Data & Knowledge Engineering 67 (2008) 51–73

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Value-based predicate filtering of XML documents ☆

Joonho Kwon [a,*], Praveen Rao [b], Bongki Moon [c], Sukho Lee [a]

[a] School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Republic of Korea
[b] Department of Computer Science and Electrical Engineering, and University of Missouri-Kansas City, Kansas City, MO 64110, USA
[c] Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA

ARTICLE INFO

Article history:
Received 15 November 2007
Received in revised form 10 June 2008
Accepted 10 June 2008
Available online 27 June 2008

Keywords:
XML filtering
Value-based predicates
Prüfer sequences

ABSTRACT

In recent years, publish–subscribe systems based on XML filtering have received much attention in ubiquitous computing environments and Internet applications. The main challenge is to process a large number of content against millions of user subscriptions. Several XML filtering systems focus on the efficient processing of structural matching of user subscriptions represented as XPath twig patterns. However, existing techniques provide limited or no support for twig patterns that contain various operators in the value-based predicates. In this paper, we present the pFiST system that filters XML documents by transforming twig patterns into sequences based on Prüfer's method. This sequencing idea for XML filtering was first demonstrated by FiST [J. Kwon, P. Rao, B. Moon, S. Lee, FiST: scalable XML document filtering by sequencing twig patterns, in: Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005, pp. 217–228]. The focus of pFiST is to support value-based predicates in twig patterns in addition to matching their structure. The pFiST system supports equality and non-equality operators, and in addition can handle logical operators such as `AND` and `OR` in the value-based predicates. Extensive experimental results show that pFiST provides good performance over data sets with different characteristics.

1. Introduction

Publish–subscribe (pub–sub) systems based on XML (Extensible Markup Language) document filtering play an important role in Internet applications by enabling selective dissemination of information. In a typical publish–subscribe (pub–sub) system, whenever new content is produced, it is selectively delivered to interested subscribers. This has 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 (Personal Digital Assistants) and desktops. Such services necessitate the development of software systems that enable scalable and efficient matching of a large number of content against millions of user subscriptions.

In XML filtering, user profiles are represented as *twig patterns*[1] using XPath expressions [2]. Generally, a twig pattern specifies patterns of selection predicates on multiple elements that have some specified tree structured relationship in an XML document [3]. Note that XML filtering is fundamentally different from XML query processing or pattern matching. The main goal of XML query processing is to find specific parts within XML documents which match a query by building suitable indexes over XML documents [3–7]. In XML filtering, the roles of twig patterns and documents are reversed – the twig patterns are indexed in

☆ This research was supported by the Ministry of Knowledge Economy, Korea, under the Information Technology Research Center support program supervised by the Institute of Information Technology Advancement (Grant Number IITA-2008-C1090-0801-0031).

* Corresponding author. Tel.: +82 2 880 7299; fax: +82 2 883 8387.
E-mail addresses: joonho@db.snu.ac.kr (J. Kwon), raopr@umkc.edu (P. Rao), bkmoon@cs.arizona.edu (B. Moon), shlee@snu.ac.kr (S. Lee).

[1] We use the term user profiles, twig patterns and queries interchangeably in the rest of the paper.

0169-023X/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2008.06.012

```
//market/stock[code= "IBM"][sell_price = 30]
//market/stock[code= "IBM"][sell_price > 25]
//market/stock[code= "HP"][sell_price < 40]
```

Fig. 1. Three twig pattern queries with value-based predicates.

order to quickly determine if they contain a match in an input document. Formally the problem of XML filtering can be stated as follows:

Given a set of twig patterns, find those patterns that appear in an input XML document.

Several XML filtering systems [8–14,1] focus on efficiently matching the structure of twig patterns. In our previous work [1], we developed a novel XML filtering system called FiST (Filtering by Sequencing Twigs). FiST focused on holistic structure matching of twig patterns without breaking them into several root-to-leaf paths and processing them individually. In addition, FiST focused on ordered twig pattern matching, which is useful for applications that require the nodes in a twig pattern to follow the document order in XML.

However, existing solutions provide limited or no support for twig patterns that contain value-based predicates with various operators. Consider three twig pattern queries in Fig. 1. Each of these queries consists of four elements `market`, `stock`, `code` and `sell_price`, and they share the same path expressions, namely, //`market`/`stock`/`code` and //`market`/`stock`/`sell_price`. Although these three twig pattern queries have the same structure, they can be reported as matches to an XML document only after their predicates are tested and are matched to the input document values.

Suppose that a document containing the stock price of `HP` is given to the system. If profile matching is done in top-down fashion (as is done by most existing approaches such as YFilter [9]), then all three queries will pass the test based on the structural part of queries, and the final match will be found only after checking against values-based predicates is completed.

To address this potential bottleneck in profile matching, we propose the pFiST (predicate enabled FiST) system to support value-based predicates in twig patterns. pFiST is an extension of the FiST system [1] and can support structure matching as well as value-based predicates in twig patterns. The value-based predicates are handled differently according to the operators in the twig patterns. Further logical operators such as `AND` and `OR` can be handled by pFiST. pFiST evaluates twig patterns in a bottom-up fashion, which means that the value-based predicates are checked before the structure matching. Thus we need not check the structure information of the first two twig patterns. Only the third twig pattern is tested and identified as a match to the document. This shows that a bottom-up evaluation has an advantage of considering less twig patterns containing value-based predicates.

The key contributions of this paper are summarized as follows:

- *Value-based predicate processing*: Our pFiST system provides a comprehensive solution to XML filtering by supporting the evaluation of value-based predicates in twig patterns in addition to matching of their structures. Thus pFiST can be used for a wide-variety of applications that require XML filtering.
- *Comparison and logical operators in predicates*: pFiST supports both `equality` operator (=) and `non-equality` operators ($<, >, <=, >=$) in a value-based predicate. Further, logical-AND and logical-OR operators can be arbitrarily specified in twig patterns and pFiST can process such twig patterns. These operators are handled by a simplification and decomposition approach.
- *Experimental evaluation*: For performance evaluation, extensive experiments were conducted over different data sets such as NITF (News Industry Text Format) and Treebank. We compared pFiST with YFilter for equality predicates by varying the size of input documents. Further, we evaluated the scalability and filtering time of pFiST for varying document sizes, varying twig patterns in terms of number, probability of operators/predicates, and selectivities. In addition, we measured the memory usage of pFiST.

The rest of the paper is organized as follows. Related work is briefly discussed in Section 2. Section 3 explains background for our work, and Section 4 gives an overview of the design of pFiST. We present the techniques for value-based predicate processing in Section 5 and describe the extensions for processing logical operators in Section 6. Section 7 discusses the results of our performance study. Finally, we conclude in Section 8.

2. Related work

The popularity of XML as a standard for information exchange has triggered several research efforts to build scalable XML filtering systems. Existing approaches can be broadly classified into four categories namely (1) automaton-based approaches, (2) index-based approaches, (3) sequence-based approaches and (4) other approaches.

Most of the previous approaches have been based on constructing automaton representations for user profiles [8–12,15]. XFilter [8] was one of the early works in XML filtering based on constructing a DFA (Deterministic Finite Automata) for user profiles. YFilter [9] is a continual work of XFilter and uses a NFA (Non-deterministic Finite Automata) based approach for shared processing of XPath expressions. Two approaches for processing value-based predicates were also studied. However, YFilter handles predicates on attributes or text data of elements which contain only equality operators and does not support AND/OR operators in value-based predicates.

There has been work on filtering using automata with buffers. XSM (XML Stream Machine) [10] adopted a transducer based approach and used a subset of XQuery as a query language. To handle a subset of XQuery properly, the authors introduced the use of internal buffers. However, XSM does not support the descendant axis in a query. A streaming XPath engine, called XSQ [11], 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. XPush [12] proposed the use of a modified deterministic pushdown automaton to simulate the execution of XPath filters and handle predicates.

More recently, Pfilter [15] was proposed for processing a special matching character '%' in value-based predicates, which is used by LIKE operator in SQL. For this reason, Pfilter extended YFilter by having two automata, one for structure matching and the other for value-based predicates. However, Pfilter does not support range-value predicates and logical AND/OR operators in value-based predicates.

In the second category, there are also several approaches [13,14] that build an index for efficient filtering. A trie-based data structure, called XTrie [13], was proposed to support filtering of complex twigs. XTrie indexes sub-strings of path expressions that only contain parent–child operators, and shares the processing of only these common sub-strings among the queries. XTrie focused only on structure matching. Bruno et al. [14] studied index-based and navigation-based XML multi-query processing and showed both techniques have their own advantages. However, they do not consider the queries that contain value-based predicates.

In the third category, several filtering systems are included [1,16,17]. FiST [1] proposed the first sequence-based XML filtering system which works in a bottom-up way by encoding XML documents and twig patterns into Prûfer sequences. Recently, branch sequencing approach [16] and BoXFilter [17] were proposed. The branch sequencing approach avoids the post-processing phase and it retrieves the matched nodes in a single parse of the document. BoXFilter is similar to FiST, but introduces the idea of early pruning by grouping sequences into envelopes. However, all these sequence-based approaches do not support the value-based predicates in twig patterns.

In the final category, there are also several approaches [18–22]. Tian et al. [18] proposed an XML-based pub–sub system using a relational database system. A predicate-based filtering system [19] has been proposed that encodes XPath expressions as ordered sets of predicates. A recent system called AFilter [20] exploits prefix and suffix commonalities in the set of XPath queries. However, neither supports the value-based predicates in a twig pattern. Note that the meaning of predicates in the predicate-based filtering system differs from that of predicates in twig patterns. The predicates translated from an XPath query encode the relative position information of each two adjacent tags [19]. For example, an XPath query /a//b is translated two predicates such as $(p_a, =, 1)$ and $(d(p_a, p_b), \geqslant, 1)$. The predicate $(p_a, =, 1)$ represents a constraint on the position of the tag a in the XPath query. The predicate $(d(p_a, p_b), \geqslant, 1)$ represents a constraint between the relative position of tag 'b' and tag 'a' and an ancestor–descendant relationship in the XPath query. More recently, Gou and Chirkova [21] have proposed two stream-querying algorithms, LQ and EQ, which are based on *lazy strategy* and *eager strategy*, respectively. However, they focused on processing a single XPath expression at a time. Koch et al. [22] proposed a technique for efficient search and navigation in XML documents by taking string matching algorithms. However, they assume that a nonrecursive schema is available. Their system does not support non-equality operators.

Much work has been done in the area of XML query processing and pattern matching [3–7,23]. In XML query processing, XML documents are indexed to quickly find all occurrences of a twig pattern. However, in XML filtering, the twig patterns are indexed in order to quickly determine those that appear in an input document. Bruno et al. [3] proposed a holistic twig pattern processing algorithm, called TwigStack. TwigStack achieves optimality for twig patterns with ancestor–descendant relationships only. TJFast [4] was proposed to access only leaf elements by exploiting extended Dewey IDs. Twig^{2}Stack [5] is a bottom-up algorithm for processing twig queries based on hierarchical stack encoding. PRIX [7,24] proposed the use of Prüfer sequences for XML indexing and twig query processing. However, the aforementioned approaches do not handle OR operators in twig patterns. Jiang et al. [6] have studied the efficient processing of twig queries with OR predicates in the context of XML indexing and query processing. FleXPath [23] is a framework that integrates structure and full-text querying in XML. Structure conditions of queries are evaluated by an XPath engine and "`contains`" predicates are processed an IR (Information Retrieval) engine. A materialized XPath views approach [25] was proposed for processing XML queries. The views may contain copies of XML fragments and can be used to answer a user query containing XPath expressions. A semantic caching system, called ACE-XQ [26], has been proposed to improve the query performance over XML documents in a distributed environment. However, FleXPath, the materialized XPath views approach, and ACE-XQ system do not support multi-query evaluation.

We now describe some of the more recent work in publish–subscribe systems [27–30]. Massively multi-query join processing (MMQJP) technique [29] was proposed for processing a large number of inter-document queries. Inter-document 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: SELECT, FROM and PUBLISH. A piggyback optimization [27] was proposed for optimizing the performance of content-based dissemination of XML data. However, they only considered a subset of XPath that uses only the child ("/") and descendant ("//") axes. Milo et al. [28] studied topic-based publish–subscribe systems and proposed a novel technique for minimizing the maintenance overhead for topics. In topic-based publish–subscribe systems, publishers and subscribers are connected together by a predefined topic. In XML filtering systems, however, subscribers specify their interests through XPath queries. Published documents are matched against

XPath queries and delivered to the interested subscribers. Chandramouli et al. [30] 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 this work does not deal with XML.

Our motivations. Most previous work on XML filtering focused on processing structural matches of twig patterns against incoming XML documents. Some of the previous work support a limited form of value-based predicates in twig patterns. They do not support non-equality operators and logical AND/OR operators in twig patterns. These shortcomings have motivated us to build a comprehensive XML filtering system that evaluates value-based predicates in twig patterns and matches their structure holistically.

3. Background

In this section, we describe the twig pattern queries supported by pFiST, explain how XML documents are processed by the SAX parser, and describe how Prüfer sequences are generated from twig patterns.

3.1. Twig pattern queries supported by pFiST

A user profile (or twig pattern) is expressed using the XPath language [2]. XPath defines expressions for selecting nodes of an XML document tree. Generally, a twig pattern specifies patterns of selection predicates on multiple elements that have some specified tree structured relationship in an XML document [3].

Fig. 2 describes a subset of XPath supported by pFiST. This subset contains location path expressions as well as value-based predicates. A location path is a structural pattern composed of sub-expression called step, joined by the '/' or '//', where '/' denotes a child location step axis and '//' denotes a descendant location step axis. A value-based predicate is an expression that has comparison operators and logical operators and is evaluated with respect to the context node in an XPath expression. Comparison operators include equality (=) and non-equality operators $(<, >, <=, >=)$ and logical operators include AND/OR operators. Note that the NOT operator is not supported by pFiST. For example, `//market/stock[code=HP OR code=IBM AND sell_price > 25]` is a valid XPath query for the grammar in Fig. 2.

3.2. SAX events for XML filtering

Incoming XML documents that need to be filtered are first parsed by a SAX (Simple API for XML) parser [31]. The SAX parser generates a *StartTagHandler* event for each opening tag of an element and an *EndTagHandler* event for each closing tag of an element. A *Characters* event reports a string, which is inside between a start tag and an end tag. A *StartDocument* event and an *EndDocument* event are reported at the start and the end of an XML document.

Fig. 3 shows an example on how the SAX parser invokes five types of events when it parses an XML document. For example, when the parser sees the start tag and the end tag of 'c', it invokes the start element event and the end element event, respectively. When the parser sees the string '$t1$', it invokes the characters event. The bold-faced events are explained in Section 5.3.

3.3. Prüfer sequences

Prüfer sequences provide a bijection between the set of labeled trees on n vertices and the set of sequences of length $n-2$ on the labels 1 to n by removing nodes from the tree one at a time [32]. A simple iterative algorithm can be used to construct the Prüfer sequence for tree T_n with n nodes labeled from 1 to n. The algorithm starts with an empty sequence. At each step, a leaf node with the smallest label is deleted and the label of its parent node is appended to partial Prüfer sequences. After $n-2$

Path	:=	RelLocationPath | AbsLocationPath
AbsLocationPath	:=	'/' RelLocationPath
RelLocationPath	:=	Step '/' RelLocationPath | Step
Step	:=	Axis NodeTest | Step '[' Predicate ']'
Axis	:=	'/' | '//' | '@'
NodeTest	:=	STRING
Predicate	:=	Expression |
		Expression '=' Expression | Expression '!=' Expression |
		Expression '<' Expression | Expression '<=' Expression |
		Expression '>' Expression | Expression '>=' Expression |
		Predicate 'AND' Predicate | Predicate 'OR' Predicate
Expression	:=	STRING | INT | FLOAT | Path

Fig. 2. Grammar of XPath subset.

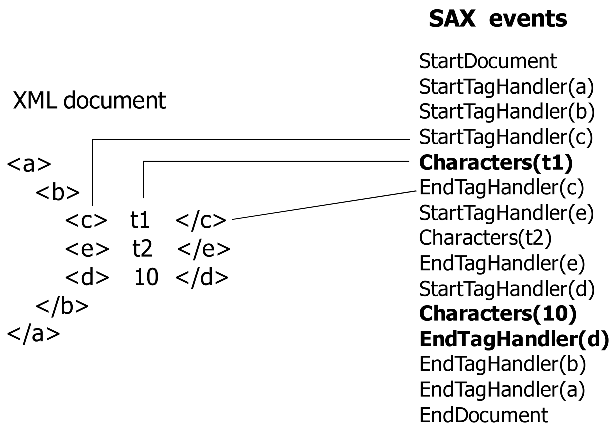


Fig. 3. SAX events example.

iterations, a single edge remains and we have produced a sequence of length $n-2$. The sequence $(a_1, a_2, a_3, \ldots, a_{n-2})$ is called the Prüfer sequence of tree T_n, where a_i is the label of the parent of a node with the ith smallest label.

The Labeled Prüfer Sequence (LPS) of an XML document tree is obtained by replacing the node numbers in the sequence with XML tags [7,24]. Because the leaf nodes do not participate in the sequence, we used the extended Prüfer method by adding a dummy child node to each leaf in the tree. In this paper, we use the term Prüfer sequence to refer to the extended LPSs.

Example 1. Consider a twig pattern in Fig. 4. The leaf nodes of the twig pattern are extended by adding dummy child nodes d_1 through d_4. The nodes of the twig pattern are labeled in postorder. The Prüfer sequence of the twig pattern using the node numbers is 2 10 4 10 6 9 8 9 10. The LPS of the twig pattern is B A C A G E F E A.

Algorithm 1 illustrates how Prüfer sequences are generated for an XML document using a SAX parser. 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, because a leaf node is considered to have a dummy child node for the purpose of producing an extended LPS (Lines 2–3). Whether the tag is a leaf or not, the top element is popped from the stack (Line 4) and the new top element is used as the next Prüfer sequence label (Line 5). The filtering algorithm *FindSubsequence* is explained later in Section 4.

Algorithm 1: Prüfer sequences generation

```
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( ));
   endif
4: S.pop( );
5: FindSubsequence(S.top( ));
end
```

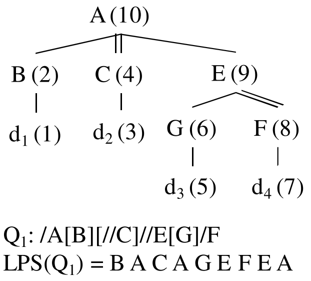


Fig. 4. Prüfer sequence for a twig pattern.

4. The proposed system

In this section, we present an overview of pFiST and describe the data structures and filtering algorithms. Note that pFiST extends our earlier work on FiST to support value-based predicates, and thus shares the basic features of FiST.

4.1. Overview of pFiST

An architectural overview of core modules and the data flow in the pFiST system is shown in Fig. 5. Twig patterns (i.e., user profiles) are parsed by the XPath parser and converted into Prüfer sequences. The collection of Prüfer sequences is stored in a hash-based dynamic index called *Sequence Index*. For each sequence, an auxiliary list called *Profile Sequence* is maintained. The filtering engine progressively constructs the Prüfer sequence representation of an input XML document and performs certain operations when the SAX parser events are invoked.

We have modified the FiST system to accommodate handling of values-based predicates. The *simplification* and *decomposition* steps for a twig pattern with AND/OR operators are added into the XPath parser. The SAX events are enhanced for handling value-based predicates. In addition, the Profile Sequence representation is extended to store the value-based predicates and the subsequence matching step is modified to process value-based predicates. The details are explained in Sections 5 and 6. However, the refinement phase of FiST, that verifies branch node matches, remains unchanged, because the value-based predicates cannot occur as branch nodes in the twig patterns.

4.2. Data structures

A twig pattern has a tree structure and its nodes have either parent–child or ancestor–descendant relationships. Each twig pattern is converted into a Profile Sequence that consists of Sequence Nodes that maintain all the information in the pattern. Each node in the Profile Sequence has four attributes namely **label**, **qid**, **loc**, and **sym**. The attribute **label** stores the Prüfer sequence label, **qid** contains an unique identifier, and **pos** denotes the position of the node in the Profile Sequence. The attribute **sym** stores a combination of values as follows:

- '/' denotes a parent–child relationship.
- '//' denotes an ancestor–descendant relationship.
- '$' denotes a branch node.
- '#' denotes the end node in the Profile Sequence.

Given a node q in the Profile Sequence, its four attributes are referred as $q_{\text{label}}, q_{\text{qid}}, q_{\text{loc}}$ and q_{sym}, respectively.

Two sample twig patterns and their corresponding Profile Sequences are given in Fig. 6a. The first Sequence Node of Q_1 has a value '/' due to a parent–child relationship with the second node. The first Sequence Node of Q_2 has a value '//' due to an ancestor–descendant relationship with the second node. Q_1 has two branch nodes A and E, which have three and two child nodes each. Thus the 2nd, 4th, 6th, 8th, 9th nodes have '$' symbols in their **sym** attributes. Some branch nodes could have two symbols at the same time. Since the 8th node of Q_1 has a parent relationship with the 9th node, it has two values '$' and '/'. The last node in the Profile Sequence has value '#'.

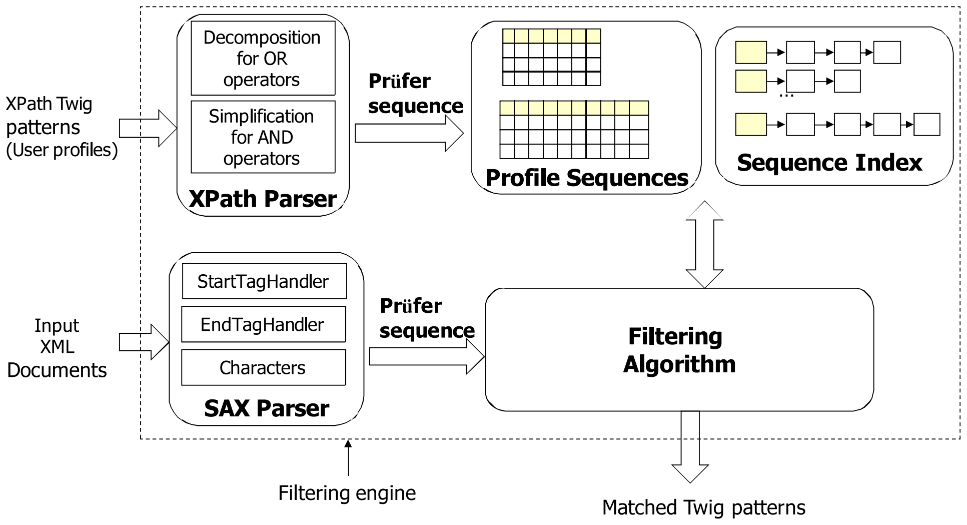
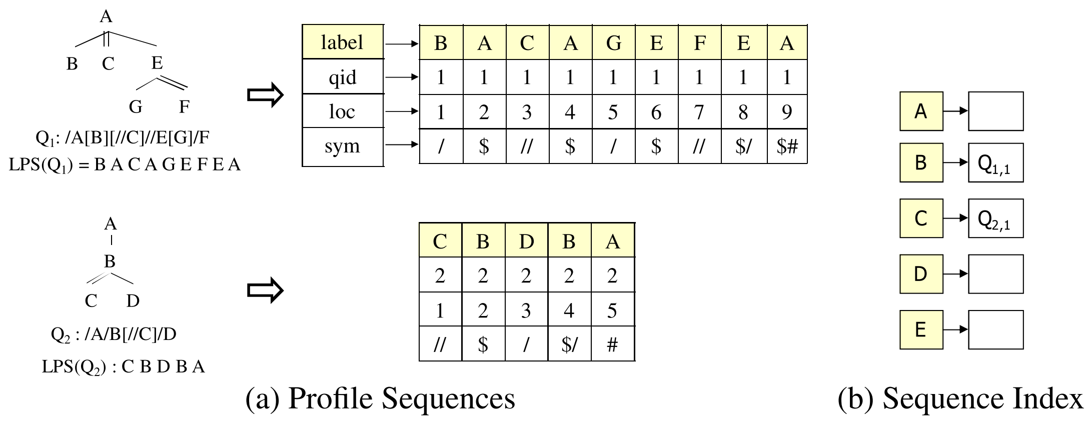


Fig. 5. Architecture overview.



label	B	A	C	A	G	E	F	E	A
qid	1	1	1	1	1	1	1	1	1
loc	1	2	3	4	5	6	7	8	9
sym	/	$	//	$	/	$	//	$/	$#

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

Fig. 6. Data structures.

A hash-based index called *Sequence Index* is maintained for efficient filtering of XML documents over twig patterns. 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, the Sequence Index contains a list of nodes from Profile Sequences that need to be matched next. The notation $Q_{i,j}$ denotes the jth Sequence Node of a twig pattern Q_i.

Initially the first nodes of the Profile Sequences are added to the Sequence Index. The Sequence Index is 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 depending on their **sym** values. On a successful test, the next nodes in the corresponding sequences are inserted into the Sequence Index. When the end of current document is reached, all nodes in Sequence Index will be removed except the first nodes of all the Profile Sequences.

Fig. 6b shows the initial Sequence Index after inserting two twig patterns shown in Fig. 6a. The first nodes of the Profile Sequences are added to the Sequence Index.

4.3. Filtering algorithm

The filtering algorithm of pFiST, which is an extension of the FiST system [1], also consists of two steps: *progressive subsequence matching* and *branch node verification*. A complete description of the filtering algorithm is provided in our earlier paper [1]. Here we briefly explain the main steps.

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

Theorem 1 [7]. *If a query tree Q is a subgraph of a document tree T, then LPS(Q) is a subsequence of LPS(T).*

Each time `EndTagHandler` is invoked, the top element of the stack indicates the ith element of the LPS of the document and the filtering procedure `FindSubsequence` is invoked (see Algorithm 1). During the subsequence matching phase, pFiST (similar to FiST) performs additional tests to eliminate most false matches by using the runtime stack.

The core filtering steps are shown in Algorithm 2. 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 4–6). The runtime stack allows parent–child and ancestor–descendant relationships to be tested during this phase. *StackTest* procedure in Line 8 is used to check whether the relationship of q and q' (the next Sequence Node of q) is satisfied according to q_{sym} by checking if it appears in the runtime stack. If a node q has a symbol '$', then we store the matched tag to facilitate the refinement phase (Lines 4 and 10). When a node q has a symbol '#', the branch node verification step is invoked to eliminate false matches. After the while loop, the next node of branch node (q_{sym} has a '$' symbol) is copied into the Sequence Index (Line 12).

Example 2. We demonstrate the execution of the filtering algorithm with an XML document T and two twig patterns in Fig. 7. When `FindSubseqeunce(C)` is invoked, the state of runtime global stack is shown in Fig. 7. First, we obtained the node $Q_{2,1}$ in the Sequence Index for key C. Because the value of $Q_{2,1_{\text{sym}}}$ is '/', the `StackTest`($Q_{2,2}$, $Q_{2,2}$,'/') is called. Because the label A is one element below the label B, the `StackTest` is successful. $Q_{2,2}$ is a branch node, so the next node $Q_{2,3}$ is added to the Sequence Index using hash key D because the label of $Q_{2,3}$ is D. Fig. 7c shows the changes during the subsequence matching.

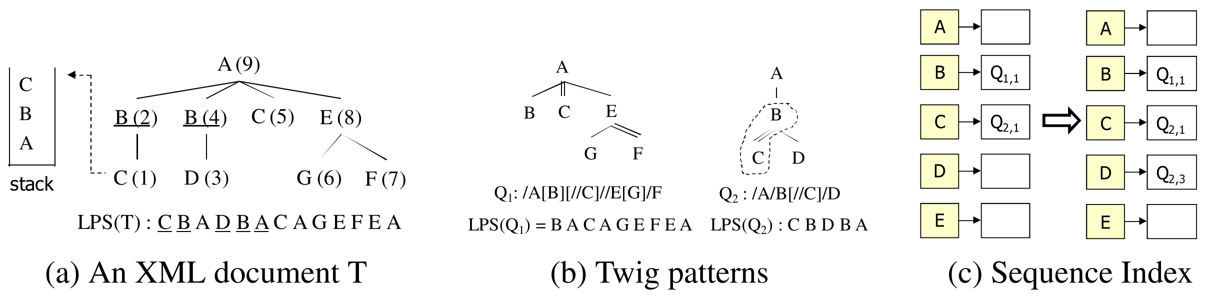


Fig. 7. An illustration of the filtering process.

Algorithm 2: Progressive Subsequence Matching

```
  Input: {L} - L is a Prüfer sequence label;
procedure FindSubsequence(L)
1:  $CurrentList \leftarrow SequenceIndex$[L];
2:  foreach SequenceNode $q$ in $CurrentList$ do
3:      while $q_{sym}$ is not '$' do
4:          if $q_{sym}$ contains '$' then store the matched tag;
5:          if $q_{sym}$ contains '#' then call the branch node verification;
6:          if $q_{sym}$ = '/' or $q_{sym}$ = '//' or $q_{sym}$ = '$/' or $q_{sym}$ = '$//' then
7:              $q' \leftarrow$ the next node of $q$;
8:              if $StackTest(q, q', q_{sym}) = false$ then exit from while loop;
9:              else $q \leftarrow q'$;
            endif
        endw
10:     store the matched tag;
11:     $q' \leftarrow$ the next node of $q$;
12:     copy $q'$ to Sequence Index using key $q'_{label}$;
  endfch
end
```

Theorem 1 guarantees having no false dismissals but it is possible to have false positives because the Labeled Prüfer Sequence does not consider the structural features of neither the query nor the XML document. Therefore, the second phase, branch node verification, is needed to discard false matches after checking the connectedness property for branch nodes in twig patterns.

Example 3. Consider the XML document *T* and twig patterns in Fig. 7. The Prüfer sequences of the document, LPS(T), is "C B A D B A C A G E F E A".[2] The Prüfer sequences of twig patterns Q_1 and Q_2, denoted by LPS(Q_1) and LPS(Q_2), are "B A C A G E F E A" and "C B D B A". Two queries are identified as candidate matches for the document during the first step, because LPS(Q_1) and LPS(Q_2) are both subsequences of LPS(T). Underlines in labels of LPS(T) show the subsequence for LPS(Q_2). The twig pattern Q_2 is discarded after branch node verification, since the first 'B' is matched to the first occurrence which is denoted as (B,2) in the document and the second 'B' is matched to the second occurrence denoted as (B,4).

5. Value-based predicate processing

In this section, we present our filtering scheme for value-based predicate processing. First, we describe observations for processing value-based predicates. Next, we propose the data structures for twig patterns containing value-based predicates. Two alternative representations for value-based predicates are proposed according to the observations. Then, we explain how the values in an XML document are used in the filtering algorithm to handle value-based predicates. To simplify the presentation, our discussion in this section focuses on the twig patterns that do not involve logical operators. We will present the techniques to support logical operators in Section 6.

[2] Spaces between labels are inserted for clarity.

5.1. Observation

For processing value-based predicates in twig patterns and XML documents, we have observed the following characteristics:

(1) When a SAX parser parses an XML document, the value is located between a start tag and an end tag and is delivered in the "characters" event.
(2) The value and its parent start tag in the XML document can be interpreted as having a parent–child relationship with an equality operator.
(3) Values in twig patterns have two modes, whereas the values in XML documents have only one mode. The value in the XML document is always an exact mode with an equality operator. However, the values of predicates in a twig pattern could have a range mode when non-equality operators appear in predicates.

Example 4. Consider an XML document in Fig. 3. A value "$t1$" is located inside its parent tag "c". This means that the tag "c" and the value "$t1$" have a parent–child relationship with an equality operator and the value "$t1$" is a value in exact mode. Consider a twig pattern Q_4 in Fig. 9a. The value of "5" in the twig pattern can be considered as a value in the range mode because of the "<" operator.

The values in the exact modes appears in both documents and twig patterns. We can transform these values into the labels of Prüfer sequences and process them in the same manner as the tags at the subsequence matching step. However, the values in the range modes appear only in twig patterns. In addition, we lose the information of non-equality operators if we transform only these values into the labels of Prüfer sequences. Thus, we need to treat values in twig patterns differently according to the operators.

5.2. Handling predicates in twig patterns

The basic idea of handling value-based predicates with equality operators is to treat value-based predicates as tag elements in twig patterns. Thus when the twig pattern is transformed into the Prüfer sequence representation, an independent sequence node for the value-based predicate is generated. This means the values in the exact mode become parts of the structure matching.

Fig. 8 shows a twig pattern Q_3 and its Profile Sequence. Q_3 has two value-based predicates with values "$t1$" and "10", which have equality operators. The 1st and 4th sequence nodes are generated for these predicates in the Profile Sequence. Attributes *label*, *qid*, *loc* and *sym* of each sequence node have appropriate values explained in Section 4.2. These sequence nodes are processed in the same manner as the nodes for elements during the subsequence matching.

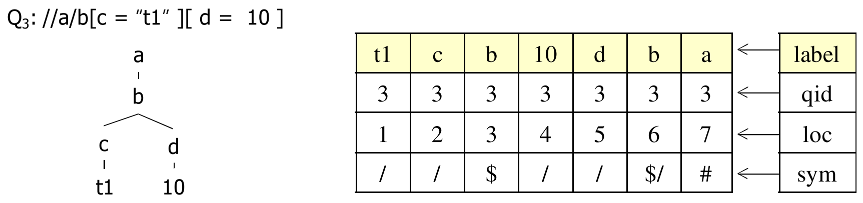
To process value predicates with non-equality operators, the sequence nodes need not be generated in the Profile Sequence. Instead, the value-based predicate information containing the value and the operator is added to the sequence node which is a parent node of the predicate. We do not generate an independent sequence node, because the range value cannot be used as a key to look up the Sequence Index.

Each predicate information is a 3-tuple (*Op*, *val*, *Type*), where:

- *Op* denotes an operator.
- *val* is the value of a value-based predicate.
- *Type* is an integer value for the type of *val*, which is determined as follows: 0 for a string type, 1 for a float type and 2 for an integer type.

Given a node q in the Profile Sequence, the predicate information is denoted by q_{pi}.

Example 5. Fig. 9 depicts a twig pattern Q_4 and its Profile Sequence. There are two value-based predicates "$c = t1$" and "$d < 5$" in Q_4. Because the first predicate "$c = t1$" has a value in exact mode due to the equality operator, the first node for value "$t1$" is generated in the Profile Sequence. However, the second predicate "$d < 5$" has a value in the range mode due to



t1	c	b	10	d	b	a	label
3	3	3	3	3	3	3	qid
1	2	3	4	5	6	7	loc
/	/	$	/	/	$/	#	sym

(a) Twig pattern (b) Profile representation

Fig. 8. Predicates with equality operators.

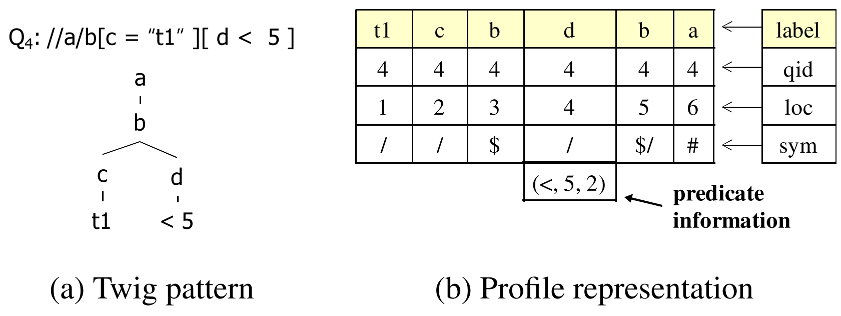


Fig. 9. Predicates with non-equality operators.

the non-equality operator. Thus the predicate information (<, 5, 2) is added to the 4th sequence node in the Profile Sequence.

5.3. Algorithms for processing value-based predicates

To handle the values (i.e., predicates) in an XML document, we enhanced SAX events and the subsequence matching function. Algorithm 3 shows SAX events, `StartTagHandler` `EndTagHandler` and `Characters`, which accommodate the filtering algorithm capable of processing value-based predicates. Algorithm 4 shows the `PredTest` procedure for processing value-based predicates. The `FindSubsequence` procedure in Algorithm 2 needs to be extended taking two parameters and to be augmented by inserting the `PredTest` procedure.

Algorithm 3: SAX handlers for processing value-based predicates

```
stack S;                                              /* a runtime global stack */
string Pred;                                          /* a string for the value predicate */
procedure StartTagHandler(tag)
1: S.push(tag)
2: foreach attribute attr in tag do
3:     FindSubsequence($attr_{value}$, Null);          /* exact mode */
4:     FindSubsequence($attr_{name}$, $attr_{value}$); /* range mode */
   endfch
end
procedure Characters(value)
5: Pred ← value
6: FindSubsequence(Pred, Null);                       /* exact mode */
end
procedure EndTagHandler(tag)
7: if tag is a leaf node then
8:     FindSubsequence(S.top( ), Pred);               /* range mode */
9:     Pred ← Null;
   endif
10: S.pop( );
12: if Pred is Null then FindSubsequence(S.top( ), Null);   /* exact mode */
13: else FindSubsequence(S.top( ), Pred);                   /* range mode */
14: Pred ← Null;
   end
```

In the `StartTagHandler` procedure, an opening tag in an XML document is stored in a runtime global stack. The runtime global stack is used during the subsequence matching. The values of an attributes are processed within this function. The `FindSubsequence` procedure is invoked twice. The first call in Line 3 is for the values in the exact mode, where the value of attribute ($attr_{value}$) is used as a key to look up the Sequence Index. The second call in Line 4 is for the values in the range mode, where attribute name ($attr_{name}$) is used as a key to look up the Sequence Index and the value of attribute ($attr_{value}$) is compared to the predicate information of a Sequence Node.

The `Characters` procedure handles values in the exact mode by calling the `FindSubsequence` procedure in Line 6. For processing values in the range modes of twig patterns, the bookkeeping for the value is done in Line 5. This value is used in the `EndTagHandler` procedure.

In the `EndTagHandler` procedure, the `FindSubsequence` procedure is also invoked twice. The `FindSubsequence` procedure in Line 8 is used to check value-based predicates with equality operators, whereas the `FindSubsequence` procedure in Line 12 is used to check value-based predicates with non-equality operators. The stored string is reset to NULL after it is processed.

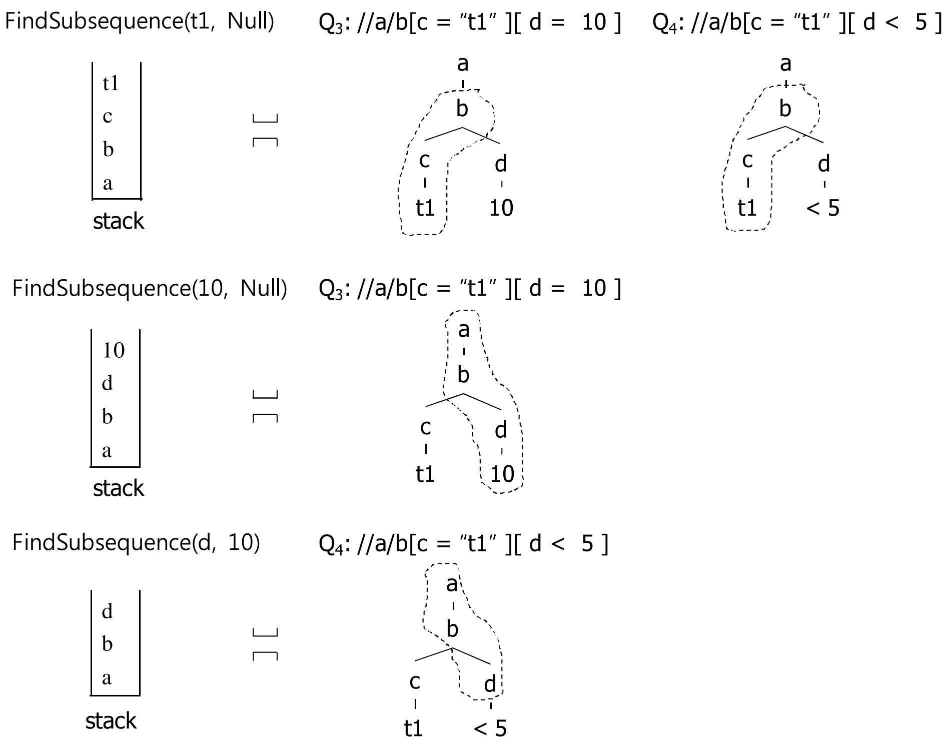
Algorithm 4: Checking value-based predicates

```
  Input:   {pi, str} - pi is predicate information;
                        str is a string from an XML document;
 procedure PredTest(pi, str)
 1: if str is NULL then return false;
 2: val is converted from str according to the type of $pi_{type}$;
 3: op ← the operator of $pi_{op}$;
 4: if Eval ($pi_{val}$, op, val) then
 5:     return true;
 6: else
        return false;
    endif
 end
```

Algorithm 4 shows the `PredTest` procedure which performs the comparison between predicate information (q_{pi}) and a string (*str*). The predicate information is obtained from a Sequence Node and a string is passed by the SAX handlers in Algorithm 3. If a *str* is NULL, nothing needs to be done (Line 1). Function `Eval`(pi_{val}, *op*, *val*) in Line 4 returns true if the relationship between values pi_{val} and *val* satisfies the operator *op*.

For processing value-based predicates, `FindSubsequence` in Algorithm 2 should be extended as follows. First, the extended `FindSubsequence` procedure takes two input parameters as shown in Algorithm 3. The first parameter (i.e., *tag*) is used as a key to lookup the Sequence Index. The second parameter (i.e., *str*) is used when the predicate information of Sequence Nodes in the current lists are checked. Second, the extended `FindSubsequence` procedure can be augmented by inserting the `PredTest` procedure before checking structure information of a twig pattern (Line 3 in Algorithm 2).

As for the augmentation, it is an easy task for pFiST because pFiST evaluates twig patterns and XML documents in a bottom fashion. However, it is a difficult to augment YFilter and other approaches by inserting `PredTest` procedure since they process twig patterns and XML documents in a top-down fashion.



FindSubsequence(t1, Null)
Q3: //a/b[c = "t1"][d = 10]
Q4: //a/b[c = "t1"][d < 5]
t1
c
b
a
stack
a
b
c
d
t1
10
< 5
FindSubsequence(10, Null)
10
d
b
a
FindSubsequence(d, 10)

Fig. 10. Value-based predicate processing.

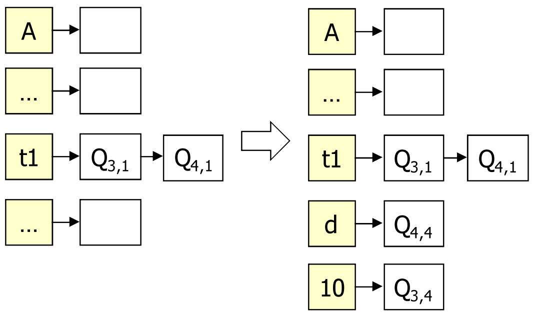


Fig. 11. Changes in Sequence Index during filtering.

We illustrate the execution of filtering algorithm with the XML document in Fig. 3a and twig patterns Q_3 and Q_4 in Figs. 8 and 9. The sequence of SAX events is also shown in Fig. 3a, however, we focus on some SAX events with bold-faced fonts in this example.

Example 6. Fig. 10 shows the changes in the global stack and main comparisons between the stack and twig patterns during filtering.

`FindSubsequence(t1, Null)` is invoked in `Characters(t1)` event. The current nodes $Q_{3,1}$ and $Q_{4,1}$ of the Sequence Index are obtained by using hash key "$t1$". Since the second parameter is Null, the `PredTest` procedure is skipped. The dotted regions of twig patterns Q_3 and Q_4 are compared to the stack. At the end of `FindSubsequence(t1)`, we found that both Q_3 and Q_4 are passed successfully. Thus the 4th sequence nodes of Q_3 and Q_4 are copied to the Sequence Index using the label "10" and "d" as keys, respectively. This is shown in Fig. 11.

In `Characters(10)` event, `FindSubsequence(10, Null)` is invoked and the current node $Q_{3,4}$ of the Sequence Index is obtained by using hash key "10". After a comparison between the stack and the dotted region of Q_3, Q_3 is identified as a match to the document.

In `EndTagHander(d)` event, `FindSubsequence(d,10)` is called for values in the range mode. The tag "d" is used to search the Sequence Index in Fig. 11. $Q_{4,4}$ existed in the current list. Since the second parameter is not Null, the `PredTest` is performed using the value of "10" and predicate information $(<, 5, 2)$ of $Q_{4,4}$. At this step, $Q_{4,4}$ fails to satisfy the predicate information.

6. Extensions

Queries in many real applications may contain logical-AND and logical-OR operators. In pFiST, when we transform twig patterns with AND/OR operators into Prüfer sequences, we avoid storing these operators as part of these sequences. To capture the semantics of these operators we propose a decomposition approach for OR operators and a simplification approach for AND operators.

6.1. Notations

We represent a twig pattern with logical operators as a tree containing two types of nodes namely (a) *location step node* (LNode) and (b) *predicate node* (PNode). A location step node (LNode) has a'/' or '//' axis with a tag name for node test. A predicate node (PNode) contains a value-based predicate of a query and the "AND/OR" operators including brackets '[' and ']'. Fig. 12 shows the tree representation for a twig pattern Q_5. An LNode is denoted by a circle and a PNode is denoted by a rectangle.

The above notations are similar to those introduced by Jiang et al. [6] for XML query processing.

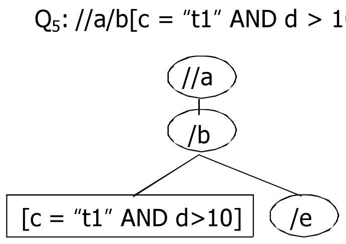


Fig. 12. Tree representation.

6.2. Predicates with OR operators

A decomposition approach is used by pFiST to handle a twig pattern with OR operators. The key idea is to split a twig pattern with OR-predicates into multiple twig patterns without OR-predicates. Although the number of twig patterns stored in the Sequence Index increases due to the decomposition of a query, the subsequence matching algorithm can be applied without any modifications.

As described in Algorithm 5, the steps for decomposing a twig pattern with OR operators are quite straightforward. Suppose a predicate in a twig query is in *disjunctive normal form*. If the number of OR operators in the predicate is k, the structural portion of the query (i.e., LNodes) will be replicated $(k+1)$ times, and each of the $(k+1)$ replicated queries will be given one of the $(k+1)$ conjunctive terms for its predicate. Generally, if a twig pattern query comes with n predicates $p_1, \ldots, p_n$ with k_i $(1 \leqslant i \leqslant n)$ OR operators each, then the structural portion of the query will be replicated $(k_1+1) \times (k_2+1) \times \cdots \times (k_n+1)$ times. Each of the replicated queries will then be given one of the (k_i+1) conjunctive terms for the ith OR-predicate p_i for each i $(1 \leqslant i \leqslant n)$. See the examples below for illustration of query decomposition.

Algorithm 5: Decomposition of twig patterns with OR operators

```
  **Input**: $\{q\}$ – q is a twig pattern having OR operators;
**procedure** Decomposition($q$)
1: **if** *q is in disjunctive normal form* **then**
2:     $num \leftarrow$ 1 + # of OR operators in $q$;
  **else**
3:     $num \leftarrow (k_1+1) \times (k_2+1) \times \cdots \times (k_n+1)$ when $q$ has $n$ predicates $p_1, \ldots, p_n$ with $k_i$ $(1 \leqslant i \leqslant n)$ OR operators each;
  **endif**
4: **foreach** *PNode p of q* **do**
5:     LNode $l \leftarrow$ the parent node of $p$;
6:     add all nodes of $q$ except $p$ to *List*;
7:     remove $p$ from $l$;
8:     delete OR operators and brackets such as '[' and ']' in $p$;
9:     create twig patterns to the value of *num* by copying the nodes from *List*
10:      **for** $i = 1$ **to** *num* **do**
11:           Add $i$th expression $e$ to LNode $l$ as a child node for the $i$th twig pattern;
       **endfor**
     **endfch**
  **end**
```

Example 7. Fig. 13 shows the decomposition of twig pattern Q_6 whose predicate is represented in disjunctive normal form. After applying the decomposition algorithm, Q_6 is decomposed into four twig patterns namely $Q_{6\_1}, Q_{6\_2}, Q_{6\_3}$ and Q_{6_4}. The twig pattern Q_{6_2} can be simplified because it contains an AND operator. (The simplication of AND operators is discussed in the following section.) Fig. 14 shows the decomposition of a twig pattern with multiple predicates. A twig pattern Q_7 is decomposed into the four twig patterns such as $Q_{7\_1}, Q_{7\_2}, Q_{7\_3}$ and Q_{7_4}.

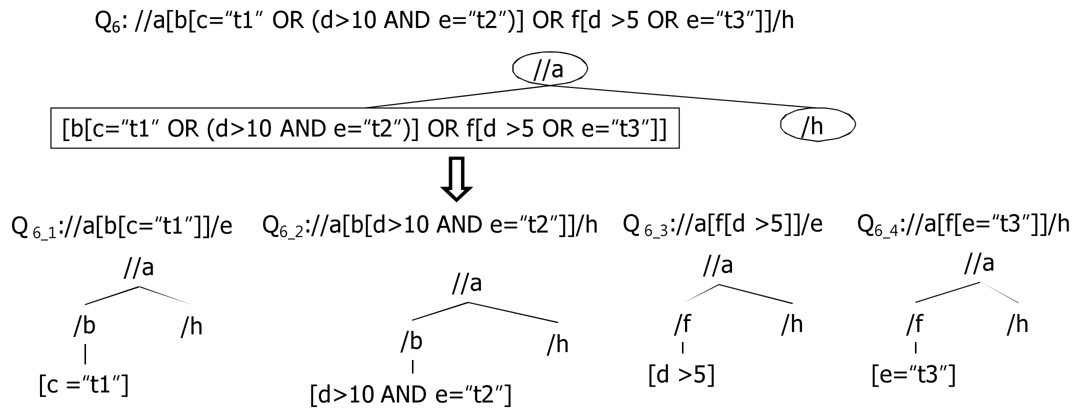

Q6: //a[b[c="t1" OR (d>10 AND e="t2")] OR f[d >5 OR e="t3"]]/h
//a
[b[c="t1" OR (d>10 AND e="t2")] OR f[d >5 OR e="t3"]]
/h
Q6_1://a[b[c="t1"]]/e
Q6_2://a[b[d>10 AND e="t2"]]/h
Q6_3://a[f[d >5]]/e
Q6_4://a[f[e="t3"]]/h
//a
/b
/h
[c ="t1"]
//a
/b
/h
[d>10 AND e="t2"]
//a
/f
/h
[d >5]
//a
/f
/h
[e="t3"]

Fig. 13. Decomposition of a twig pattern with OR operators in disjunctive normal form.

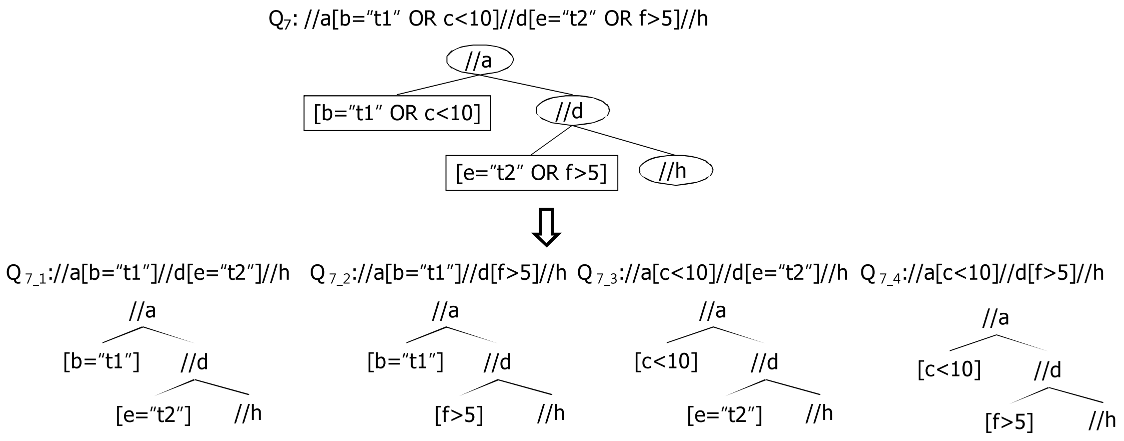

Q7: //a[b="t1" OR c<10]//d[e="t2" OR f>5]//h
//a
[b="t1" OR c<10]
//d
[e="t2" OR f>5]
//h
Q7_1://a[b="t1"]//d[e="t2"]//h
Q7_2://a[b="t1"]//d[f>5]//h
Q7_3://a[c<10]//d[e="t2"]//h
Q7_4://a[c<10]//d[f>5]//h
[b="t1"]
[c<10]
[e="t2"]
[f>5]

Fig. 14. Decomposition of a twig pattern with multiple predicates.

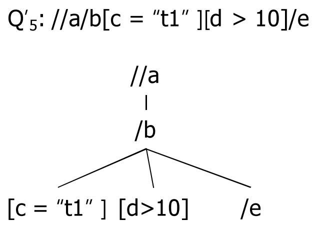


Fig. 15. Simplified twig pattern for Q_5 after removing the AND operator.

6.3. Predicates with AND operators

A simplification method is used by pFiST to handle a twig pattern with AND operators. The key idea of simplification can be understood using the following example. A predicate $[e_1$ AND $e_2]$ in a twig pattern has the same semantics as that of having two successive predicates $[e_1]$ and $[e_2]$. Thus we translate the predicates with AND operators of a LNode into a set of successive predicates of the same LNode without AND operators. After simplification of twig patterns, we can apply the filtering algorithm without any modifications.

The steps for simplifying a twig pattern with AND operators are given in Algorithm 6. For each PNode p of a twig pattern, the parent LNode l of p is identified and p is removed from l (Lines 2 and 3). After deleting AND operators and brackets such as '[' and ']' (Line 4), the expressions in p are enumerated (Line 5). Each expression is inserted as a child of LNode l which is a parent node of the PNode p (Line 6). In other words, the expressions become successive predicates of the parent node.

Algorithm 6: Simplification

```
  Input: {q} – q is a twig pattern having AND operators;
procedure Simplification(q)
1: foreach PNode p of the twig pattern q dol
2:     LNode l ← the parent node of p;
3:     remove p from l;
4:     delete AND operators and brackets such as '[' and ']' in p;
5:     foreach expression e in p do
6:         Add the expression e with brackets into l as the child node;
       endfch
     endch
  end
```

Example 8. Fig. 15 shows the twig pattern obtained after simplification of Q_5 in Fig. 12. Two expressions in the PNode become children of the LNode '/b'.

Table 1
Characteristics of DTDs

Name	Number of distinct tags	Max. depth of document
NITF	123	10
Treebank	250	20

7. Experimental results

In this section, we present the results of some experiments to analyze the performance of value-based predicate filtering.

7.1. Experimental setup

7.1.1. Data sets and twig patterns

In our experiments, we used two different data sets: `NITF` and `Treebank`. `NITF` is an XML-based DTD designed for the markup and delivery of news content [33]. `Treebank` is an XML-based encoding format for the representation of linguistic corpora [34]. Features of the data sets are shown in Table 1. The first column shows the number of distinct tags in XML documents. The second column shows the maximum depth of XML documents. NITF is rather shallow, while Treebank is deeper and has many recursions in elements.

To test the system, we required generators for both documents and queries. For documents, we generated the data sets from the NITF and Treebank DTDs using a modified version of the IBM's XML Generator [35]. We introduced the parameter *MaxValue* which determines the number of values that the data of elements and attributes of elements can take. We used the default values for the other parameters.

In each data set, documents were grouped by their sizes in bytes. In subsequent discussions, these document groups will be referred to as "`5k`", "`10k`", "`20k`" and "`30k`". Each document group contained 200 XML documents, and all the reported experimental results were averaged over the entire set of documents.

To generate twig patterns expressed in the XPath language, we used a modified version of the XPath generator from the YFilter package [9]. The characteristics of the parameters and their values used to generate twig patterns as workload are summarized in Table 2. The parameter L bounds the maximum depth of the twig pattern and is set to 6 for NITF and 10 for Treebank. We varied the twig patterns that were indexed by the filtering system from 50,000 to 150,000 in steps of 25,000. The number of value-based predicates was varied from 1 to 4. The parameters p_p, p_e, p_n and p_o determine the probability of predicates, an equality operator, a non-equality operator and an OR operator in each twig pattern, respectively. The parameter θ controls the skewness of the Zipf distribution [36] used for selecting element names and data values. The element names and data values were chosen from uniform distribution at the value of 0. Otherwise, the choice of the element names and data values was skewed according to the value of θ.

7.1.2. Environments

All experiments were performed on a 2.4 GHz Pentium IV machine with 512 MB memory running Linux. The pFiST code was compiled with GNU g++ compiler version 3.3.2.

7.2. Effect of equality operators

In this experiment, we compare our algorithms with the YFilter system [9] which is the most popular XML filtering technique. Since traditional XML filtering systems have focused on processing structure matching of twig patterns, we chose YFilter, as it has been also shown to support value-based predicates. However, YFilter supports only equality operators on attributes and text data in value-based predicates, hence we generate twig patterns containing only equality operators. The effects of other operators will be analyzed in Section 7.3.

When pFiST was compared with YFilter, their performance trends were measured in *scaleup* for fair comparison, which was used in [1]. This is because YFilter (obtained from the University of California at Berkeley) is implemented in Java, while

Table 2
Parameters for synthetic twig patterns

Parameter	Description	Values
N_t	Number of twig patterns	50,000–150,000
N_p	Number of value-based predicates in a twig pattern	1–4
L	Maximum depth of a twig pattern	6 or 10
p_p	Probability of predicates	0.6–1.0
p_e	Probability of an equality operator	0.0–0.5
p_n	Probability of a non-equality operator	$1 - p_e$
p_o	Probability of an OR operator	0.0–0.5
θ	Skewness of element names and data values	0.0–1.0

pFiST is implemented in C++ with Xerces XML Parser version 2.5.0 [37]. To measure the scaleup performance, we used the following formula:

$$\text{scaleup} = \frac{tAvg - tAvg_{\text{base}}}{x - x_{\text{base}}}, \qquad (1)$$

where *tAvg* is the filtering time measured for the case under observation at *x* and $tAvg_{\text{base}}$ is the filtering time measured for the base case at x_{base}. We assume that the *x*-axis grows in steps of 1 for all aspects of scalability.

7.2.1. Varying sizes of input XML documents

We measured the scalability of the system as the sizes of XML documents. The results are summarized in Fig. 16. Along the *x*-axis, we show the increase in the documents sizes by using data sets "`10k`", "`20k`" and "`30k`". The scaleup of YFilter grew quicker than that of pFiST indicating that YFilter's filtering cost increased much faster than pFiST. We also observed that the gap in the scaleup between YFilter and pFiST was widened as the size of the documents and the number of value-based predicates per twig pattern (N_p) were increased. These results demonstrate that pFiST scales better then YFilter, as the document sizes and N_p increase.

7.3. Effect of various operators

Existing techniques such as YFilter do not support non-equality operators and AND/OR operators. Thus due to the lack of a comparable system, we only present the performance evaluation of pFiST for various twig patterns containing different operators.

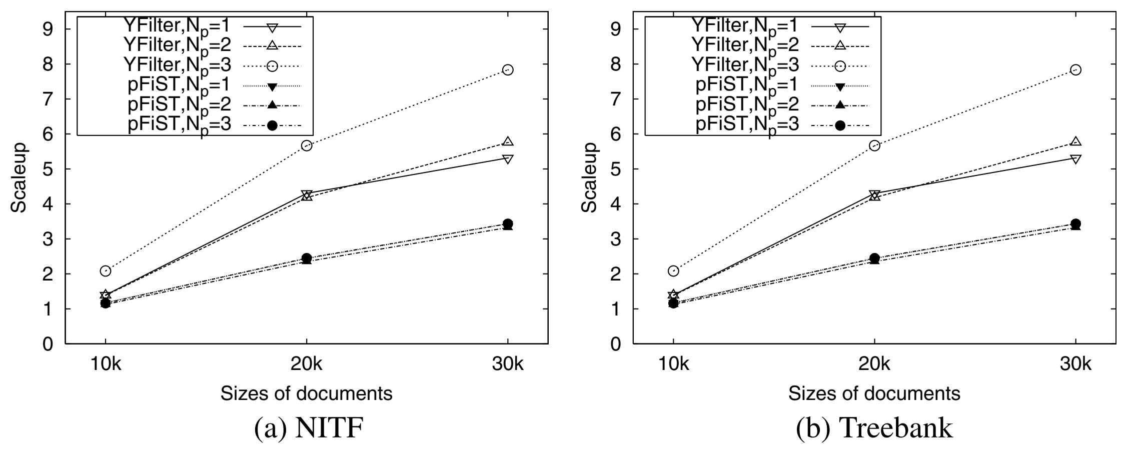


Fig. 16. Varying sizes of input documents.

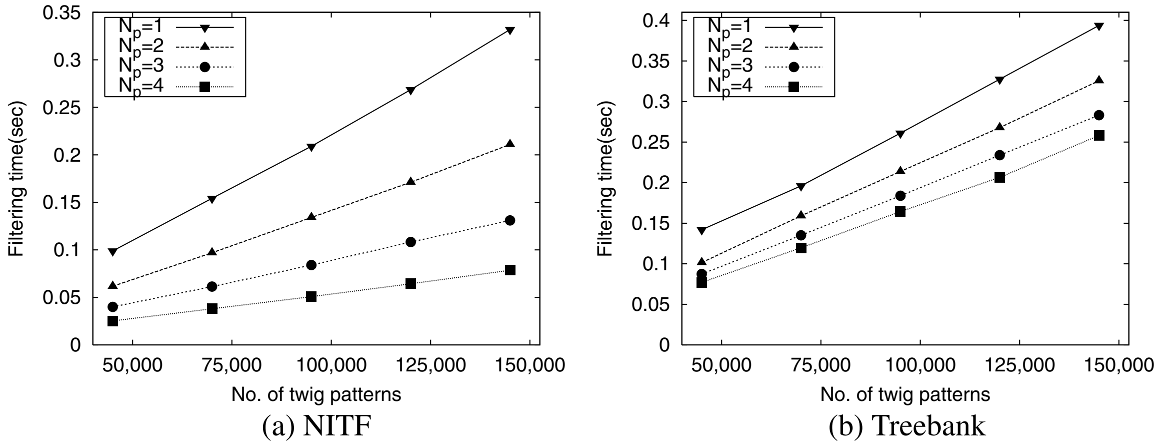


Fig. 17. Varying the number of twig patterns when the number of value-based predicates is varied from 1 to 4 for each twig patterns.

7.3.1. Varying the number of twig patterns

We measured the scalability of the system as the number of twig patterns and the number of value-based predicates in each twig pattern increase. Fig. 17 shows the filtering time for various number of twig patterns. The filtering time for both NITF and Treebank data sets grew as the number of twig patterns increase. We observed that the filtering time grew slowly as the number of value-based predicates increased. If the number of value-based predicates is large, the number of candidate twig patterns is reduced during the subsequence matching, and this yields a decrease in the filtering time.

7.3.2. Varying sizes of input XML documents

We measured the scalability of the system by varying the sizes of input XML documents. The results are summarized in Fig. 18. Along the x-axis, we show the increase in the documents sizes by using data sets “5k”, “10k” and “20k”. For each of the plots, the number of predicates per twig pattern (N_p) is varied from 1 to 3. The filtering time of both NITF and Treebank data sets grew linearly as the sizes of documents increase. We also observed that filtering time grew more slowly as the number of predicates increased.

7.3.3. Varying the number of value-based predicates

In this experiment, we investigate the performance advantage of the pFiST system as the number of predicates per twig pattern is increased. Fig. 19 summarizes the filtering time as the number of predicates is varied. The number of twig patterns is fixed to 150,000. Let us analyze the results shown in Fig. 19. The filtering time for both data sets displays a downward trend. This is consistent with the trend in Fig. 17. The number of branches is increased as the number of predicates are increased, which causes smaller number of candidate twig patterns. This explains the downward tendency.

Another observation from Fig. 19 is that the filtering time of NITF data set decreased considerably, whereas that of Treebank decreased slightly. The reason is explained from the characteristics of data sets. As shown in Table 1, NITF data sets

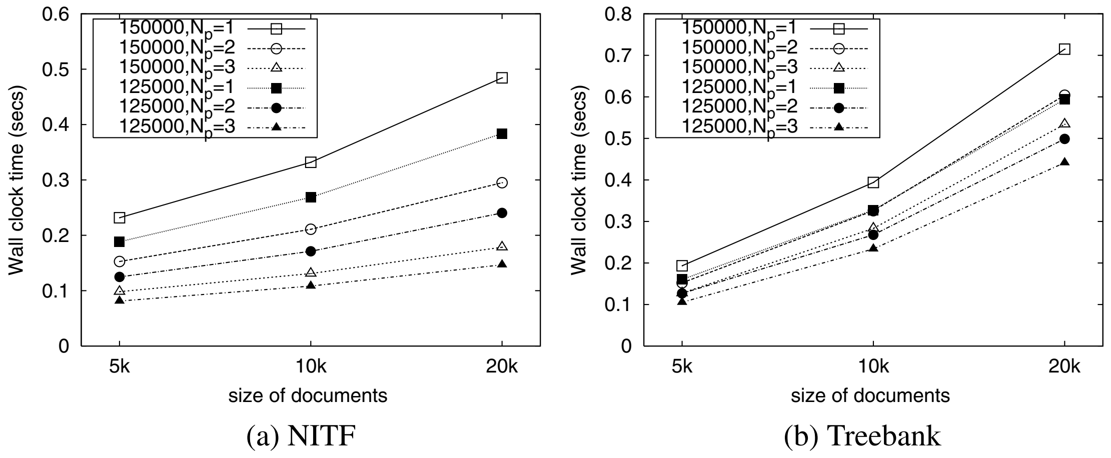


Fig. 18. Varying sizes of input documents.

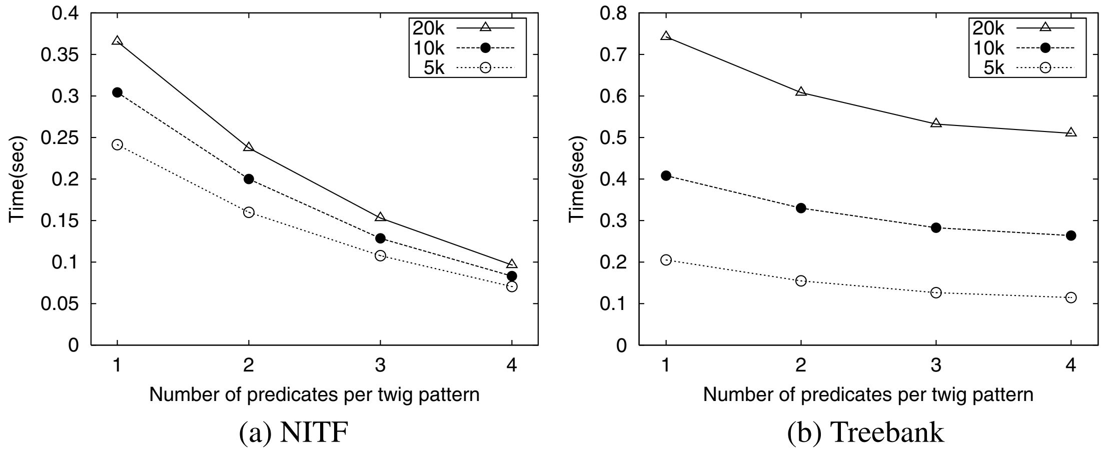


Fig. 19. Varying the number of predicates when the number of twig pattern is fixed to 150,000.

have a smaller number of elements in twig patterns than Treebank data sets. Therefore, NITF data sets are affected more considerably than Treebank data sets in terms of the number of value-based predicates.

7.3.4. Varying the probability of equality operators

To see the effect of the probability of an equality operator, we conducted experiments varying the probability from 0.1 to 0.5. We assumed that $p_e + p_n = 1$. Thus the lower probability of an equality operator means the higher probability of non-equality operators. We set all the parameters to default values except p_e and p_n.

Fig. 20 shows the results for the case when the probability of the equality operator in each query has varied. As the probability of the equality operator increases, the filtering time is decreased. It is due to the fact that the chance to be the candidate match during the subsequence matching is reduced as the twig pattern has a large number of equality operators in value-based predicates.

7.3.5. Varying the probability of predicates

In this experiment, we have measured the influence of selectivity of value predicates. In order to focus on the effect of selectivity of predicates, first we generate the twig pattern by setting the probability of predicates (p_p) to 1.0. Then, we deleted some value-based predicates of twig patterns according to the probability of predicate. Thus twig patterns used in these experiments have the same structures and have different value-based predicates. By changing the probability of predicates, we can see the effect of selectivity of value predicates.

We observed a downward trend in the filtering time as the probability of predicates decreased in Fig. 21 and also observed that more significant changes occurred in the filtering time for Treebank than NITF. This is explained from Table 3. Average number of matched twig patterns decreased as the probability of predicate decreased. This explains the downward trend in the filtering time. In addition, the reduction rate of Treebank is more severe than that of NITF. This difference caused the significant decrease in the filtering time for Treebank.

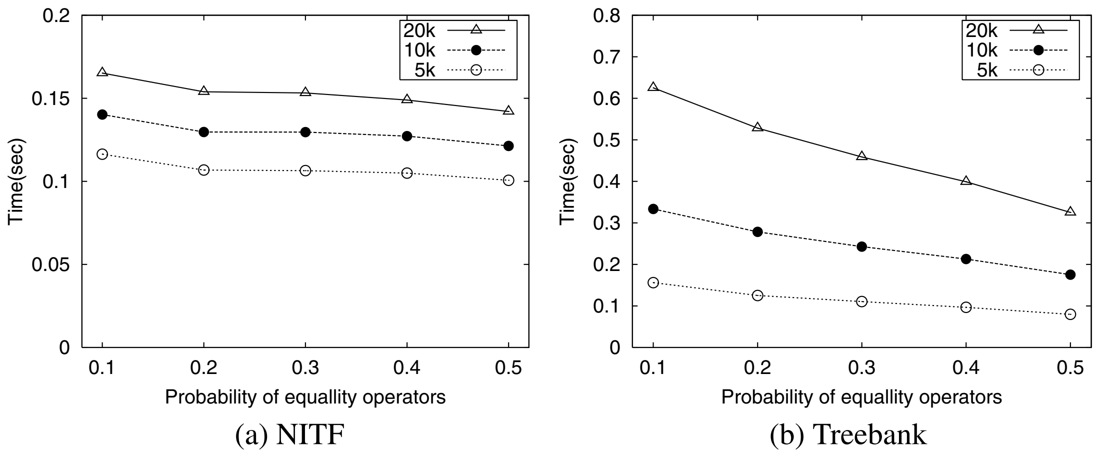


Fig. 20. Varying the probability of equality operators when the number of twig pattern is fixed to 150,000.

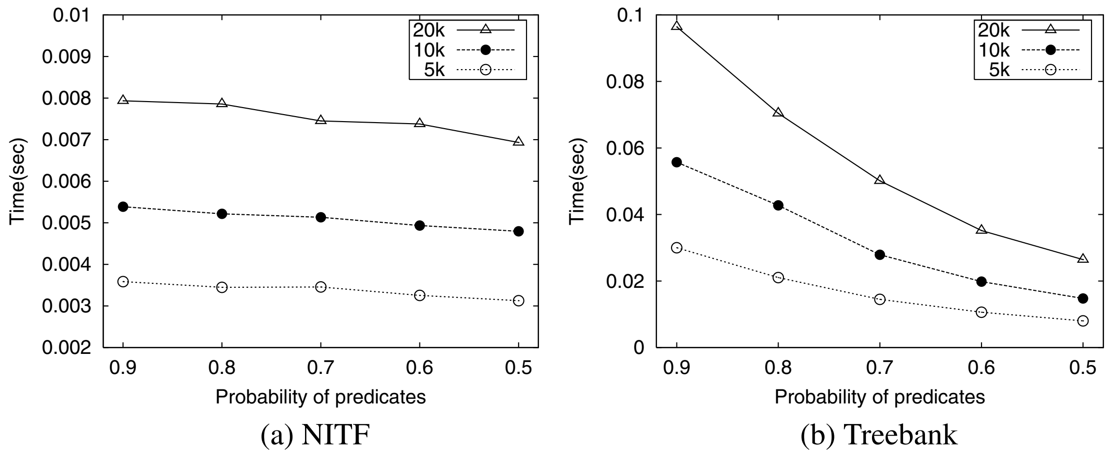


Fig. 21. Varying the probability of predicates when the number of twig pattern is fixed to 150,000.

Table 3
Average number of matched twig patterns to input documents

Probability of predicates	NITF			Treebank		
	5k	10k	20k	5k	10k	20k
0.9	291.235	324.92	391.88	1818.1	2675.51	3474.55
0.8	267.95	302.15	364.46	1485.94	2187.72	2840.87
0.7	253.19	282.395	339.07	1218.03	1788.92	2325.93
0.6	219.285	248.58	299.895	968.32	1422.71	1865.26
0.5	202.245	225.805	269.775	752.13	1101.38	1450.61

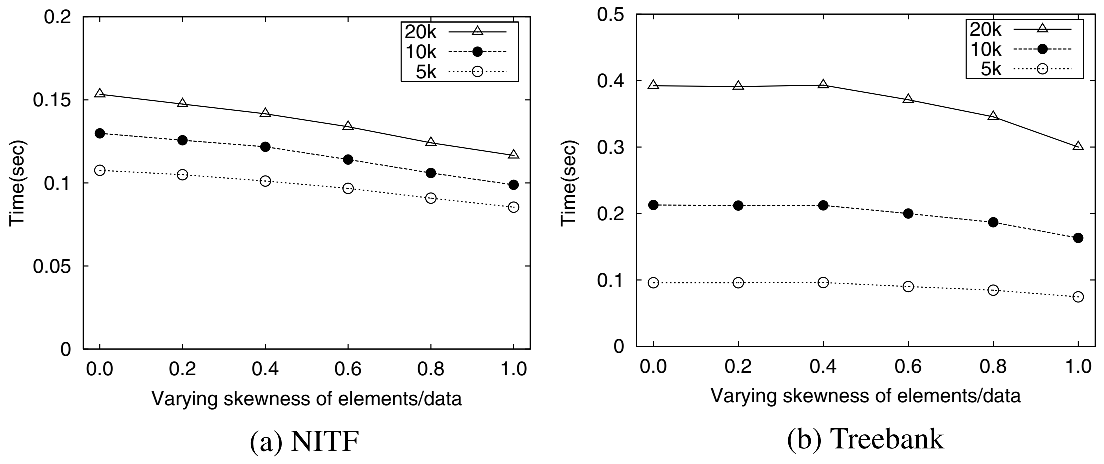


Fig. 22. Varying the skewness of element names and data values when the number of twig pattern is fixed to 150,000.

7.3.6. Varying the skewness of elements names and data values

To see the effect of distribution of elements names and data values, we conducted experiments varying the skewness parameter θ from 0.0 to 1.0. Note that the element names and data values of twig patterns are chosen according to the skewness parameter, but those of XML documents are uniformly selected. Fig. 22 shows the results. The graphs show that the performance of pFiST decreased slightly with highly skewed element names and data values. This is because that XML documents are not skewed and pFiST indexed the sequence nodes which also contain structural information rather than element names and data values.

7.3.7. Varying the probability of OR operators

We have measured the influence of the OR operators on the performance of the filtering algorithms. In order to focus on the effect of OR operators, we first generate the twig patterns which contain only AND operators then replace AND operators with OR operators according to the probability p_o; that is, the twig patterns used in this experiments have the same structures.

First, we measured the time for parsing twig patterns to see the effects of OR operators. As explained in Section 6, we need the simplification step for parsing twig patterns with AND operators and both the decomposition step and simplification step for parsing twig patterns with OR operators.

As expected, the total number of twig patterns was increased after the decomposition step with the probability of OR operators, which is described in Table 4. Fig. 23a shows the ratio for parsing twig patterns with OR operators to parsing twig patterns with only AND operators. Although the total number of twig patterns was increased as the probability of OR operators was increased, the total parsing time for twig pattern containing OR operators was slightly decreased.

Table 4
The number of twig patterns after the decomposition

Probability of OR operators	NITF	Treebank
0.0	150,000	150,000
0.1	174,314	175,595
0.2	195,802	198,068
0.3	215,021	217,094
0.4	230,898	232,990
0.5	245,025	246,551

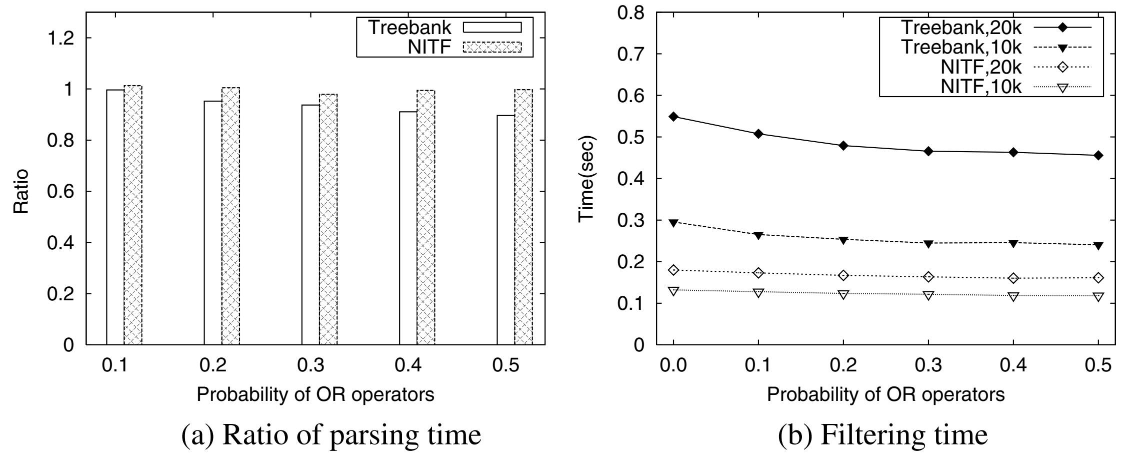


Fig. 23. Varying the probability of OR operators.

This is explained as follows: When we parse twig patterns with OR operators, the decomposition of twig patterns with OR operators makes a large number of twig patterns. However, the decomposed twig pattern has a small number of branches. This reduces costs for the simplification step of decomposed twig patterns and the overall parsing time.

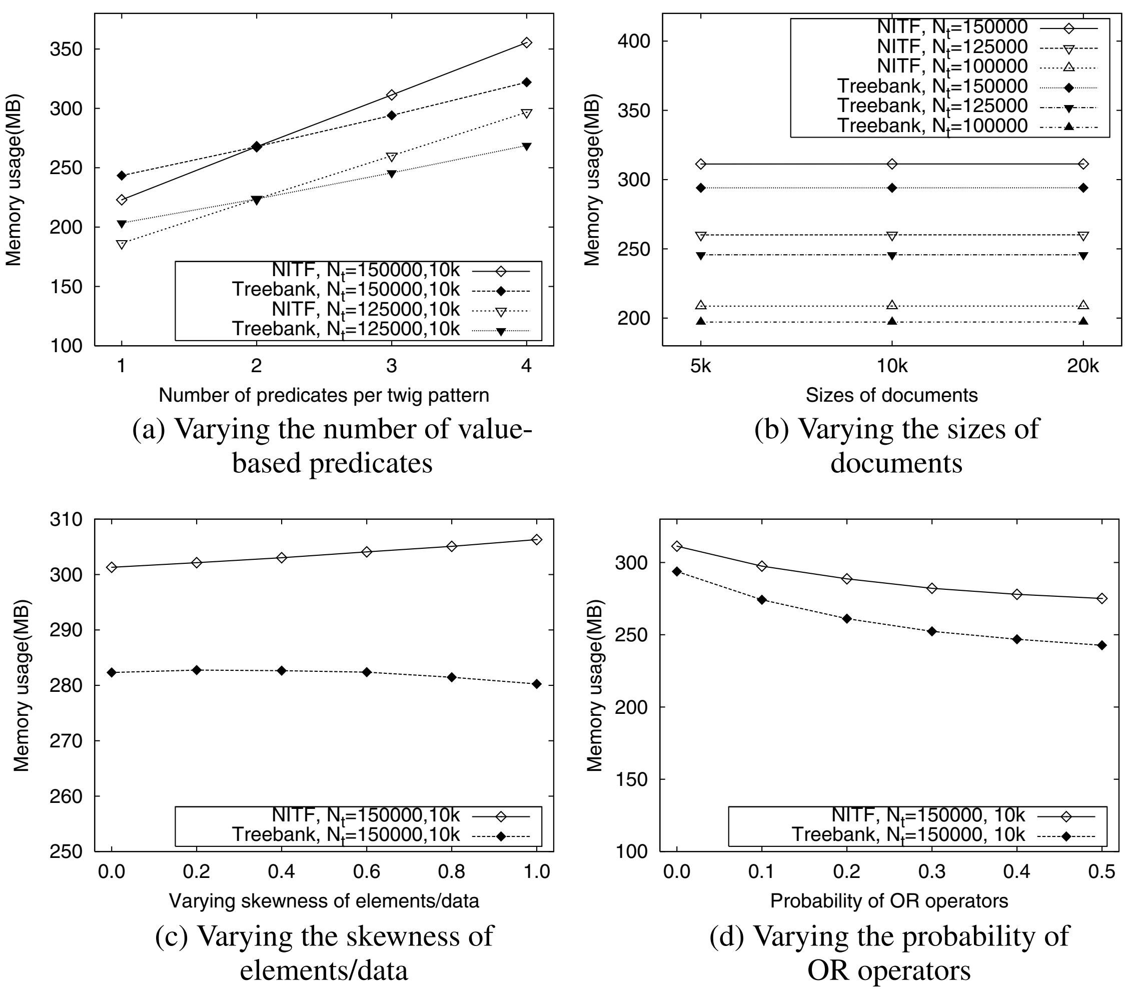


Fig. 24. Memory usage.

Second, we measured the filtering time with varying the probability of OR operators. The observation is that the filtering time for NITF did not change noticeably and the filtering time for Treebank decreased slightly. This result explains the effectiveness of the decomposition of twig patterns with OR operators although it makes more number of twig patterns.

7.3.8. Memory usage

In this experiment we investigate the impact of processing value-based predicates on memory usage. We measured the total memory consumptions by reading the statistics given in /proc/self/statm, which provides memory statistics of processes.

Fig. 24a depicts the memory usage of the pFiST system as the number of predicates increases. As expected, the memory usage is increased with the number of predicates is increased. Another observation is that the memory usage of NITF data sets is changed more considerably than that of Treebank data sets due to the characteristics of the DTD.

Fig. 24b shows the effect of input document sizes on memory usage. The sizes of the input XML documents ranged from "5k" to "20k" and the number of value-based predicates per twig pattern (N_p) was fixed to 3. The memory consumption by pFiST was insensitive to the sizes of input documents. This is because the size of a runtime stack is bounded by the depth of an input document. The average depth of input documents is 10 for NITF and 20 for Treebank.

Fig. 24c depicts the memory usage for varying the skewness of element names and data values. As explained in Section 7.3.6, pFiST is less sensitive to skewness of element names and data values. Thus, the memory usage is changed slightly with the skewness.

Fig. 24d shows the memory usage for various probability of OR operators. Since the decomposition step makes a twig pattern with complex structures to multiple twig patterns with simple structures, the memory usage is decreased as the probability of OR operator is increased.

8. Conclusions

In XML filtering systems, twig patterns (or user profiles) contain structural patterns as well as value-based selections. Although existing XML filtering systems can process structure matching of twig patterns efficiently, they provide limited or no support for value-based predicates containing various operators. In this paper, we have presented the design of the pFiST system for value-based predicate filtering of XML documents. We have developed an algorithm, which can process structure matching and also handle value-based predicates in twig patterns. The value-based predicates can include equality operator (=) and non-equality operators ($<, >, <=, >=$). Value-based predicates can contain AND and OR operators and pFiST processes them via a simplification and decomposition approach. The experimental results conducted over different data sets and twig patterns demonstrate that pFiST can efficiently support value-based predicate filtering of XML documents.

Acknowledgements

We are grateful to the anonymous referees for their valuable comments.

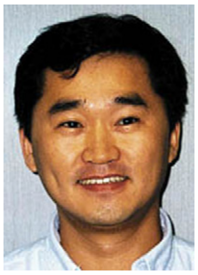
References

[1] J. Kwon, P. Rao, B. Moon, S. Lee, FiST: scalable XML document filtering by sequencing twig patterns, in: Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005, pp. 217–228.
[2] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernández, M. Kay, J. Robie, J. Siméon, XML path language (XPath) 2.0 W3C working draft 16, Tech. Rep. WD-xpath20-20020816, World Wide Web Consortium, August 2002. <http://www.w3.org/TR/xpath20>.
[3] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: Proceedings of the 2002 ACM-SIGMOD Conference, Madison, Wisconsin, 2002, pp. 310–321.
[4] J. Lu, T.W. Ling, C.Y. Chan, T. Chen, From region encoding to extended dewey: on efficient processing of XML twig pattern matching, in: Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005, pp. 193–204.
[5] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, K.S. Candan, Twig^{2}Stack: bottom-up processing of generalized-tree-pattern queries over XML documents, in: Proceedings of the 32nd VLDB Conference, Seoul, Korea, 2006, pp. 283–294.
[6] H. Jiang, H. Lu, W. Wang, Efficient processing of XML twig queries with OR-predicates, in: Proceedings of the 2004 ACM-SIGMOD Conference, ACM, New York, NY, USA, 2004, pp. 59–70.
[7] P. Rao, B. Moon, PRIX: indexing and querying XML Using Prüfer sequences, in: Proceedings of the 20th IEEE International Conference on Data Engineering, Boston, MA, 2004, pp. 288–299.
[8] M. Altinel, M.J. Franklin, Efficient filtering of XML documents for selective dissemination of information, in: Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000, pp. 53–64.
[9] Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, P. Fischer, Path sharing and predicate evaluation for high-performance XML filtering, ACM Trans. Database Syst. 28 (4) (2003) 467–516.
[10] B. Ludäscher, P. Mukhopadhyay, Y. Papakonstantinou, A transducer-based XML query processor, in: Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002, pp. 227–238.
[11] F. Peng, S.S. Chawathe, XPath queries on streaming data, in: Proceedings of the 2003 ACM-SIGMOD Conference, ACM Press, San Diego, CA, 2003, pp. 431–442.
[12] A.K. Gupta, D. Suciu, Stream processing of XPath queries with predicates, in: Proceedings of the 2003 ACM-SIGMOD Conference, ACM Press, San Diego, CA, 2003, pp. 419–430.
[13] C.Y. Chan, P. Felber, M.N. Garofalakis, R. Rastogi, Efficient filtering of XML documents with XPath expressions, in: Proceedings of the 18th IEEE International Conference on Data Engineering, San Jose, CA, 2002, pp. 235–244.

[14] N. Bruno, L. Gravano, N. Koudas, D. Srivastava, Navigation- vs. index-based XML multi-query processing, in: Proceedings of the 19th IEEE International Conference on Data Engineering, Bangalore, India, 2003, pp. 139–150.
[15] C. Byun, K. Lee, S. Park, A keyword-based filtering technique of document-centric XML using NFA representation, Int. J. Appl. Math. Comput. Sci. 4 (3) (2007) 136–143.
[16] A. Raj, P. Kumar, Branch sequencing based XML message broker architecture, in: Proceedings of the 23rd IEEE International Conference on Data Engineering, Istanbul, Turkey, 2007, pp. 656–665.
[17] M.M. Moro, P. Bakalov, V.J. Tsotras, Early profile pruning on XML-aware publish–subscribe systems, in: Proceedings of the 33rd VLDB Conference, Vienna, Austria, 2007, pp. 866–877.
[18] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, J. Myllymaki, Implementing a scalable XML publish/subscribe system using a relational database system, in: Proceedings of the 2004 ACM-SIGMOD Conference, Paris, France, 2004, pp. 479–490.
[19] S. Hou, H.-A. Jacobsen, Predicate-based filtering of XPath expressions, in: Proceedings of the 22nd IEEE International Conference on Data Engineering, Atlanta, Georgia, USA, 2006, p. 53.
[20] K.S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, D. Agrawal, AFilter: adaptable XML filtering with prefix-caching suffix-clustering, in: Proceedings of the 32nd VLDB Conference, Seoul, Korea, 2006, pp. 559–570.
[21] G. Gou, R. Chirkova, Efficient algorithms for evaluating XPath over streams, in: Proceedings of the 2007 ACM-SIGMOD Conference, Beijing, China, 2007, pp. 269–280.
[22] C. Koch, S. Scherzinger, M. Schmidt, XML prefiltering as a string matching problem, in: Proceedings of the 24th IEEE International Conference on Data Engineering, Cancun, Mexico, 2008, pp. 626–635.
[23] S. Amer-Yahia, L.V.S. Lakshmanan, S. Pandit, FleXPath: flexible structure and full-text querying for XML, in: Proceedings of the 2004 ACM-SIGMOD Conference, Paris, France, 2004, pp. 83–94.
[24] P. Rao, B. Moon, Sequencing XML data and query twigs for fast pattern matching, ACM Trans. Database Syst. 31 (1) (2006) 299–345.
[25] A. Balmin, F. Özcan, K.S. Beyer, R. Cochrane, H. Pirahesh, A framework for using materialized XPath views in XML query processing, in: Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004, pp. 60–71.
[26] L. Chen, S. Wang, E.A. Rundensteiner, Replacement strategies for XQuery caching systems, Data Knowledge Eng. 49 (2) (2004) 145–175.
[27] C.Y. Chan, Y. Ni, Efficient XML data dissemination with piggybacking, in: Proceedings of the 2007 ACM-SIGMOD Conference, Beijing, China, 2007, pp. 737–748.
[28] T. Milo, T. Zur, E. Verbin, Boosting topic-based publish–subscribe systems with dynamic clustering, in: Proceedings of the 2007 ACM-SIGMOD Conference, Beijing, China, 2007, pp. 749–760.
[29] M. Hong, A.J. Demers, J. Gehrke, C. Koch, M. Riedewald, W.M. White, Massively multi-query join processing in publish/subscribe systems, in: Proceedings of the 2007 ACM-SIGMOD Conference, Beijing, China, 2007, pp. 761–772.
[30] B. Chandramouli, J. Phillips, J. Yang, Value-based notification conditions in large-scale publish/subscribe systems, in: Proceedings of the 33rd VLDB Conference, Vienna, Austria, 2007, pp. 878–889.
[31] D. Megginson, Simple API for XML. <http://sax.sourceforge.net/>.
[32] H. Prüfer, Neuer Beweis eines Satzes über Permutationen, Arch. Math. Phys. 27 (1918) 142–144.
[33] NITF, NITF: news industry text format. <http://www.nitf.org/>.
[34] Treebank, The Penn Treebank Project. <http://www.cis.upenn.edu/~treebank/>.
[35] A.L. Diaz, D. Lovell, XML generator, September 1999. <http://alphaworks.ibm.com/tech/xmlgenerator>.
[36] G.K. Zipf, Human Behaviour and the Principle of Least Effort: An Introduction to Human Ecology, Addison-Wesley, 1949.
[37] Apache, Apache Xerces C++ Parser. <http://xml.apache.org/xerces-c/>.

Joonho Kwon is a Ph.D. candidate in the School of Electrical Engineering and Computer Science, Seoul National University, Seoul, Korea. He received his M.S. and B.S. degrees in the Department of Computer Engineering from Seoul National University, Seoul, Korea, in 1999 and 2001, respectively. His current research interests include XML filtering, XML indexing and query processing and Web services.

Praveen Rao is an Assistant Professor of Computer Science and Electrical Engineering at the University of Missouri-Kansas City. His research interests include XML indexing and query processing, XML filtering and stream processing, XML data management in peer-to-peer systems, and indexing graph databases and graph mining. He received his Ph.D. and M.S. degrees in Computer Science from the University of Arizona in 2007 and 2001, respectively. He received his B.E. degree in Computer Engineering from the University of Pune (India) in 1999. Praveen was a software engineer for Amazon.com during 2001–2002.



Bongki Moon is an Associate Professor of Computer Science at the University of Arizona. His current research areas of interest are XML indexing and query processing, information streaming and filtering, spatial and temporal databases, and parallel and distributed processing. He received his Ph.D. degree in Computer Science from University of Maryland, College Park, in 1996, and his M.S. and B.S. degrees in Computer Engineering from Seoul National University, Korea, in 1985 and 1983, respectively. He was a communication systems research staff at Samsung Electronics Corp. and Samsung Advanced Institute of Technology, Korea, from 1985 to 1990. He received the National Science Foundation CAREER Award in 1999.

Sukho Lee received his BA degree in Political Science and Diplomacy from Yonsei University, Seoul, Korea, in 1964 and his M.S. and Ph.D. in Computer Sciences from the University of Texas at Austin in 1975 and 1979, respectively. He is currently a professor of the School of Computer Science and Engineering, Seoul National University, Seoul, Korea, where he has been leading the Database Research Laboratory. He served as the president of Korea Information Science Society in 1994. He served as the honorary chair in the International Symposium on Database Systems for Advanced Applications (DASFAA), 2004 and in the International Conference on Very Large Data Bases (VLDB), 2006. His current research interests include database management systems, spatial database systems, multimedia database systems and web services.