sequence nodes, which affects the memory consumption by `FiST`, remained almost the same as the degree of duplicates increased. In contrast, the size of a segment table, which affects



(a) Auction (b) NITF (c) Treebank

Fig. 28. Memory usage of FiST and iFiST.

Table IV. User Profile Statistics

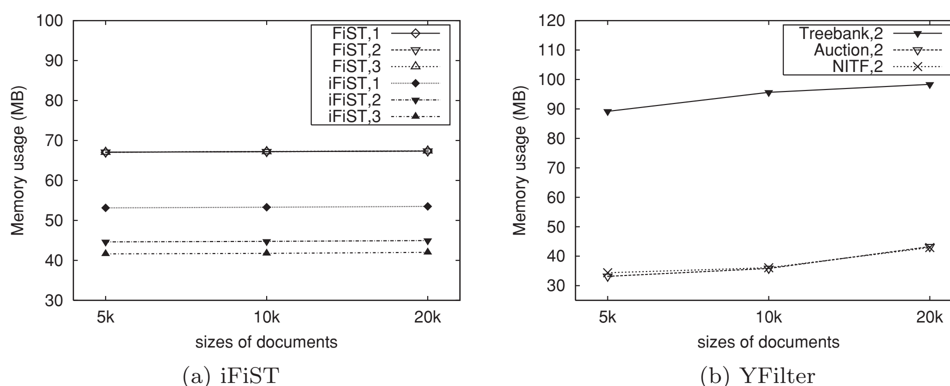
DTD	D_s	# of Sequence Nodes	# of Segments	# of Segments in the Segment Table
Auction	1	1,237,706	295,645	40,859
	2	1,233,870	295,658	23,903
	3	1,299,688	295,689	17,498
	4	1,232,692	295,468	14,255
	5	1,230,395	295,555	11,986
NITF	1	1,179,274	299,327	54,139
	2	1,176,070	299,328	31,130
	3	1,174,323	299,343	22,660
	4	1,175,540	299,312	18,155
	5	1,174,895	299,355	15,174
Treebank	1	1,194,958	298,209	117,896
	2	1,195,196	298,224	63,721
	3	1,194,414	298,143	44,387
	4	1,198,468	298,224	34,322
	5	1,195,295	298,105	28,167

the memory consumption by iFiST, decreased considerably as the degree of duplicates increased.

Second, we examined the effect of input document sizes on memory usage. We show only the results from Treebank because the results from other DTDs showed trends similar to that of Treebank. The number of user profiles was fixed at 75,000, whereas the degree of duplicates was varied from 1 to 3. The sizes of input XML documents ranged from “5k” to “20k”. The results are shown in Figure 29(a). The memory consumption by iFiST and FiST was insensitive to the sizes of input documents. This is because the size of a runtime stack, for both iFiST and FiST, is bounded by the depth of an input document. The average depth of input documents increased only slightly as the size of document increased, as shown in Table V.

In summary, iFiST has demonstrated that it can reduce overall memory requirement considerably by shared processing of user profiles, and its memory requirement is not affected by the size of input documents.

7.5.2 *iFiST versus YFilter.* Both iFiST and YFilter use a runtime stack but for fundamentally different purposes. iFiST uses a runtime stack to store the elements along a path from the current element being processed to the root of the document. The size of a runtime stack is thus bounded by the depth of



(a) iFiST (b) YFilter

Fig. 29. Memory usage of iFiST and YFilter for varying document sizes.

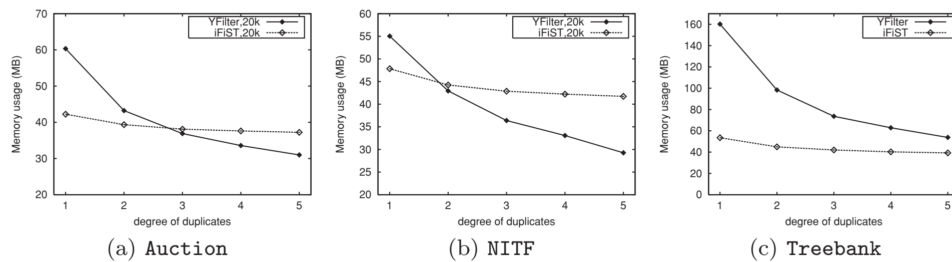
Table V. Average Depth of Input Documents

Sizes of Document	DTDs		
	Auction	NITF	Treebank
5k	10.79	8.13	18.59
10k	11.45	8.98	20.91
20k	12.07	9.41	21.00

a document being processed. On the other hand, YFilter uses a runtime stack to track the active and previously processed states during the execution of its NFA. Since YFilter runs on a Java virtual machine, we measured memory consumptions of YFilter by using the functions `totalMemory()` and `freeMemory()` in the `Runtime` class. The functions `totalMemory()` and `freeMemory()` return the total amount of memory and the amount of free memory in the Java virtual machine, respectively. The memory consumption of YFilter is computed by subtracting `freeMemory()` from `totalMemory()`.

We measured the memory consumption of iFiST and YFilter by fixing the number of user profiles to 75,000 and varying the degree of duplicates from 1 to 5. The input document set was `20k`. In Figure 30, we observed a downward trend in memory usage with increase in the degree of duplicates. As the degree of duplicates increased, the memory usage of YFilter decreased more considerably than that of iFiST. Although the number of segments reduced with the degree of duplicates in iFiST, the elements in the candidate list of the segment table were not shared among twig patterns and the number of elements were still large during the subsequence matching. This is why the memory consumption of iFiST did not reduce considerably. As future work, we plan to improve the sharing in the candidate list.

We also examined the effect of input document sizes on memory usage for YFilter. Recall that the memory consumption of iFiST was insensitive to the document sizes. (See Figure 29(a).) We show only the results when the degree of duplicate is 2 because the results from other DTDs showed similar trends. The number of user profiles was fixed at 75,000. The results are shown in Figure 29(b). The memory consumption by YFilter increased as the size of



(a) Auction (b) NITF (c) Treebank

Fig. 30. Memory usage iFiST and YFilter.

documents increased. The memory consumption of YFilter for Auction and NITF datasets increased by 30% from 5k to 20k.

7.6 Summary

We have carried out empirical evaluation of iFiST, in comparison with YFilter and FiST, under various operational conditions with datasets of different characteristics. The results from the experiments are summarized next.

- iFiST scaled better than YFilter with an increasing number of user profiles and documents of growing size, when the number of distinct tags was relatively large and the documents to filter were heavily recursive and deep (e.g., Treebank).
- On the other hand, when the number of distinct tags was relatively small and the documents to filter were lightly recursive and shallow (e.g., Auction), YFilter scaled better than iFiST with an increasing number of user profiles and documents of growing size.
- iFiST always scaled better than YFilter with an increasing number of branches in the user profiles.
- The performance benefit gained from shared processing of common user profiles was significant. When the degree of duplication was high in the set of user profiles, iFiST was up to approximately 48% faster and consumed about 40% less memory than FiST.
- The performance of iFiST was insensitive to the sizes of input documents to filter.

8. CONCLUSION

XML-enabled publish-subscribe (pub-sub) systems play an increasingly important role in e-commerce and Internet applications for selectively disseminating information to subscribed users with matching interests. One of the key challenges posed by the pub-sub systems is to provide scalable services for a potentially large and increasing number of subscribers in a timely fashion. To rise to this challenge, we have developed a novel XML document filtering system called iFiST. The iFiST system adopts a tree-to-sequence mapping strategy so that each incoming XML document can be holistically matched up against a set of user profiles of twig patterns in a bottom-up manner. Unlike the previous

pub-sub systems, the holistic matching allows us to find all matching user profiles without breaking a twig pattern into multiple linear paths and matching them separately. The iFiST system reduces the filtering time and the memory consumption even further by avoiding redundant processing of user profiles with common patterns.

Our experimental study shows that iFiST can outperform the state-of-the-art filtering system, YFilter, by achieving superior scalability, particularly when user profiles consist of complex XPath expressions and XML documents are heavily recursive and deep. Considering the fact that iFiST supports ordered twig pattern matching, iFiST is believed to deliver desirable services for applications that require order-sensitive filtering of deeply nested XML documents against user profiles consisting of complex and detailed XPath expressions.

ACKNOWLEDGMENTS

We would like to thank Yanlei Diao and her colleagues for providing the YFilter code for our experiments. We are also grateful to the anonymous reviewers for their constructive comments.

REFERENCES

ALTINEL, M. AND FRANKLIN, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th VLDB Conference*. 53–64.

APACHE. Apache Xerces C++ parser. http://xml.apache.org/xerces-c/.

BAR-YOSSEF, Z., FONTOURA, M., AND JOSIFOVSKI, V. 2004. On the memory requirements of XPath evaluation over XML streams. In *Proceedings of the 23rd ACM Symposium on Principles of Database Systems*. 177–188.

BAR-YOSSEF, Z., FONTOURA, M., AND JOSIFOVSKI, V. 2005. Buffering in query evaluation over XML streams. In *Proceedings of the 24th ACM Symposium on Principles of Database Systems*. 216–227.

BERGLUND, A., BOAG, S., CHAMBERLIN, D., FERNÁNDEZ, M. F., KAY, M., ROBIE, J., AND SIMÉON, J. XML path language (XPath) 2.0. http://www.w3.org/TR/xpath20/.

BOAG, S., CHAMBERLIN, D., FERNÁNDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMÉON, J. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/.

BOW, C., HUGHES, B., AND BIRD, S. 2003. Towards a general model of interlinear text. In *Proceedings of EMELD Workshop*.

BRUNO, N., GRAVANO, L., KOUDAS, N., AND SRIVASTAVA, D. 2003. Navigation- vs. index-based XML multi-query processing. In *Proceedings of the 19th IEEE International Conference on Data Engineering*. 139–150.

BRUNO, N., KOUDAS, N., AND SRIVASTAVA, D. 2002. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM-SIGMOD Conference*.

CANDAN, K. S., HSIUNG, W.-P., CHEN, S., TATEMURA, J., AND AGRAWAL, D. 2006. AFilter: Adaptable XML filtering with prefix-caching and suffix-clustering. In *Proceedings of the 32nd VLDB Conference*. 559–570.

CARZANIGA, A., RUTHERFORD, M. J., AND WOLF, A. L. 2004. A routing scheme for content-based networking. In *Proceedings of IEEE InfoCom 2004*. 918–928.

CASTRO, M., DRUSCHEL, P., MARIE KERMARREC, A., AND ROWSTRON, A. 2002. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE J. Select. Areas Comm. 20*, 8, 1489–1499.

CHAN, C. Y., FELBER, P., GAROFALAKIS, M. N., AND RASTOGI, R. 2002a. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the 18th IEEE International Conference on Data Engineering*. 235–244.

Chan, C. Y., Felber, P., Garofalakis, M. N., and Rastogi, R. 2002b. Efficient filtering of XML documents with XPath expressions. *VLDB J. 11*, 4, 354–379.

Chan, C. Y. and Ni, Y. 2007. Efficient XML data dissemination with piggybacking. In *Proceedings of the ACM-SIGMOD Conference*. 737–748.

Chandramouli, B., Phillips, J., and Yang, J. 2007. Value-Based notification conditions in large-scale publish/subscribe systems. In *Proceedings of the 33rd VLDB Conference*. 878–889.

Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D., and Candan, K. S. 2006a. Twig^2 Stack: Bottom-Up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of the 32nd VLDB Conference*. 283–294.

Chen, Y., Davidson, S. B., and Zheng, Y. 2006b. An efficient XPath query processor for XML streams. In *Proceedings of the 22nd IEEE International Conference on Data Engineering*. 79.

Chiu, A.-T. and Hsu, J.-L. 2006. An automaton-based filtering system for streaming musicxml. In *Proceedings of the International Conference on Semantic Web & Web Services*. 177–178.

Clark, J. 1999. XSL transformations (XSLT) version 1.0. http://www.w3.org/TR/xslt/.

Diao, Y., Altinel, M., Michael J. Franklin, H. Z., and Fischer, P. 2003. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Datab. Syst. 28*, 4, 467–516.

Diao, Y., Rizvi, S., and Franklin, M. J. 2004. Towards an Internet-scale xml dissemination service. In *Proceedings of the International Conference on Very Large Databases*. 612–623.

Diaz, A. L. and Lovell, D. 1999. XML generator.
http://www.alphaworks.ibm.com/tech/xmlgenerator.

Fenner, W. and Srivastava, D. 2005. XTreeNet: Scalable overlay networks for XML content dissemination and querying. In *Proceedings of the 10th International Workshop on Web Content Caching and Distribution*. 41–46.

Gong, X., Yan, Y., Qian, W., and Zhou, A. 2005. Bloom filter-based XML packets filtering for millions of path queries. In *Proceedings of the 21st IEEE International Conference on Data Engineering*. 890–901.

Green, T. J., Gupta, A., Miklau, G., Onizuka, M., and Suciu, D. 2004. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Datab. Syst. 29*, 4, 752–788.

Green, T. J., Miklau, G., Onizuka, M., and Suciu, D. 2003. Processing XML streams with deterministic automata. In *Proceedings of the 9th International Conference on Database Theory*. 173–189.

Gupta, A. K. and Suciu, D. 2003. Stream processing of XPath queries with predicates. In *Proceedings of the ACM-SIGMOD Conference*. ACM Press, 419–430.

He, B., Luo, Q., and Choi, B. 2005. Cache-Conscious automata for XML filtering. In *Proceedings of the 21st IEEE International Conference on Data Engineering*. 878–889.

He, B., Luo, Q., and Choi, B. 2006. Cache-Conscious automata for XML filtering. *IEEE Trans. Knowl. Data Eng. 18*, 12, 1629–1644.

Hong, M., Demers, A. J., Gehrke, J., Koch, C., Riedewald, M., and White, W. M. 2007. Massively multi-query join processing in publish/subscribe systems. In *Proceedings of the ACM-SIGMOD Conference*. 761–772.

Hou, S. and Jacobsen, H.-A. 2006. Predicate-Based filtering of XPath expressions. In *Proceedings of the 22nd IEEE International Conference on Data Engineering*. 53.

Kwon, J., Rao, P., Moon, B., and Lee, S. 2005. FiST: Scalable XML document filtering by sequencing twig patterns. In *Proceedings of the 31st VLDB Conference*. 217–228.

Kwon, J., Rao, P., Moon, B., and Lee, S. 2007. Value-Based predicate filtering of streaming XML data. In *Proceedings of the 1st International Workshop on Data Management in Ubiquitous Computing*. 266–271.

Kwon, J., Rao, P., Moon, B., and Lee, S. 2008. Value-Based predicate filtering of XML documents. *Data Knowl. Eng. 67*, 1, 51–73.

Lewis, W. D. Personal communications. http://zimmer.csufresno.edu/~wlewis/.

Li, G., Hou, S., and Jacobsen, H.-A. 2008. Routing of XML and XPath queries in data dissemination networks. In *Proceedings of the 28th International Conference on Distributed Computing Systems*. 627–638.

Li, Q. and Moon, B. 2001. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th VLDB Conference*. 361–370.

LU, J., CHEN, T., AND LING, T. W. 2004. Efficient processing of xml twig patterns with parent child edges: A look-ahead approach. In *Proceedings of ACM the 13th International Conference on Information and Knowledge Management*. 533–542.

LUDäSCHER, B., MUKHOPADHYAY, P., AND PAPAKONSTANTINOU, Y. 2002. A transducer-based XML query processor. In *Proceedings of the 28th VLDB Conference*. 227–238.

MEGGINSON, D. Simple API for XML. http://sax.sourceforge.net/.

MILIARAKI, I., KAOUDI, Z., AND KOUBARAKIS, M. 2008. Xml data dissemination using automata on top of structured overlay networks. In *Proceedings of the 17th International World Wide Web Conference*. ACM, New York, 865–874.

MILO, T., ZUR, T., AND VERBIN, E. 2007. Boosting topic-based publish-subscribe systems with dynamic clustering. In *Proceedings of the ACM-SIGMOD Conference*. 749–760.

MORO, M. M., BAKALOV, P., AND TSOTRAS, V. J. 2007. Early profile pruning on XML-aware publish-subscribe systems. In *Proceedings of the 33rd VLDB Conference*. 866–877.

MÜLLER, K. 2004. Semi-Automatic construction of a question treebank. In *Proceedings of the 4th International Conference on Language Resources and Evaluation*.

MUSICXML. MusicXML definition. http://www.recordare.com//xml.html.

NITF. NITF: News industry text format. http://www.nitf.org/.

PENG, F. AND CHAWATHE, S. S. 2003. XPath queries on streaming data. In *Proceedings of the ACM-SIGMOD Conference*. ACM Press, 431–442.

PRÜFER, H. 1918. Neuer Beweis eines satzes über permutationen. *Archiv für Mathematik und Physik 27*, 142–144.

RAMASUBRAMANIAN, V., PETERSON, R., AND SIRER, E. G. 2006. Corona: A high performance publish-subscribe system for the World Wide Web. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation (NSDI'06)*. USENIX Association, 2–2.

RAO, P., CAPPOS, J., KHARE, V., MOON, B., AND ZHANG, B. 2007. Net-x: Unified data-centric Internet services. In *Proceedings of 3rd International Workshop on Networking Meets Databases (NetDB'07)*.

RAO, P. AND MOON, B. 2004. PRIX: Indexing and querying XML using Prüfer sequences. In *Proceedings of the 20th IEEE International Conference on Data Engineering*. 288–299.

RAO, P. AND MOON, B. 2006. Sequencing XML data and query twigs for fast pattern matching. *ACM Trans. Datab. Syst. 31*, 1, 299–345.

SHAH, R., RAMZAN, Z., JAIN, R., DENDUKURI, R., AND ANJUM, F. 2004. Efficient dissemination of personalized information using content-based multicast. *IEEE Trans. Mobile Comput. 3*, 4, 394–408.

TIAN, F., REINWALD, B., PIRAHESH, H., MAYR, T., AND MYLLYMAKI, J. 2004. Implementing a scalable XML publish/subscribe system using a relational database system. In *Proceedings of the ACM-SIGMOD Conference*. 479–490.

TREEBANK. The Penn treebank project. http://www.cis.upenn.edu/˜treebank/.

XMARK. XMark - An XML benchmark project. http://www.xml-benchmark.org/.

YAN, T. W. AND GARCIA-MOLINA, H. 1999. The SIFT information dissemination system. *ACM Trans. Datab. Syst. 24*, 4, 529–565.

Received October 2008; revised June 2009; accepted July 2009