XML filtering with XPath expressions containing parent and ancestor axes

Bo Ning\textsuperscript{a,*}, Chengfei Liu\textsuperscript{b}

\textsuperscript{a}Dalian Maritime University, Liaoning, China
\textsuperscript{b}Swinburne University of Technology, VIC, Australia

\begin{abstract}
More and more XML data is generated and used for data exchange. In this paper, we address the problem of filtering XML documents with large number of XPath expressions, which may contain ‘ancestor’ and ‘parent’ axes. XPath expressions with these axes are more powerful and flexible for users to describe their interests in publish/subscribe systems. First, we analyze the characteristics of the ‘parent’ axis and propose a series of rules to eliminate it in XPath expressions. Then we propose a new index structure called NIndex, which is designed to efficiently store and index large number of XPath expressions. NIndex offers several features which make it especially attractive for the large scale selective dissemination of information, including the ability to handle complex XPath expressions with ‘ancestor’ and ‘parent’ axes, and efficient pruning. Based on NIndex, we design a new filtering algorithm with low complexity for our problem. Our experiment results show that our algorithm performs well across a range of XPath expressions and documents.
\end{abstract}

\section{1. Introduction}

The proliferation of the Internet, and the exploding volume of information available on the Internet has fueled the development of a wide range of new applications based on selective dissemination of information (SDI) \cite{3}. These applications include stock exchange, advertisement systems, electronic personalized newspapers, online shopping, online auctioning and entertainment delivery, and require timely distribution of data to a large set of customers. In an SDI (or publish/subscribe) system, users subscribe to a data server with continuous queries or profiles that are expressed in some well-defined languages for expressing their information needs. The SDI system performs the matching task and ensures timely delivery of published data to all interested subscribers.

With XML becoming the standard of data representation and exchange on the Internet, effective and efficient methods have been studied for searching useful information from ordinary and probabilistic XML documents by both structured queries and keyword queries \cite{2,7,25,26,30,33,42}. XML is also adopted for content-based publish/subscribe systems because published XML messages have flexible document structures and subscription rules can be expressed by a powerful language such as XPath \cite{9} and XQuery \cite{5}. In an XML publish/subscribe system, XML filtering is a core part, in which continuously arriving XML documents are routed to users according to their subscriptions expressed as queries. These queries typically specify patterns of selection on multiple elements.

Tremendous effort has been put into efficiently filtering XML streams with a collection of XPath expressions \cite{3,8,12,15–24,32,36,41}, including XFilter \cite{3}, YFilter \cite{12}, XTree \cite{8}, XPush \cite{18} and the lazy filtering algorithm (LF) \cite{16}. Most of these systems only support forward axes such as child and descendant axes.
However in real applications, XML queries submitted by users may contain predicates with ancestor and parent axes. Firstly, documents for dissemination usually have different schemas [1,29,39], consequently different relationships between ancestor tags of a certain tag $t$ in a query may exist in different documents. Sometimes, users are unconcerned about exact relationships between those ancestor tags as long as they are ancestors of tag $t$. Moreover, users may not always be aware of the schemas of XML documents. For the above reasons, users have to use the predicates with ancestor or parent axes to constrain the answers.

For example, Alice wants to buy some books from a book selling website. She is interested in those computer books authored by “John”, and published by “Addison-Wesley”. The documents (a) and (b) shown in Fig. 1 all satisfy the Alice’s request, and therefore both of them should be disseminated to Alice. This can be realized by submitting the following query $Q$ with an ancestor axis easily.

$$Q : // \text{Computer//book/ancestor :: Addison-Wesley}\  
// \text{author[name = “John”]}$$

However, if Alice does not use the ancestor axis and submits the following query $Q_a$ instead.

$$Q_a : //\text{Computer/Addison-Wesley//book//author[name = “John”]}$$

then only document (a) will be returned because the forward axis in $Q_a$ can express only one kind of relationship between the ancestor tags “Computer” and “Addison-Wesley” of the book tag, i.e. “Computer” is the ancestor of “Addison-Wesley”. Document (b) will not be returned because the ancestor tags “Computer” and “Addison-Wesley” have a different relationship, i.e. “Addison-Wesley” is the ancestor of “Computer”. From the example, we can clearly see that using the ancestor axis provides more expressive power than not using it. For improving flexibility, we also support the parent axis.

In this paper, we propose a new index structure named NIndex, which supports efficient filtering of XML documents based on XPath expressions (XPEs) which contain ancestor and parent axes. NIndex structure offers several novel features that make it especially attractive for large-scale publish/subscribe systems.

Firstly, we analyze the characteristics of the ‘parent’ axis and propose a series of rules to eliminate it from those XPEs with the ‘parent’ axis. Dan Olteanu et al. proposed rewriting rules [34] to transform absolute XPath location paths with reverse axes into equivalent reverse axis free paths, and to rewrite all axes to a minimum set of forward axes. However, the transformed XPEs contain join operations, which are not convenient to handle in an XML document filtering system. In this paper, we only consider transformation of the ‘parent’ axis and transformed XPEs contain only simple axes including ‘ancestor’, ‘descendant’ and ‘child’.

Secondly, to improve filtering performance, our NIndex scheme captures common sub-expressions among multiple XPEs. Xtrie [8] is indexed on a set of substrings by using a substring table in which levels of elements are recorded. However, the level attribute may cause redundancy despite the identical sub-patterns. Different occurrences of the same sub-pattern are treated as different items in the substring table if they appear at different levels. Therefore our index is more concise to represent XPEs. It reduces the number of unnecessary index probes and avoids redundant matchings.

Thirdly, NIndex is designed to support effective filtering of XML documents based on complex XPEs containing ancestor and parent axes. Xaos [4] is a query processing system that also supports ancestor and parent axes, and it uses the X-dag structure to express queries. However the X-dag structure can only handle a single XPE, and does not provide support to handle the case of multiple XPEs.

Finally, the experiments show that our algorithm is time-efficient.

The remainder of this paper is organized as follows. In Section 2, we introduce the background and give the definition of a PXPE-tree. Section 3 discusses how to eliminate the parent axis in a PXPE-tree. In Section 4, we introduce the structure of NIndex which is the index for XPEs, and design updating algorithms for maintaining NIndex. In Section 5, we propose a
filtering algorithm to address the XPE retrieval problem, which is based on NIndex. Section 6 shows our experimental results. Section 7 discusses related works and compare them with our work. Section 8 concludes the paper.

2. Background and definitions

2.1. Data model for XML streams

An XML document can be modeled as a rooted, labeled, and ordered tree, which we call an XML data tree. Each node in the data tree corresponds to an element, attribute, or text value in the XML document. An XML streaming algorithm accepts imputed XML documents as a stream of SAX\[6\] events. Two core SAX events are startElement(qname) and endElement(qname), which are activated, respectively, when the opening or closing tag of a streaming element arrives, and accept the name of that element, qname, as the input parameter.

2.2. XPath and axes

XPath \[9\] is a popular language for querying XML data. It has been used in many XML applications and in some other languages for querying and transforming XML data, such as XQuery and XSLT. The primary purpose of XPath is to address parts of an XML document, by navigating through the hierarchical structure of an XML document. There are 13 axes in XPath, such as ‘ancestor’ and ‘descendant’.

In this paper we address a subset of XPath known as Partial XPath (see Fig. 2). This paper does not explicitly process the attribute axis @, as it can be handled in a way similar to the child axis/.

2.3. XML-based SDI system

An XML-based SDI system filters and delivers XML documents based on user interests. There are two main sets of inputs to the system: user profiles and XML documents. User profiles describe the information preferences of individual users. They will be converted into a format that can be efficiently stored and evaluated by the filter engine. In this paper these profiles are specified as partial XPEs, which are applied to all incoming documents in the format of XML streams.

The problem of effectively identifying the subscriptions that match an incoming XML document can be defined as: Given a large collection $P$ of XPEs and an input XML document $D$, find the subset of XPEs in $P$ that match $D$.

Fig. 3 shows the structure of an XML-based SDI system. The core technique for expediting XPE retrieval is to construct an appropriate index structure on a given collection of XPath expression subscriptions.

2.4. PXPE-tree

A partial XPath expression (PXPE) is presented as a rooted unordered tree, called PXPE-tree, $T = (V_T, E_T)$ with labeled vertices and edges. We use the term PXPE-node to refer to the vertices of a PXPE-tree. For each NodeTest in the expression, there is a PXPE-node in the PXPE-tree labeled with the NodeTest. Each PXPE-node has a unique incoming edge, which is labeled with the axis specified before the NodeTest. One of the PXPE-nodes is designated to be the output PXPE-node. Functions, label: $V_T \rightarrow \text{String}$, and axis: $E_T \rightarrow \{\text{ancestor}, \text{descendant}, \text{parent}, \text{child}\}$ return the labels associated with the PXPE-nodes and edges respectively. Fig. 4 provides examples of PXPE-trees.

2.5. Common sub-pattern

Before giving the definition of a common sub-pattern, we assume that the symbol subtree($r$) expresses the subtree rooted at PXPE-node $r$ in PXPE-tree $T$. Because the PXPE-tree is unordered, subtree($r$) disregards the order of branches in a branch node.

**Definition 1** (Common sub-pattern). Given PXPE-trees $T_1$ and $T_2$, if PXPE-tree $T_C$ with $n$ as the root is a subtree($n$) of $T_1$ and $T_2$, we call $T_C$ a common sub-pattern of $T_1$ and $T_2$.

\[
\text{Path} ::= \text{Step} \mid \text{Path} \text{Step} \\
\text{Step} ::= \text{Axis} \text{NodeTest} \mid \text{Axis} \text{NodeTest} \mid [\text{Predicate}] \\
\text{Axis} ::= \text{’ancestor’} \mid \text{’descendant’} \mid \text{’parent’} \mid \text{’child’} \\
\text{NodeTest} ::= \text{name} \\
\text{Predicate} ::= \text{Path} \mid \text{Predicate ‘and’ Predicate} \mid \text{Predicate ‘or’ Predicate} \mid \text{’not’ Predicate}
\]

Fig. 2. Grammar of partial XPath.
Definition 2 (Highest common sub-pattern). Given PXPE-trees $T_1$ and $T_2$, $S_T = (T_{c_1}, \ldots, T_{c_n})$ is the set of common sub-patterns of $T_1$ and $T_2$. We call $T_{c_i}$ with the largest height the highest common sub-pattern of $T_1$ and $T_2$.

Two PXPE-trees may have many common sub-patterns, but only a single highest common sub-pattern. For example, in Fig. 4, subtree $(B)$ and subtree $(C)$ are common sub-patterns of $Q_1$ and $Q_2$, and subtree $(B)$ is the highest common sub-pattern. In $Q_5$ and $Q_6$, subtree $(A)$ is the highest common sub-pattern. In $Q_7$ and $Q_8$, subtree $(B)$ is the highest common sub-pattern. Subtree $(A)$ is not a common sub-pattern, because PXPE-node $A$ has an additional predicate with an ‘ancestor’ axis.

3. Elimination of ‘parent’ axis

The ‘parent’ axis specifies the child-parent relationship between two PXPE-nodes. In fact, the child-parent relationship can be expressed by the parent-child relationship using the ‘child’ axis. Therefore we do not need to design the respective part in the algorithm for the ‘parent’ axis in the XML document filtering, we simply transform the PXPE-trees with ‘parent’ axes to parent axis free XPEs. In this section, we propose a series of lemmas to eliminate the ‘parent’ axis in a PXPE-tree. Finally, we conclude that any ‘parent’ axis can be eliminated.

Lemma 1. There is one parent axis at most in out-edges of PXPE-node $A$; otherwise the PXPE-tree is invalid.

It is impossible for an element in XML document to have two parents, therefore a PXPE-node with more than one ‘parent’ axis cannot find any corresponding element in documents. So Fig. 5a is an invalid PXPE-tree.

Lemma 2. If PXPE-node $A$ is the root of a PXPE-tree and has a child $B$ whose axis is ‘parent’, PXPE-node $B$ can be placed as the root of the transformed PXPE-tree and $B$ has a child PXPE-node $A$ with ‘child’ axis.

From Fig. 5b, we can see that pattern $A[parent::B]$ is equivalent to $B/A$, because they represent the same relationship between $A$ and $B$ in XML documents.

Lemma 3. If PXPE-node $A$ has a child $B$ whose axis is ‘parent’ and PXPE-node $B$ has a child $C$ whose axis is ‘child’, PXPE-node $A$ can be placed as a child of $B$ with a ‘child’ axis. In the transformed PXPE-tree, $B$ has two children $A$ and $C$ with the ‘child’ axes.
Lemma 4. If PXPE-node A has a child B whose axis is ‘parent’ and PXPE-node B has a child C whose axis is ‘descendant’, PXPE-node A can be regarded as the child of B with a ‘child’ axis. In the transformed PXPE-tree, B has two children A with a ‘child’ axis and C with a ‘descendant’ axis. Lemmas 3 and 4 explore the cases where the PXPE-node with a ‘parent’ axis has a subtree connected with the ‘child’ or ‘descendant’ axes. The relationship between A and B has no impact on the subtree of B, therefore the parent axis can be eliminated by placing PXPE-node A as a child of B with a ‘child’ axis (see Fig. 5c and d). Notice that the twig-like PXPE-Tree in Fig. 5d only specifies the relationships between the parent PXPE-node and its child PXPE-nodes. The relationships among the children are not specified, and those children may be on the path of an answer.

Lemma 5. If the axis of PXPE-node A is ‘child’, no axis of any child PXPE-node of A can be ‘parent’; otherwise the PXPE-tree is invalid.

Lemma 6. If the axis of PXPE-node A is ‘descendant’, and C is A’s parent in a PXPE-tree, and A has a child PXPE-node B with ‘parent’ axis, the ‘parent’ axis of B can be eliminated by placing PXPE-node B between A and C, setting the axis of A to ‘child’, and setting the axis of B to ‘descendant’. In Fig. 5e, the incoming edge of A is the ‘child’ axis, which conflicts with the relationship between A and B. This is because the PXPE-tree expresses that A has more than one parent in XML documents. Therefore it is invalid. If the incoming edge is the ‘descendant’ axis (see the left PXPE-tree of Fig. 5f), and PXPE-node B is A’s parent, the PXPE-tree can be transformed into the right PXPE-tree of Fig. 5f, where the parent axis can also be eliminated.

Theorem 1. Any ‘parent’ axis in a PXPE-tree can be eliminated.

Proof 1. The simplest case of the ‘parent’ axis is A[parent:B]. It can be transformed into a parent axis free PXPE-tree by Lemma 2. Lemmas 3 and 4 consider the cases in which the PXPE-node B have subtrees, and the parent axis can be eliminated by placing PXPE-node A as B’s child. Lemmas 5 and 6 consider the cases where A is a child PXPE-node in a PXPE-tree. If the incoming edge of A is ‘child’, the PXPE-tree is invalid according to Lemma 5. If the incoming edge of A is ‘descendant’, the parent axis can be eliminated according to Lemma 6. At this point, a single ‘parent’ axis in a PXPE-tree can be eliminated. For a general PXPE-tree including multiple ‘parent’ axes, we can conduct the one parent axis elimination procedure multiple times to transform it into a parent axis free unless the PXPE-tree is invalid.

According to Theorem 1, any ‘parent’ axis can be eliminated, therefore we do not consider the ‘parent’ axis during the indexing of XPEs.
4. NIndex: indexing XPath expressions

In this section, we explain how to build NIndex, the index of XPEs. From Theorem 1, we can transform valid queries with the ‘parent’ axis into queries without the ‘parent’ axis. Therefore the queries indexed by NIndex do not consider the ‘parent’ axis. In the following, we first introduce the structure of NIndex, then discuss how to update in NIndex.

4.1. Structure of NIndex

We index PXPE-nodes in NIndex, and store only one copy of common sub-patterns. We cluster PXPE-nodes into tables by their tag names. Each table $T_t$ stores all PXPE-nodes with tag $t$, and each row in $T_t$ corresponds to a PXPE-node which is the root of a sub-pattern in one or more PXPE-trees. For searching the respective table with a tag name, a mapping table implemented in the form of a hash table is built over all $(\text{tagname}, p_{\text{_table}})$ pairs for retrieving the pointer to a table of any tag name. In table $T_t$, we use field ID to number the rows so that by using the tag name $t$ and the ID we can get a PXPE-node’s location and access the row. For efficient retrieval, we may index the rows by the ID field by using hash table or B + tree. Those PXPE-node tables reside in memory throughout the stream processing.

Each row in a tag table is used to describe a PXPE-node $c_n$ and the sub-pattern rooted at $c_n$. It includes four kinds of fields: (1) structural information of the PXPE-node, including axis, parent, pos, p_dc and p_a. The axis field records the axis of the current PXPE-node $c_n$. desc and anc are respectively short for ‘descendant’ and ‘ancestor’. The parent field is the $(\text{tagname}, \text{row_num})$ pair of the parent node of $c_n$ in the PXPE-tree. The information of the parent node is recorded in the $\text{row_num}^{\text{th}}$ row in tag table $T_{\text{tagname}}$. The pos field denotes the position of $c_n$ among its sibling nodes in the PXPE-tree. The p_dc field records the set of children of $c_n$ with the ‘descendant’ or ‘child’ axes, in which each child is identified by a $(\text{tagname}, \text{row_num})$ pair. The p_it a field records the set of children of $c_n$ with the ‘ancestor’ axis, in which each node is also identified by a $(\text{tagname}, \text{row_num})$ pair. A PXPE-node is a leaf if both of its p_a and p_dc are empty. (2) Field stack for filtering ‘descendant’ and ‘child’ axes. In this field we record a pointer to a run-time stack. In the algorithms, one run-time stack is created for each non-leaf query node. We utilize the technique for filtering ‘descendant’

![Fig. 6. Indexing Q1–Q6 using NIndex.](image)
and ‘child’ axes proposed by Gou et al. [16], and use the stack structure as follows: Each element \( e \) in stack records the depth of \( e \) in the data tree and a flag which is a bit array. The size of this array equals to the count of the child PXPE-nodes with the ‘descendant’ and ‘child’ axes of the current node \( c_n \) in the PXPE-tree. Given a row \( c_m \in p\_dc[c_n] \), e.flag[s[pos[cm]]] indicates whether a match with subtree \( (c_m) \) with only ‘descendant’ and ‘child’ axes has been found under \( e. \) (3) Field \( r\_queries \), records the IDs of those PXPE-trees that are rooted at the current node \( c_n \). If it is not empty, \( c_n \) is the root of some PXPE-trees. (4) Field \( s\_times \) records the number of times that the current row is shared. \( c_n \) is a common sub-pattern when its \( s\_times \) is greater than one.

In addition, for each table \( T_m \), we assign an \( a\_flag \), which records whether there is an open tag of the element with tag \( m \) during the process of filtering. It is used for filtering the ‘ancestor’ axis.

For example, in Fig. 6, a hash table is used to maintain mapping from tags to their corresponding tag tables. We take the default axis for the root of XPEs as ‘descendant’, and the parent of the root as 0. In NIndex, each row may refer to a stack of a row, and field \( a\_flag \) of a table. At the start of the filtering, \( a\_flag \) of each table is initialized to zero, and the stack of each row is set to null.

4.2. Updating of NIndex

4.2.1. Insertion

To insert a PXPE-tree into NIndex, we need to add its PXPE-nodes to the respective table by a post-order traversal of the PXPE-tree. The post-order traversal guarantees that every node is visited after its descendants are visited.

The algorithm of insertion is shown in Algorithm 1. We keep getting a PXPE-node \( c_n \) by the post-order traversal of the PXPE-tree until all nodes are traversed. For each \( c_n \), we search the table \( T_{cn\_tag} \) in NIndex to see whether \( c_n \) has a common sub-pattern with other rows of the table. Fields \( axis, p\_dc \) and \( p\_a \) are used to check whether a row with the same sub-pattern exists or not. If such a row is found, we do not insert a new row to the table to avoid redundancy. Instead, we only need to add \( s\_times \) by one. We further check if \( c_n \) is the root of PXPE-tree \( T \). If so, we also add \( ID \) of \( T \) to \( r\_queries \) to show that PXPE-tree \( T \) is rooted at node \( c_n \) of this shared row. If no row with the same sub-pattern is found, we add a new row to the table, fill in structural information of \( c_n \) except the \( parent \) field, and initialize \( s\_times \) to one. If \( c_n \) is the root, add the \( ID \) of \( T \) to \( r\_queries \); otherwise set \( r\_queries \) to empty. Afterwards we set \( c_n \) as the \( parent \) field value for all the rows in \( p\_dc \) and \( p\_a \) of \( c_n \). The algorithm for insertion guarantees that all the common sub-patterns are stored only once in NIndex, therefore NIndex has lower space complexity.

**Algorithm 1. Insertion**

**Input:** A new PXPE-tree \( T \), to be inserted into NIndex.

**Step 1:** If all PXPE-nodes are traversed, exit; otherwise get PXPE-node \( c_n \) by post-ordered traversal.

**Step 2:** In the table \( T_{cn\_tag} \), retrieve the row which has the same value in three fields, \( axis, p\_dc \) and \( p\_a \).

**Step 3:** If the row is found and \( c_n \) is not the root of PXPE-tree \( T \), add \( s\_times \) by one; goto step 1.

**Step 4:** If the row is found and \( c_n \) is the root of PXPE-tree \( T \), add \( ID \) of \( T \) to \( r\_queries \) in current row, and add \( s\_times \) by one; goto step 1.

**Step 5:** If the row is not found, add a new row to table \( T_{cn\_tag} \) with the structural information of PXPE-node \( c_n \), initialize the attribute \( s\_times \) to 1 and \( r\_queries \) to empty. If \( c_n \) is the root of PXPE-tree \( T \), add \( ID \) of \( T \) to \( r\_queries \).

**Step 6:** For each row in the lists of \( p\_dc \) and \( p\_a \) of \( c_n \), set \( c_n \) as the value of its parent field.

**Step 7:** goto step 1.

For example, when PXPE-tree \( Q_7 \) in Fig. 4 is inserted to NIndex shown in Fig. 6, PXPE-node \( C \) is visited first. We get the list \((axis, p\_dc \) and \( p\_a) \) of \( C \) from \( Q_7 \), then search for any row in \( T_C \) with the same value in this list. Row 1 in \( T_C \) is found, and then we add \( s\_times \) of row 1 by one. Then \( D, B \) and \( A \) are visited in order, the processes are similar to that of \( C \). When \( F \), which is the root of \( Q_7 \), is visited, we cannot find the respective row in \( T_F \). Therefore we add a new row to \( T_F \), and set its respective fields, for example, add the pair of the \( A \)'s tag name and the row \( ID \) in \( T_A \) to \( F \)'s \( p\_dc \), set its axis as the default ‘desc’. Because \( F \) is the root of \( Q_7 \), we need to add \( Q_7 \) to \( r\_queries \) of the new row.

4.2.2. Deletion

To delete an existing PXPE-tree from NIndex, we start to retrieve the root \( r \) of the PXPE-tree in NIndex, and traverse all the PXPE-nodes recorded in fields \( p\_dc \) and \( p\_a \) of \( r \). Firstly, we find the row for the root in NIndex, delete the PXPE-tree’s \( ID \) from its \( r\_queries \), and subtract \( s\_times \) by 1. If \( s\_times \) equals to zero, the current row is removed from the respective table, and the parent fields of all the children (in fields \( p\_dc \) and \( p\_a \) of the current row) are updated by removing the location of the deleted row. Then we recursively process for each row in \( p\_dc \) and \( p\_a \) of the removed row. This process will be completed until all
the nodes in the PXPE-tree are visited. The order of deleting from top to bottom can guarantee that all the nodes in the PEPX-tree can be updated in NIndex, by using the children’ location fields p_d and p_a.

For example, to delete Q3 shown in Fig. 4 from NIndex shown in Fig. 6, we search for the root B of Q3 in T_B and row 2 in T_B is found. As the value of s_times is 1, we can confirm that Q3 rooted at B as a sub-pattern is not shared by other PXPE-trees. Therefore we can delete row 2 in T_B. We then recursively process those rows in the p_d and p_a of row 2. No processing is needed for p_a as it is empty. For p_d, we need to process (C, 2). We subtract the value of s_times of row 2 of T_C by one. Now the value of s_times equals to zero, so we delete row 2 in T_C. As the p_d and p_a of row 2 of T_C are all empty, the deletion of Q3 ends.

5. Filtering algorithm based on NIndex

Based on NIndex, we propose a filtering algorithm that checks, for a given document D and a collection of PXPE-trees, which PXPE-tree matches D. As discussed above, each PXPE-tree T rooted at node r is stored as a row c_i in the table T_n (here n is the tag name of r) in NIndex. p_d and p_a of c_i record all child nodes of r with ‘descendant’/’child’ and ‘ancestor’ axes, respectively. To process ‘descendant’/’child’ axes, a stack will be created for c_i when the start tag of an element e with tag name n arrives. A bit array for p_d will also be created to check the arrival of the descendant/child elements of element e. To process ‘ancestor’ axis, a_flag is set for table T_a when the start tag of an element with tag name a arrives. When the end tag of an element with tag name n arrives, we can check whether D satisfies T as follows: (1) Check ‘descendant’/’child’ axes of c_i by examining the bit array for p_d to see if all bits of the array are set to 1; (2) Check the ‘ancestor’ axis of c_i by examining, for each tag name a in p_a, whether a_flag for the table T_a is set. If both hold, we can immediately send document D to those users who subscribe the request specified in T.

Algorithm 2. NIndex :: startElement (n)

1: \( depth = depth + 1; \)
2: \( T_n = \text{mapping}(n); \)
3: \( T_n.a\_flag++; \)
4: if \( T_n \neq FAIL \) then
5: for each \( c_i \) in \( T_n \) do
6: \( c_p = \text{parent}[c_i]; \)
7: if stack[\( c_p \)] is empty then break;
8: \( p = \text{top}(\text{stack}[c_p]); \)
9: if \( p.\text{flags}[\text{pos}[c_i]] == 0 \) then
10: \( \text{if} \ axis[c_i] == \text{'desc'} \text{ or} \ p.\text{depth}+1==\text{depth} \text{ then} \)
11: if \( \text{leaf}[c_i] == \text{'Y'} \) then
12: \( \text{p.\text{flags}[\text{pos}[c_i]]=1; \) }
13: else
14: \( s = \text{newElement}(\text{stack}[c_i],\text{depth}); \)
15: \( \text{push}(\text{stack}[c_i],s); \)
16: \( \text{Function mapping}(n) \)
17: if HashTable[\( n \)] \( \neq NULL \) then
18: return HashTable[\( n \)];
19: else
20: return FAIL
Algorithm 3. NIndex :: endElement (n)

1. \( T_n = \text{mapping}(n); \)
2. if \( T_n \neq \text{FAIL} \) then
3.   for each \( c_i \) in \( T_n \) do
4.     if \( \text{leaf}[c_i] == 'Y' \) or stack\([c_i]\) is empty then break;
5.     s = top(stack\([c_i]\));
6.     if \( s.\text{depth} == \text{depth} \) then
7.       pop(stack\([c_i]\));
8.       q.Flags = q.Flags | s.Flags;
9.     AFlag = 1;
10.    for each \( c_j \) in p_a[c_i] do
11.       if \( T_{c_j}.\text{tablename}.aflag == 0 \) then
12.         AFlag = 0;
13.       if evaluate(s.flags, f[c_i]) == true and AFlag == 1 then
14.         c_p = parent(c_i);
15.         if \( c_p == \emptyset \) then
16.           add r.queries(c_i) to result set \( S_{result} \);
17.         else
18.           p = top(stack\([c_p]\));
19.           p.flags[pos\([c_i]\)] = 1;
20.           if axis\([c_i]\] == 'desc' then
21.             clearPredStack(c_i);
22.         destroy(s);
23.     end if
24. end for
25. end if
26. end for
27. if stack\([c_n]\) is not empty then
28.   destroy all elements in stack\([c_n]\);
29.   for each \( c_i \) in p_d[c_n] do
30.     if \( \text{leaf}[c_i] == 'N' \) then clearPredStackS(c_i);
31. end for

As SAX is an event-driven model, our filtering algorithm can be specified as two algorithms to handle two core events startElement (Algorithm 2) and endElement (Algorithm 3), respectively. When a start tag or an end tag of an element arrives, the depth records the level of the element in the document. By the tag name and mapping table, the table \( T_n \) can be retrieved from NIndex (Line 2 in Algorithm 2 and Line 1 in Algorithm 3). All the rows in \( T_n \) need to be processed (Line 5 in Algorithm 2 and Line 3 in Algorithm 3). In what follows, we introduce the detailed processing for each row. We first explain the handling of the ‘descendant’/’child’ axes, then explain the handling of the ‘ancestor’ axis.

Given a row \( c_i \) of tag name \( n \) in table \( T_n \), the handling of the ‘descendant’/’child’ axes in startElement\((n)\) (as shown in Algorithm 2) is described as follows: when the start tag of an element with tag name \( n \) arrives, we obtain the parent \( c_p \) of \( c_i \) first. Nothing needs to be done if \( c_p \) has not arrived (Line 7) or \( c_i \) under \( c_p \) has a match, i.e. line 9 yields a false. Otherwise, if \( c_i \) is a
non-leaf node, create a stack element \(s\) for element \(c_i\) with \(s\text{.depth} = \text{depth}\) of \(e\) and initialize all bits in \(s\text{.flags}\) to 0 (Line 14), and push element \(s\) down to stack[\(c_i\)] (Line 15); if \(c_i\) is a leaf node, we can immediately set the flag bit for \(c_i\) in \(c_p\) (Line 12). The handling of the ‘descendant’/‘child’ axes in endElement(\(n\)) (as shown in Algorithm 3) includes evaluating the bit array \(s\text{.flags}\) (Line 13) to determine whether a match with subtree(\(c_i\)) has been found under \(p\) (Line 19), and passing all those bits in \(s\text{.flags}\) that have been set to 1 down to the new top element of stack[\(c_i\)] (Line 8).

Now we explain how to handle the ancestor axis. Given a PXPE-node \(c_i\) with tag name \(n\), for each of its child PXPE-node \(c_j\) with tag name \(m\) and an ‘anc’ edge in the PXPE-tree, \(a\text{.flag}\) of \(T_m\) must be non-zero (Lines 9–12 in Algorithm 3). This means to satisfy PXPE-node \(c_i\), an element of tag \(n\) must have all the ancestor elements that match their PXPE-node \(c_j\) in the PXPE-tree. The \(a\text{.flag}\) of \(T_m\) is increased by 1 when startElement(\(m\)) event happens (Line 3 in Algorithm 2), and decreased by 1 when endElement(\(m\)) event happens (Line 23 in Algorithm 3). When an element \(e_m\) of tag \(m\) comes, startElement(\(m\)) is executed and \(a\text{.flag}\) of table \(T_m\) becomes non-zero. When an element \(e_n\) arrives between startElement(\(m\)) and endElement(\(m\)), we can confirm that \(e_n\) has an ancestor element \(e_m\) in the document. So the relationship between PXPE-nodes \(c_i\) and \(c_j\) is matched.

If both the ‘descendant’/‘child’ axes and the ‘ancestor’ ancestor axis are satisfied (Line 13 in Algorithm 3), there is a matching of PXPE-node \(c_i\) in the current document. If \(c_i\) is the root of the PXPE-tree, we add the query IDs in \(r\text{.queries}\) to the result set (Line 16 in Algorithm 3).

For example, Fig. 7a is a PXPE-tree. When XML document in Fig. 7b comes, the first event is startElement(\(d_1\)). Because the axis of node \(D\) in \(T\) is ‘ancestor’, we increase the \(a\text{.flag}\) of table \(T_d\) by 1. So when \(a_1\) comes, we know that ancestor \(d_1\) exists from the \(a\text{.flag}\) of table \(T_d\). When \(b_1\) and \(c_1\) come, the respective bits of the bit array in the stack of the row for \(A\) are set to 1. When endElement(\(a_1\)) happens, all the ‘ancestor’ axis and the ‘descendant’/‘child’ axes are matched. Since \(A\) is the root of \(T\), \(T\text{'s ID}\) is added to the result set.

6. Performance evaluation

6.1. Experiment setup

6.1.1. XML documents

NITF (News Industry Text Format) DTD [11] and IBM’s XML Generator tool [13] were used to generate our XML document data set. NITF DTD contains 123 elements with 513 attributes. We generated a set of XML documents with the number of elements varying from 1000 to 8000 and the size varying from 9 K to 900 K.

6.1.2. XPath expressions

We implemented an XPath expression generator that takes a DTD as input and creates a set of valid XPath expressions based on the following five input parameters. The parameter \(P\) controls the size of the set of indexed XPEs (ranging from 10,000 to 50,000). The parameter \(L\) controls the depth of the PXPE-trees in terms of the maximum number of levels (ranging from 10 to 30). The parameter \(p_d\) controls the probability of having a ‘descendant’ axis at each node. The parameter \(p_a\) controls the probability of having an ‘ancestor’ axis at each node. The parameter \(p_w\) controls how wide the PXPE-trees of XPEs are, where the value 0 means to generate single path XPEs, and a higher value will increase the number of branches in the PXPE-trees.

6.1.3. Algorithms

We compared the performance of three algorithms: (1) YFilter, (2) XTrie, and (3) NIndex. All algorithms were implemented in C++ and compiled using VC 6.0. Experiments were conducted on a PC with 2.4 GHz Pentium processor and 512 MB main memory running Windows XP. All index structures were resident in main-memory for all the experiments. For each input XML document, we measured the total filtering time which includes the CPU time to parse the input document, to probe and update the index, and to report the matched expressions. We used the SAX parser from Apache [40] for parsing XML documents.
6.2. Performance study

Fig. 8 compares the scalability of the algorithms as a function of $P$, the size of the set of indexed XPEs. The results show that the filtering time increases almost linearly with $P$, with NIndex being the fastest algorithm, XTrie performing better than YFilter, and YFilter being the worst.

Fig. 9 compares the scalability of the algorithms as a function of the size of the XML documents (which is in terms of the number of elements or tag-pairs and varies from 1000 to 8000). The results clearly show that the filtering time increases linearly with the document size for all the algorithms, and the algorithm based on NIndex performs best.

Fig. 8. Varying $P(L = 20, p_d = 0.1, p_a = 0)$.

Fig. 9. Varying document size ($P = 10k, L = 20, p_d = 0.1, p_a = 0$).

Fig. 10. Varying $p_d$ ($P = 10k, L = 20, p_a = 0$).
Fig. 10 shows that increasing $p_d$, the probability of having a ‘descendant’ axis in the XPEs causes the increase of the filtering time of all algorithms. NIndex performs best and the filtering time increases almost linearly with $p_d$. For XTree, having more ‘descendant’ axes in an XPE is likely to result in a larger number of shorter substrings in its simple decomposition, which not only increases the number of entries in the substring-table but also leads to more matchings (due to shorter substrings). For YFilter, having more ‘descendant’ axes in the XPEs translates to more instances of partially matched expressions, thereby resulting in more processing overhead.

Fig. 11 compares the effect of the depth of XPEs on the performance of the filtering algorithms. The results show that the filtering time of all algorithms decreases slightly as the depth of the XPEs increases. This is because tree patterns with longer branches are more selective, thereby resulting in fewer matches.

In Fig. 12, we tested the effect of $p_a$. The results show that $p_a$ has no effect on our filtering algorithm.
In Fig. 13, we tested the effect of $p_b$. We vary $p_b$ from 0 to 0.5, and the filtering time slightly increases. The results show that the number of predicates with all kinds of axes has no effect on the filtering algorithm based on NIndex.

7. Related works and comparisons

For the XPath query evaluation problem, most existing works focus on structural joins and indexing. Al-Khalifa proposed algorithms Tree-Merge join and Stack-Tree join in [2], and Bruno proposed algorithm Path-Stack [7]. Several indexing schemes were proposed for XPath query processing including LORE [31], dataguides [14], ToXin [35], XISS [27], index fabric [10], and ViST [38]. The core problem of these works is to efficiently match the descendant axis in XML documents. The ‘ancestor’ and ‘parent’ axes were not considered. Xaos [4] proposed to support ‘ancestor’ and ‘parent’ axes by using X-dag. In Xtwig [37], a holistic algorithm was presented to efficiently process complex XPath query with the ancestor axis. However all of the above works were not designed for processing multiple XPath expressions at a time.

There were also many works on efficiently filtering XML documents with a collection of XPath expressions. The XML data is usually modeled as data stream, and many automata-based algorithms were proposed. The XFilter system [3] handles simple XPath location path expressions by transforming them into a Deterministic Finite Automaton. The YFilter system [12] is an extension of XFilter in which a group of simple XPath location path expressions are combined into a single Non-deterministic Finite Automaton (NFA), which corresponds to the union of these path expressions. XTrie [8] can handle twig pattern expressions containing predicates which are internally represented as XTrie-trees. XPush [18] supports twig pattern with nested predicates and shares the common predicates among multiple XPath expressions by using an Alternating Finite Automaton (AFA). The lazy filtering algorithm (LF) [16] proposed by Gou is not an automata-based algorithm, and it uses SAX method to parse XML documents. However all these algorithms can only handle ‘child’ and ‘descendant’ axes.

For the research work on the XPath rewriting, Olteanu et al. [34] proposed several rewriting rules to transform absolute XPath location paths with reverse axes into equivalent reverse axis free ones. However the transformation is based on the syntax of the XPath language, and it cannot be easily applied to query processing and document filtering, because the transformed query patterns contain equality joins of node identity inevitably, and some of them need to scan the documents multiple times.

8. Conclusions

In this paper, we addressed the problem of filtering XML documents with large number of user submitted queries in XPath expressions. We proposed a new index structure called NIndex. It offers several features that make it especially attractive for large scale SDI systems. NIndex is designed to support effective filtering based on complex XPath expressions containing the reverse axes ‘ancestor’ and ‘parent’. It reduces the number of unnecessary index probes and avoids the redundant matchings. Updating algorithms of NIndex for Insertion and Deletion were introduced. Based on NIndex, a new filtering algorithm with low computational complexity was also presented with extensive experimental studies showing that our algorithm performs well across a range of XPath expressions and documents.

Acknowledgements

This research was supported by the Australian Research Council Discovery Projects (Grant Nos. DP0878405 and DP110102407), the National Natural Science Foundation of China (Grant Nos. 60972090 and 61073057), the Fundamental Research Funds for the Central Universities of China (Grant No. 2011ZD010).

References