Efficient Evaluation of
Forward XPath Axes over XML Streams

Abdul Nizar M., P. Sreenivasa Kumar

Indian Institute of Technology Madras
Chennai - 600 036
INDIA
{nizar,psk}@cse.iitm.ernet.in

Abstract

Although many algorithms have been proposed for evaluating XPath queries containing un-ordered axes (*child*, *descendant*, *parent* and *ancestor*) against streaming XML data, there are very few efforts towards developing algorithms for processing path expressions with ordered axes (*following*, *following-sibling*, *preceding* and *preceding-sibling*). In this paper, we show how order information can be built into the conventional twig-structure, in order to represent path expressions with *following* and *following-sibling* axes in addition to *child* and *descendant* axes. We then discuss an efficient way of encoding and matching XPath queries with *forward* (*child*, *descendant*, *following*, *following-sibling*) axes against streaming XML data. The algorithm processes branches of the twig in left-to-right order. A branch is never processed unless constraints specified in the preceding branches are satisfied by the stream. Also, the algorithm avoids repeated processing of branches whose constraints have already been satisfied by the stream. Experiments over real-world, synthetic and benchmark data sets show that our system outperforms the currently available algorithm by wide margins.

1   Introduction

Due to the wide-spread use of XML data, especially in the web context, content- and structure-based filtering and information extraction from streamed XML documents attracted interest of the research community. Systems that operate on XML streams fall into two categories – filtering systems and querying systems. In these systems, user interest is expressed in the form of XPath[3] or XQuery[2] queries and the incoming document is processed against these queries. An XML filtering system, identifies if the query has a match

in the document and routes the document to the user. An XML querying system, on the other hand, retrieves *all* the document fragments in the stream satisfying the query.

XPath expressions consisting of *child* ('/') and *descendant* ('//') axes are conventionally represented using tree structures known as twig queries. Twig representation is quite suitable for XPath processing as the process is closely tied to the tree representation scheme adopted for XML data. Naturally enough, a major share of the XML stream query processing algorithms use twig structure to encode the query expression. Performance of *holistic* twig-based algorithms has been shown to be much better than that of systems based on other approaches and formalisms, particularly when the data is recursive.

XPath has four ordered axes – *following*, *following-sibling*, *preceding*, *preceding-sibling* – and there are queries that can be effectively expressed using these axes. For instance, in the context of the XML document representing a journal paper, the XPath expression *//section[name= "Motivation"]/following::figure* returns all *figure* elements that appear in the paper *after* the section titled *Motivation*. Here the result does *not* include *figure* elements *within Motivation* section.

As the conventional twig structure does not carry any order information, it can not be used to represent and process XPath expressions with ordered axes. In this paper we show how conventional twig structure can be extended by adding additional constraints to represent XPath expressions with ordered axes. We then present a stream querying algorithm for XPath expressions with forward – *child*, *descendant*, *following* and *following-sibling* – axes. Our contributions are summarized below:

1. We show how ordering constraints can be incorporated into conventional twig structure so that it can effectively represent XPath expressions with *following* and *following-sibling* axes.

2. We present an algorithm to effectively process XPath expressions with forward axes against XML streams and establish its correctness.

3. We experimentally show that the proposed algorithm outperforms existing state-of-the art algorithm for ordered axes by wide margins.

The rest of the paper is organized as follows: Section 2 presents the background and motivates the work. Related work is discussed in Section 3. Section 4 formally shows how ordering constraints can be added to twigs so that they can effectively represent XPath expressions with ordered axes. In Section 5, we discuss our query processing algorithm for path expressions with forward axes. Section 6 presents the experimental results and Section 7 concludes the paper.

2 Background and Motivation

An XML document can be modelled as an ordered node-labelled tree. Order in which opening tags of elements appear in the document is called the *document order* of elements. Document order is the same as pre-order of nodes in the document tree. Edges in the tree represent element containment relationships. (See Figure 1(a). Pre-order number of a node is shown near the node and level number is shown on the right side.)

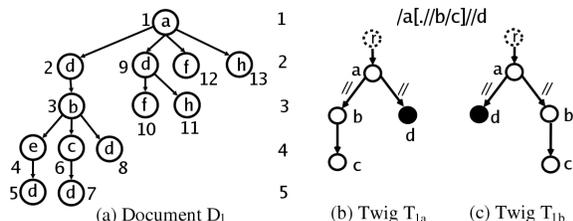Conventionally, XPath queries with *child*('/') and *de-*

Figure 1: XML Document Tree and Twigs

scendant('//') axes ($XPATH^{/,//}$) are effectively modelled as twig queries. A twig is a tree structure where nodes are labelled with node tests and directed edges are used to represent axes. Figure 1(b) shows the twig T_{1a} representing the path expression /a[.//b/c]//d. In the figure, edges without labels (P-C edges) represent child axes while edges with '//' as label (A-D edges) represent descendant axes. There is a root node labelled r. We call r the root of the twig. This node is used to distinguish between twigs representing absolute path expressions (those starting with '/') and relative path expressions (those starting with '//'). Node d shown in black is the result node. Note that the Twig T_{1b} in Figure 1(c) is isomorphic to T_{1a} and is an alternative twig representation of the *same* path expression.

Majority of algorithms for processing $XPATH^{/,//}$

queries against streaming data encode the path expression as twig and systematically find matches ([5, 10, 13]). The efficiency of the twig-based algorithms is attributed to the fact that the completely nested structure of XML document can be effectively combined with the LIFO nature of stacks to find the matches. In XML stream processing systems, the *open-tag* and *close-tag* events in the input stream trigger updation of stacks associated with nodes of the twig. This approach is particularly suitable for recursive data where number of matches for a query can be exponential in the size of the query.

An XPath expression containing ordered axes also, looks for matches containing nodes with certain constraints (in addition to those imposed by '/' and '//'). For example, the path expression /a//b/c/following::d looks for matches in which d-node is appearing after c-node in *document order* but is *not* a descendant of the latter. Conventional twig structure fails to convey such order information as it does not impose any order on the nodes selected. For instance, we can not represent the above XPath query as twig T_{1a} of Figure 1(a) as it looks for d-nodes in the document irrespective of their position with respect to b-node or c-node. All the d-nodes in document D_1 (Figure 1(a)) are produced as answers where as for /a//b/c/following::d, only $\langle d, 8\rangle$ and $\langle d, 9\rangle$ are answers. Thus algorithms for $XPATH^{/,//}$-queries based on conventional twig structure *can not* be directly extended to handle queries with ordered axes.

Twig-based streaming algorithms (*e.g.*:- [10, 13]) have been shown to out-perform systems based on other approaches and formalisms (*e.g.*:-[12, 7]), particularly when the data is recursive. However, the research efforts towards processing query expressions with ordered axes against streaming data, to the best of our knowledge, is limited to the SPEX system proposed in [7, 16]. The system compiles the XPath query into a network of push down transducers. The transducer system responds to the events from the input XML stream to find matches for the path expression. It does not use the twig-based representation of XPath expressions.

Thus, it would be interesting to see how ordering constraints can be incorporated into conventional twig structure to enable them to effectively represent path expressions with ordered axes and to study the behaviour of stream processing algorithms based on such extensions.

3 Related Work

XML stream filtering and querying is an active area of research and many stream filtering systems ([11],[9],[14],[15]) and querying systems ([8],[10],[5],[13]) have been proposed. Most of the filtering systems use automata- sequence- or substring-based approaches and achieve speed and scalability

by exploiting prefix and suffix sharability among the queries to be processed.

YFilter[11], which was originally a filtering system, was later enhanced to perform querying of multiple XPath expressions in a shared fashion. It constructs a single non-deterministic finite automaton (NFA) for all the path expressions. Cadan *et.al.* proposed AFIlter[8] system for XPath expressions without predicates. The system uses an *axis-view* data structure which is a directed graph compactly representing common inter-node relationships in all the path expressions. The graph is used to guide shared evaluation of path expressions. The system also proposes optimizations based on prefix sharing and suffix clustering of path queries to liverage scalability.

The $\chi\alpha o\varsigma$ a system proposed by Josifovsky *et.al.*[6] performs stream processing of XPATH expressions with *child, descendant, parent* and *ancestor* axes. The *TurboXPath* system[19] handles stream processing of XQuery-like queries. The authors claim that the system can be extended to handle queries with *following* and *following-sibling* axes, but it is not clear how the extension can be done. Also, they do not report the performance results for queries with these axes. The XSQ system proposed in [12] handles complex XPath Queries with *child, descendant* and closure axes with predicates and can handle aggregations. The system uses a hierarchy of push-down transducers with associated buffers.

The TwigM algorithm proposed in [10] avoids proliferation of exponential number of twig matches in recursive XML streams by using compact encoding of potential mataches using a stack structure. The system proposed by Aneesh *et.al.*[5] performs shared processing of twigs in document order by breaking twigs into branch sequences, which are sequences around branch points. The system due to Gou and Chirkova[13], which processes conventional twigs, achieves better time and space performance than TwigM when processing recursive XML documents. It can also handle predicates with boolean connectives. The authors propose two variants of the algorithm – Lazy Querying (LQ) and Eager Querying (EQ) – of comparable performance.

The SPEX[7] system processes XPath expressions with forward axes by mapping it to a network of transducers. Query re-writing methods [17] are used to transform expressions with backward axes to ones containing only forward axes. Most transducers used are single-state pushdown automata with output tape. For path expressions without predicates, the transducer network is a linear path; otherwise, it is a directed acyclic graph. Each transducer in the network processes, in stepwise fashion, the XML stream it receives and transmits it unchanged or annotated with conditions to its successor transducers. The transducer for to the result node holds potential answers,

to be output when conditions specified by the query are found to be true by the corresponding transducers. Due to the absence of built-in order information, the system processes and caches large number of stream elements which will be found useless later.

4   Representing Ordered Axes

In this section, we discuss how conventional twigs can be extended to represent XPath expressions with ordered axes. We call the resulting structure *Order-aware Twigs(OaTs)*. Due to space limitations, we keep the details to the minimum needed to understand the query processing algorithm. A formal, detailed discussion of OaTs, including the algorithm for transforming XPath expressions to OaTs and the proof of equivalence of XPath expressions and OaTs, can be found in [4].

Note that the match of a twig against an XML document is conventionally defined as a mapping from the nodes in the twig to the nodes in the document satisfying twig-node labels and relationships between twig-nodes.

We add two types of ordering constraints – *LR-Ordering* and *SLR-Ordering* – to conventional twigs to form OaTs. LR-ordering is specified from a node x to node y such that x and y appear in two *disjoint* downward paths from the twig root r. It has the interpretation that in a match of the twig against some document D, the nodes – say p and q – matching x and y should be such that q appears after p in document order in D, but is *not* a descendant of p. x and y are called *tail* and *head* of of LR-ordering, respectively. LR-Ordering can effectively represent *following* axis appearing in the path expressions[1]. For instance, the XPath query /a//b/c/fl::d[2] looks for d-nodes which are appearing after the close-tag of a c-node child of a b-node descendant of the document root node a. To represent the axis fl::d, a new node labelled d is connected to the root of the twig through an A-D edge and an LR-edge (dashed edge) is added from node c to d (see Figure 2(a)). Clearly, semantics of T_2 is the same as that of the query. Note that the OaT T_3 in Figure 2(b) is different from T_2 and represents the expression /a//b[c]/fl::d.

An SLR-Ordering is specified from a node x to node y such that x and y connected to its parent via P-C edges. It has the interpretation that in a match of the twig against some document D, the nodes – say p and q – matching x and y should be such that q appears after p in document order in D. SLR-Ordering can effectively represent *following-sibling* axis. The OaT T_4 in Figure 2(c) represents the XPath expres-

---

[1]LR- and SLR-orderings can represent backward ordered axes also. See [4].

[2]For brevity, we use fl, fs, pr and ps as abbreviations for following, following-sibling, preceding and preceding-sibling, respectively.

sion //b/c[fs::d]. In the figure, SLR-Ordering is shown using a solid arrow from c to d.
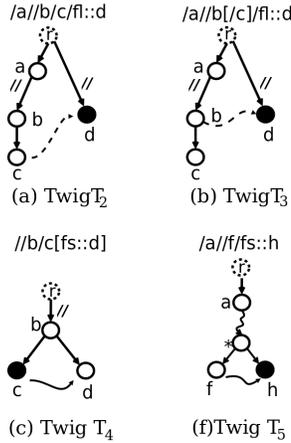
If there is an LR-Ordering (*resp.*, SLR-Ordering) from

(a) TwigT$_2$ (b) TwigT$_3$

(c) Twig T$_4$ (f)Twig T$_5$

Figure 2: Examples of OaTs

node n_1 to node n_2 in an OaT, n_1 is called the *tail* of the LR-Ordering (*resp.*, SLR-Ordering) and n_2 is called the *head*.

Closure Edges: The basic twigs need to be further extended to handle XPath expressions containing an axis step with *following-sibling* or *preceding-sibling* axis that appears (inside predicate or otherwise) *immediately* after an axis step with *descendant* axis. For example, in the query /a//f/fs::h, the axis step fs::h appears immediately after //f. Here h can be a right-sibling of an f-child of a or right-sibling of f-child of descendant of a. We handle this situation by introducing a new type of edge known as the *closure edge*. A closure edge from node n_1 to a node n_2 with a wild card label ('*') indicates that in a match of the OaT, the document node d_2 matching with n_2 can be the same as the document node d_1 matching with n_1 or a descendant of d_1. Figure 2(d) shows how a closure-edge can be used represent the path expression /a//f/fs::h. Here the *zig-zag* edge between the a-node and *-node is a closure-edge.

Figure 3 illustrates how an XPath Expression with ordered axes can be systematically transformed into an OaT. Interested readers are referred to [4] for a detailed account on how to generate such OaTs from XPath expressions.

To summarize, an Order-aware Twig (OaT) is a tree structure rooted at a node labelled 'r' known as the *root* of the OaT. There are three types of relationship edges – P-C edge, A-D edge and closure edge – and two types of constraint edges – LR edge and SLR edge. The match of an OaT against an XML document is a mapping from nodes in the OaT to nodes in the document satisfying the node labels and relationships and constraints between the nodes of the OaT.
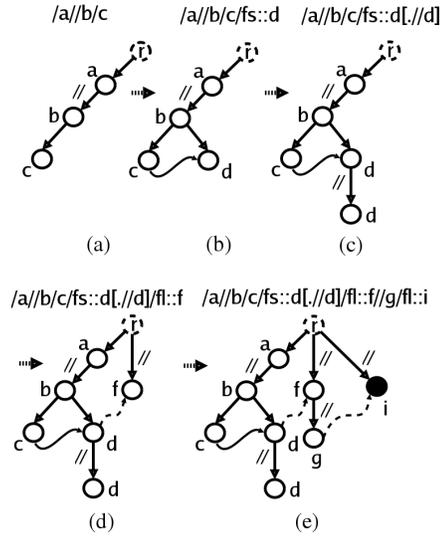

(a)   (b)   (c)

(d)   (e)

Figure 3: Transformation of Path Expression to OaT

5   Solution for Forward Axes

In this section, we discuss in detail the algorithm for matching OaTs representing path expressions with *child, descendant, following* and *following-sibling* axes against streaming XML data. The solution also supports predicates containing *child* and *descendant* axes that appear immediately *before* a *following* or *following-sibling* axis or at the end of the path expression. Note that queries with predicates that have arbitrary combination of *child, descendant following* and, *following-sibling* axes are not handled by the current algorithm.

From the illustration in Figure 3, it is clear that, for every axis step with *following* axis, a new node with LR -ordering is added to the OaT (see, for instance, Figure 3(d)&(e)). A similar observation can be made in the case of *following-sibling* axis. Thus, when a path expression belonging to the subset mentioned above is transformed to an OaT, the subtrees representing predicates will appear under the tail of SLR or LR-Ordering or under the result node (see Figure 3(e) representing the final OaT for the expression /a//b /c/fs::d[.//d]/fl::f//g/fl::i). Also, assuming that SLR and LR-Orderings are from left to right, the result node appears in the *right-most* branch of the OaT's root.

5.1   Encoding the OaT

We call the path in the OaT from the root node r to the *parent* of the result node, the *stem* of the OaT. A node in the twig can be one of *three* types - *boolean* node, *result* node or *stem* node. All nodes except the result node and the nodes along the stem (stem nodes) are boolean nodes. Boolean nodes represent existential semantics of XPath. For instance, the expression

	1	2	3	4	5	6	7	8	9
label	r	a	b	c	d	d	f	g	i
fanOut	3	1	2	0	1	0	1	0	0
nodeType	s	b	b	b	b	b	b	b	r
relShip	-	c	d	c	c	d	d	d	d
parent	-	1	2	3	3	5	1	7	1
chPosn	-	1	1	1	2	1	2	1	3
LNode	-	-	-	-	-	-	5	-	8
SLNode	-	-	-	-	4	-	-	-	-
root	y	n	n	n	n	n	n	n	n
LRTail	n	n	n	n	y	n	n	y	n
LRHead	n	n	n	n	n	n	y	n	y
TRPosn	-	-	-	-	1	-	2	2	3
children	[2,7,9]	[3]	[4,5]	[-]	[6]	[-]	[8]	[-]	[-]
dchildren	[7,9]	[3]	[-]	[-]	[6]	[-]	[8]	[-]	[-]

Figure 4: Node Table for OaT in Figure 3(e)

/a//b/c/fl::d looks for d nodes that 'follow' *at least* one c node that is the child of a b-descendant of an a-node which is the document's root. Hence, in the OaT for this expression shown in Figure 2(a), the nodes a, b and c are boolean nodes. That is, even if there are multiple branches in the document satisfying the label and structural constraints of nodes a, b and c, every branch other than the first one can be ignored as d is the only result node.

The OaT is encoded as a *node table* to facilitate query processing. The node table has one entry for each node appearing in the query. The node table entry for a node N holds the following information:- (i) *label*: Label of node N. (ii)*fanOut*: Number of children of N. (iii)*nodeType*: Holds value 'b' (*resp.*, 'r', 's') if N is a boolean (*resp.*, result, stem) node. (iv) *relShip*: Holds value 'c' (*resp.*, 'd', 'l') if N is a child (*resp.*, descendant, closure) node of its parent. (v) *parent*: Index, in node table, of the parent node of N. (vi) *chPosn*: position of N as a child of its parent. (vii) *LNode* (*resp. SLNode*): Represents index, in the node table, of the node which is the tail of LR-Ordering (resp., SLR-Ordering) for which N is the head. The *LNode* and *SLNode* fields are used to ensure LR and SLR constraints during OaT matching. (viii) *root*: Tells if N is root of the OaT ('Y') or not ('N'). (ix) *LRTail* (*resp. LRHead*): Tells if N is tail (*resp.* head) of LR-Ordering ('Y') or not ('N'). (x) *TRPosn*: Applicable to the nodes which are head or tail of LR-Ordering and represents position, in the OaT's root, where the path containing node N begins. (xi) *children*: child nodes of N. (xii) *dChildren*: child nodes that are connected to N via A-D edges. It is a subset of *children*.

The node table can be generated in a single pass over the OaT. Figure 4 shows the node table for the OaT in Figure 3(e). It is assumed that the global variable *TR* represents index of the node table entry corresponding to the root of the OaT. We use N_i to denote the ith node table entry and also the OaT node represented by that entry.

5.2 Matching Algorithm

In this section, we outline the query processing algorithm. Initially we assume that there are no closure nodes in the OaT. Later, in Section 5.3, we show how the algorithm can be extended for closure nodes. The algorithm maintains a stack at every query node. Each frame in the stack represents an element in the stream that matches, partially or completely, with the query node to which stack is associated. Let Q be a node in the OaT and P be its parent. A stack frame F belonging to the stack of Q has the structure, $\langle levelNo, sPreNo, parIndex, parFrameNo, posn, PNVect \rangle$ where (i) *levelNo*: depth, in the document tree of the node represented by F. (ii) *sPreNo*: Pre-order number of the node. (iii) *parIndex*: Index of P in the node table. (iv) *parFrameNo*: position of the frame in P's stack that matches with F based on the relationship constraint (P-C or A-D) specified between P and Q. We call this frame the parent frame of F. (v) *PNVect*: A vector of pre-order numbers[3], whose size is the same as the fan-out of Q. Each position in this vector is reserved for holding the pre-order number of the element that matches with a child node C of Q. In other words, the position is reserved for a frame in C's stack for which F is the parent frame. (vi) *posn*: position reserved for F in the pre-order vector of its parent frame.

The algorithm starts by pushing a dummy frame $\langle$ 0, 0, -1, -1, -1,$[-,\ldots,-]\rangle$ into the stack for the root node r. It then proceeds by responding to events generated by a SAX parser. Two global variables *gDepth* (initialized to 1) and *gPreNo* (initialized to 0) are maintained by the algorithm to keep track of level number and and pre-order number of elements in the XML stream, respectively, during parsing. Note that the algorithm has to respond to *open-tag* and *close-tag* events of elements whose tag names appear as labels of the nodes in the twig query only.

We assume that the following functions are available. (i) *indexSet(t)*: Returns indexes of node table entries j such that $N_j.label = t$ or $N_j.label =$'*'. We assume that, for an open-tag, the indexes of node table entries are returned in the same order as the appearance of the corresponding nodes in the OaT in post order. Similarly, for close tag, the indexes of node table entries are returned in pre-order. (We elaborate on the reasons for this requirement in Sections 5.2.1 & 5.2.2.) This function can be efficiently implemented with two hash tables which are constructed during query encoding. (ii) *parent(N_j)*: Node Table entry for parent of N_j. (iii) *isLRCompatible(N_j)*: Returns *true* if for $k = N_j.LNode$, the position corresponding to N_k in *PNVect* of the (only) frame in the stack of the OaTs root node is filled; otherwise, returns

---

[3]Though a bit vector is sufficient for the XPath subset we consider in this paper, the pre-order number vector is used in view of its usefulness in future extensions of the algorithm.

false. (iv) *isSLRCompatible*(N_j): Returns *true* if for $k = N_j.SLNode$, the position corresponding to N_k in *PNVect* of the top frame in the stack of the parent node of N_k is filled; otherwise, returns *false*. (v) *evaluate*(F): Returns *true* if all the fields of the pre-order number vector of frame F have been filled; otherwise returns *false*. (vi) *makeFrame*(...): Returns a new stack frame to be pushed to the stack of a query node. If the query node is a leaf node (*fanOut* = 0) the frame does not contain *PNVect*, otherwise the there will be a *PNVect* of size equal to *fan-out*. (vii) *updateDescPosns*(N_j,F) where F is a frame just popped out from N_j.Stack: Updates the empty descendant positions in the *PNVect* of the top frame of N_j.Stack (that is, child positions of $N_j.dChildren$ in N_j) with the corresponding postions in the *PNVect* of F. (viii) *reclaim*(F): Reclaims the memory allotted to the stack frame F. (ix) Standard stack functions *pop*(N_j), *push*(N_j, F) and *top*(N_j) which operate on the stack associated with node N_j (x) *isEmpty*(N_j.Stack): Checks if the stack associated with N_j is empty.

5.2.1 The Open-Tag Handler

Algorithm 1 shows the steps in *open-tag* event processing. The steps between dotted lines are for closure node processing and may be ignored for now. For the open tag $<t>$ of an element e with depth $d = gDepth$ and pre-order number $sPreNo = gPreNo$ appearing in the input stream, the *open-tag* handler proceeds by identifying all the node table entries N_j for which $N_j.label = t$ or $N_j.label = $ '*'. For each entry, the handler checks whether a frame representing the element e should be pushed to the corresponding stack, a decision based on values of various fields such as *node-Type*, *relShip*, *LRHead*, *fanOut* etc.

The intuitive idea behind open tag handling is as follows: Let N_i be the parent of node N_j in the query. A frame corresponding to e is pushed to $N_j.stack$ when (i) the element represented by the top frame F in $N_i.stack$ and e satisfy the relationship between nodes N_i and N_j and (ii) e satisfies the LR- and SLR-Orderings (if any) specified on N_j. Condition (i) ensures that when a frame is pushed into the stack of N_j, there is a chain of frames in the stacks of the nodes in the path from r to N_j, satisfying the A-D and P-C relationships specified along the path. Condition (ii) ensures that such a chain of frames have already appeared in the stacks along all the paths from r to leaf nodes (*i.e.*, nodes with fan out zero) of the OaT that appear *before* the path from r to N_j. This avoids unnecessary pushing of a lot of frames and lightens the 'work load' of *close-tag* event handler, leading to algorithm speed up. Further, if N_j is a boolean node and the position corresponding to N_j in the pre-order vector of F in the stack of N_i is not empty, no frame for e need to be pushed to the stack of N_j.

Figure 5 illustrates some of the steps in the eval-

Algorithm 1: *Open-tag* Event Handler

Input : t:Opening tag of an element in the stream
Global: $gDepth$: Depth of the element in the document, $gPreNo$: Pre-order number of the element, TR: Index, in the node table, of the OaT's root

1 **Open-Tag Handler**(t)
2 $d \leftarrow gDepth$; $gDepth \leftarrow gDepth + 1$
3 $gPreNo \leftarrow gPreNo + 1$; $sPreNo \leftarrow gPreNo$
4 **foreach** $j \in$ **indexSet**(t) **do**
5   **if** N_j *is head of LR-Ordering* **then**
6   **if** *(isEmpty(N_{TR}.Stack) $\lor$ not isLRCompatible(N_j)* **then**
7   Continue with the next iteration

8    $N_i \leftarrow$ **parent**(N_j)
9    $c \leftarrow N_j.chPosn$
10   **if** *isEmpty(N_i.Stack)* **then**
11   Continue with the next iteration;

12   **if** *(N_j is a child node) $\land$ ($d - N_i$.Stack[top(N_i)].levelNo > 1)* **then**
13   Continue with the next iteration;

14   **if** *(N_j is a boolean node) $\land$ (N_i.Stack[top(N_i)].PNVect[c] is not empty)* **then**
15   Continue with the next iteration;

16   **if** *not isSLRCompatible(N_j)* **then**
17   Continue with the next iteration;

18   **if** N_j *is a result node* $\land N_j.fanOut = 0$ **then**
19   Report Result;
20   Continue with the next iteration;

21    $F \leftarrow$ **makeFrame**(d, $sPreNo$, $N_j.parent$, **top**(N_i), c, $N_j.fanOut$)
22   **push**(N_j, F)
   ----
23   **foreach** $k \in N_j.children$ **do**
24   **if** N_k *is a closure node* **then**
25        $F_s \leftarrow$ **makeFrame**(d, $sPreNo$, j, **top**(N_j), $N_k.chPosn$,$N_k.fanOut$)
26   **push**(N_k, F_s)

---

uation of the OaT of Figure 3(e) (repeated as Figure 5(a); the two d-nodes are renamed as d_1 and d_2 for easy reference). In Figures 5(c) to 5(f), stack for a query node with label l is represented as S_l. For brevity, we have shown, in the stack frames, *levelNo*, *sPreNo* and *PNVect* fields only. Values of the remaining fields are clear from pointers between the frames.

Figure 5(c) shows contents of the stacks associated with the nodes in the left-most path of the OaT in Figure 5(a), immediately after the *open-tag* event corresponding to node $\langle c, 4 \rangle$ in document D_1 (Figure 5(b)) has occurred. Note that, the frames in the stacks linked by pointers form a path in the document satisfying the relationships specified between nodes along

the left-most path in the OaT. Also note that, lines 5–7 of the *open-tag* handler prevent the frame for $\langle i, 2\rangle$ from being pushed to S_i (not shown in the figures). This is because the function *isLRCompatible*() returns *false* as position 2 in the pre-order number vector of the (only frame) of S_r is not yet filled, which indicates that the LR-Ordering constraint in the OaT from node g to node i is not satisfied.

The *open-tag* handler will not push any frame to stack S_c when the opening tag event corresponding to node $\langle c, 5\rangle$ as the position in pre-order vector of the frame in S_b has already been filled by the *close-tag* event corresponding node $\langle c, 4\rangle$ (see the next section). Note that, a frame representing $\langle d, 6\rangle$ is pushed to S_{d_1} (Figure 5(d)) for, the top frame in S_b and $\langle d, 6\rangle$ satisfy the P-C relationship between nodes b and d and the position in the pre-order number vector, corresponding to node c, of the frame is already filled, which indicates that the SLR-ordering from node c to node d is satisfied.

As mentioned earlier, it is assumed that the *open-tag* handler processes the node entries returned by *indexSet()* in post-order of the query tree. This restriction is needed to avoid erroneous computation when same-label and wild card nodes appear in the OaT. For example, when open tag-event for node $\langle d, 6\rangle$ occurs, there are node table entries to be considered – entries at indexes 5 and 6. If these entries are processed in that order, line 22 of Algorithm 1 will push a frame to the stack (S_{d_2} – not shown in the figures) associated with node table entry at index 6, which is an error. The correct order of processing is 6 followed by 5.
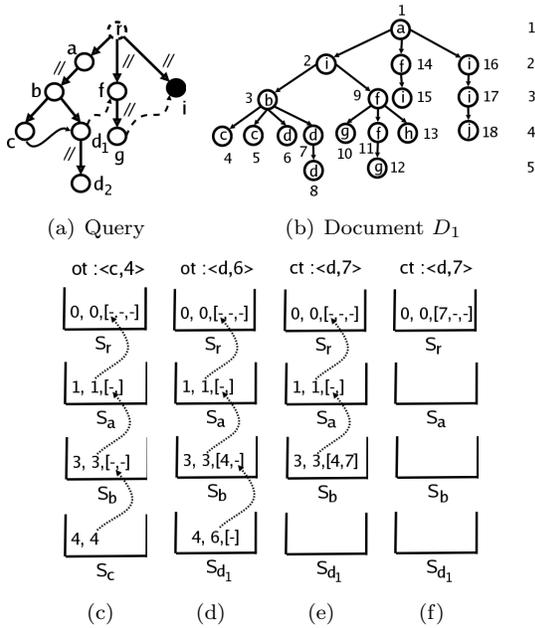

(a) Query   (b) Document D_1

(c)   (d)   (e)   (f)

Figure 5: Snap shots of Query Evaluation

5.2.2 The Close-Tag Handler

Algorithm 2 handles *close-tag* events. Again, the block between dotted lines is for closure node processing. For each node table index j returned by the *indexSet()* function in response to the close tag $</t>$ of an element e with depth $d = gDepth$, the *close-tag* handler checks if a frame representing e is present as topmost element in $N_j.stack$ and, if so, pops out the frame (lines 4–6). If *all* the postions in the pre-order number vector of the frame are filled, the *sPreNo* field of the frame is used to update the associated position in the pre-order number vector of the parent frame (lines 20–23). For instance, Figure 5(e) shows contents of relevant stacks after the processing the *close-tag* event for $\langle d, 7\rangle$. Note that, the second position of the pre-order vector of the top frame in S_b is filled with value 7.

If N_j is a tail of LR-Ordering, we know that a sequence of frames representing the elements in the stream, which satisfies the relationships specified along the path from N_j to the OaT's root, have been identified. Thus the pre-order vector position for N_j in the (only) frame of the twig root is updated. All the frames in the stacks associated with nodes in the path from N_j to the twig root can now be safely flushed out (lines 26–28). Figure 5(f) is the snapshot at the end of processing the *close-tag* event for $\langle d, 7\rangle$.

The advantage of clearing the stacks is that the close-tag processing for nodes corresponding to the frames in these stacks is avoided. Also note that, as the nodes are boolean and the position in the pre-order vector of the (only) frame of stack of the OaT's root is already filled, no new frames will be pushed to any of these stacks. This avoids execution of the *open-tag* handler beyond line 4 and that of the *close-tag* handler *completely* for nodes along that branch of the root. Clearly, it can lead to considerable performance gain.

5.2.3 Returning Results in Document Order

Note that, for the subset of XPath expressions we consider, the result node appears in the right-most branch of the OaT's root node. Also, if the XPath expression does not end with a predicate, the result node of the OaT is a leaf node; otherwise, the result node has a sub-tree rooted at it representing the predicate. If the result node is a leaf node, then validity of a document element as a result can be decided at the opening tag of the element (lines 18-19 of the *open-tag* handler) and hence the results are reported in document order. If the result node has a sub-tree under, validity of a document element as result can be determined at the close tag after the frame corresponding to the element present in the stack of the result node is evaluated (lines 20–25 of *close-tag* handler). In this case, we need additional processing to ensure that the results are reported in document order.

Algorithm 2: *Close-tag* Event Handler

Input : t:Closing tag of an element in the stream

Global: $gDepth$: Depth of the element in the document, TR: Index, in the node table, of the OaT's root

1 ***Close-tag* Handler**(t)

2 $gDepth \leftarrow gDepth - 1$

3 foreach $j \in \mathbf{indexSet}(t)$ **do**

4   **if** *(isEmpty(N_j.Stack)* $\vee$ N_j.Stack[**top**(N_j)].levelNo $\neq$ gDepth) **then**

5   Continue with the next iteration;

6 $F_c \leftarrow pop(N_j)$

     ----

7   **foreach** $k \in N_j.children$ **do**

8   **if** N_k *is a closure node* **then**

9 $F_p \leftarrow pop(N_k)$

10   **if** *not isEmpty(N_k.Stack)* $\wedge$ $N_k.stack$[**top**(N_k)].levelNo=F_p.levelNo **then**

11   *Copy $F_p.PNVect$ to* $N_k.stack$[**top**(N_k)].PNVect

12   **if** *evaluate(F_p) = true* **then**

13 $F_c.PNVect[F_p.posn] \leftarrow$ $F_p.sPreNo$

14 $Reclaim(F_p)$

     ----

15   ***updateDescPosns***(N_j, F_c)

16   **if** $j = TR$ **then**

17   **if** N_j *is result node* $\wedge$ ***evaluate(F_c)*** $=$ *true* **then**

18   Report Result;

19   Exit;

20   **if** *evaluate(F_c) = true* **then**

21 $p \leftarrow F_c.parIndex$;

22 $N_p \leftarrow$ ***parent***(N_j)

23 $N_p.Stack[F_c.parFrameNo].PNVect[F_c.posn]$ $\leftarrow F_c.sPreNo$

24   **if** N_j *is result node* **then**

25   Report Result;

26   **if** N_j *is tail of LR-Ordering* **then**

27 $N_{TR}.Stack$[**top**(N_{TR})]. $PNVect[N_j.TRPosn] \leftarrow F_c.SPreNo$

28   *Flush out all frames in the stacks of N_j and all its ancestors excluding twig root;*

29 $reclaim(F_c)$

We use a queue of result ids [4] which is maintained in the (only) stack frame of the root of the OaT. We assume that pre-order number is used as result id. The stack frame has an additional field to point to a record in the result queue. This pointer field is used only by frames in the stack of the result node of the OaT.

At the open-tag of the potential result element, say e,

---

[4] In the implementation we also maintain, along with each id, a pointer to the actual data structure area which holds the potential result element.

the *open-tag* handler pushes a frame into the stack of the result node and adds the id of the element to the tail of the queue. The pointer field in the stack frame is made to point to this new result id.

At the close-tag of e, the *close-tag* handler evaluates the stack-frame for e. If the result is false, the corresponding result id is deleted from the queue, if the deleted id was at the head of the queue, all the remaining ids in the queue are valid result ids (otherwise, they would have been removed during previous *close-tag* events) and hence can be output in head-to-tail order. Similarly, if the result of evaluation of the frame is true and the corresponding result id is at the head of the queue, all the ids can be output in head-to-tail order.
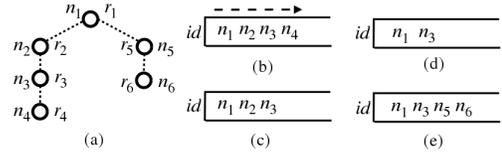

Figure 6: Illustrating Result Queue

Example 1 *In the partial document tree of Figure 6(a), n_1 to n_6 represent the pre-order numbers of potential result elements r_1 to r_6. Suppose that all elements except r_2 and r_4 are valid results. Figure 6(b) shows contents of the queue after open-tag event for the element r_4 is handled. At the close-tag of element r_4, id n_4 is deleted from the queue and no further action is needed as it was not at the head of the queue (Figure 6(c)). At the close-tag of r_2, id n_2 is deleted from the queue (Figure 6(d)). If r_1 were not present in the document, n_3 would have been output at this point as n_2 were at the head of the queue. Similarly, if r_2 is a valid result and r_1 is not present, both n_2 and n_3 can be output in that order. Figure 6(e) shows the contents of the queue before the close-tag event of r_1.*

5.2.4 Correctness of the algorithm

It can be seen that, before pushing a frame corresponding to an element in the stream to the stack of a query node, the algorithm ensures that the constraints specified on the node are satisfied (lines 10–13, 16–17 and 5–7 of Algorithm 1). In particular, suppose there is an LR-Ordering from node N_i to N_j of the OaT. Before pushing a frame to $N_j.stack$, the function *isLR-Compatible*() checks if the position in the pre-order number vector of the only frame in the root (that is, $N_i.TRPosn$) is already updated. Note that such an updation is done with the pre-order number of a frame in $N_i.stack$ that represents some element e in the stream. As the updation of the $N_j.TRPosn$ is done at the close-tag of e, a frame representing an element in the stream which is a descendant of e will

never be pushed to $N_j.stack$. This ensures correctness of the algorithm.

5.2.5 Value Predicates

In the OaT representation of a path expression with value predicates, the text value in the predicate appears under a leaf node. For such a leaf node, we maintain two additional fields in the node table – *val* and *oprtr*. The *val* field represents the text value that appears in the value predicate and *oprtr* is one of the relational operators or an XPath function such as *contains*. During query processing, the *text* event from the SAX parser in is intercepted to check the validity of the predicate.

5.3 Handling Closure Nodes

Note that a closure node is a wild-card node indicating a 'self-or-descendant' situation and that all children of a closure node are connected to the closure node through P-C edges (that is, the children are of type 'c'). Thus, a trivial solution would be to maintain, in the node table, two parents for the children of the closure node – the closure node and its parent node. For instance, each of the nodes g and h in the partial OaT of Figure 7(a), can have two parents – '$*$' and 'f'. When the *open-tag* event of element $\langle g, 10 \rangle$ of the partial document of Figure 7(b) occurs, the *open-tag* handler can check relationship with the top frame in the stack of parent node '$*$' and, if it fails, with the frame in the stack of parent 'f' (by modifying lines 8-13 of the *open-tag* handler). This trivial solution needs additional relationship checking. More seriously, it leads to non-uniform stack frame structure – the pre-order number vector of the stack frames for the parent of the closure node need to keep track of the additional positions for children of closure node. A better and efficient method is to use *self-frame*s. Intuitively, a self-frame is a convenient way to check the closure-edge relationship.

Lines 23–26 of Algorithm 1 are the additional steps needed to handle closure nodes during *open-tag* event processing. Figure 7 shows snapshots of processing a closure node. In the stacks of the figure, self-frames are shown with a small bubble to the left of it. Whenever a frame F is pushed to the stack of the parent of a closure node, a self-frame is pushed to the stack of the closure node. It is called a self-frame as the frame and its parent frame F represent the same element in the document. For instance, Figure 7(c) shows contents of relevant stacks after the *open-tag* event of element $\langle f, 9 \rangle$ of the partial document of Figure 7(b) is processed. The frame $\langle 3, 9, [-, -] \rangle$ that is pushed to S_* is a self-frame whose parent frame $\langle 3, 9, [-] \rangle$ in S_f also represents the same element. At the *open-tag* event of $\langle g, 10 \rangle$, a frame is pushed to S_g and linked with the top frame in S_* (Figure 7(d)). Note that, corresponding to the same element, a frame is pushed to S_* as the

element satisfies the closure-edge relationship between query nodes f and *. Subsequently, at the *close-tag* event of $\langle g, 10 \rangle$, the *PNVect* of the self-frame in S_* is updated. The stack contents after processing *open-tag* events for $\langle f, 11 \rangle$ $\langle g, 12 \rangle$ are shown in Figure 7(e). Here $\langle 4, 11, [-, -] \rangle$ is pushed to the stack S_* twice – as an ordinary child frame for $\langle 3, 9, [-] \rangle$ and a self-frame for $\langle 4, 11, [-] \rangle$, in that order (remember that the *open-tag* handler processes the '$*$'-node *before* the 'f'-node).

Lines 7–14 of Algorithm 2 show the additional steps for closure node processing. Before evaluating a frame F in the stack of the parent of closure node, the *close-tag* handler removes and evaluates the corresponding self-frame from the stack(s) of its closure node child(ren) and, if the result is *true*, updates the pre-order position(s) in F. For instance, when the *close-tag* event for $\langle f, 11 \rangle$ occurs, the self-frame in S_* is evaluated first, before evaluating its parent frame in S_f. Both the frames are removed from respective stacks as they are evaluated to *false*. As previously discussed, the same frame may be pushed as an ordinary frame and a self-frame. Thus, when the self-frame is removed, its pre-order vector can be copied to the pre-order vector of the ordinary frame, which will invariably be the next top frame in the stack (see Figure 7(f)). Subsequently ordinary frame $\langle 4, 11, [12, -] \rangle$ is also removed from S_* (see Figures 7(g) & (h) – remember that the *close-tag* handler processes the '$*$'-node *after* the 'f'-node). Figures 7(i) – (j) show actions of *open-tag* and *close-tag* handlers for the element $\langle h, 13 \rangle$. At the close-tag of $\langle f, 9 \rangle$, the frame in S_f is updated with its self-frame in S_* so that it will be evaluated to true during subsequent processing of the stream.

Note that, to ensure correct computation, the self-frame should be removed whenever its parent frame is removed before removing the ordinary frame in the closure node. Thus, we assume that the *indexSet()* function in the *close-tag* handler returns node indices in pre-order. As the frames are popped out in the reverse order of being pushed in, whenever a frame is popped out from the stack, its self-frames, if any, will be the topmost frames of respective stacks and can be directly popped out. Thus, no additional field is needed in the frame structure to distinguish a self-frame from an ordinary frame.

6 Experiments

In this section we compare performance of our algorithm with SPEX[7] on real world, synthetic and benchmark data sets. To the best our knowledge, SPEX is the only stream query processing system that implements ordered axes. Java implementation of the system is publicly available (*http::://spex.sourceforge.net*). Our algorithm was also implemented in Java. Xerces SAX parser from *http::/sax.sourceforge.net* was used to parse the XML documents. We ran all our experiments on a 1000 MHz
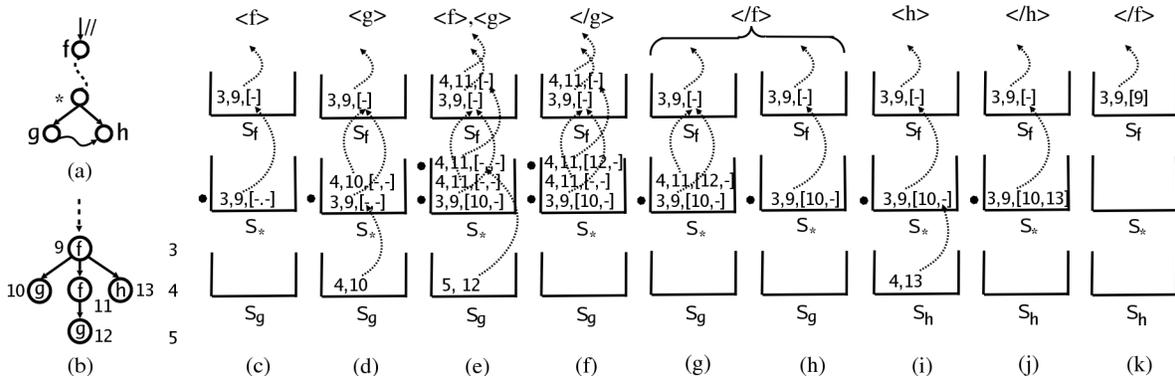
Figure 7: Illustrating Closure Node Processing

Table 1: Queries Tested

Query No	Query Expression
SQ1	///Features//SIGNAL//Descr/fs::LIPID /fs::CONFLICT/fl:Ref
SQ2	//Entry//Ref[.//MedlineID]/fs::keyword[(.="signal")]/fl::Features[/TRANSMEM/Descr]/fl::*/*
SQ3	//Entry//*[.="Fungi"]/fs::/Features/*/Descr[contains(.,"L->F")]/fl::Features/fs::*/Descr
TQ1	//NP/DT/fl::VP/NP/CD/fs::NNS/fl::VP/VB
TQ2	//S//NP/fs::COMMA[contains(., "gm")]/fl::*//NNP/fl::*[.//NNP]
TQ3	//VP/NP//NNS[contains(., "BOSF")]/fl::S/fs::*//DT/fl::*[.//VP]
XQ1	//mail/txt/keyword/fs::bold/fl::mail
XQ2	//item//*/text/keyword[contains(.,"armed")]/fs::keyword/fl::item/*[contains(., "of")]
XQ3	//item/description[.//text/keyword]//*/fs::mailbox//text/keyword[contains(., "george")]/fl::description

AMD Athlon 3000+ machine with 2GB memory running Linux. Java virtual machine (JVM) version 1.5 was used for conducting the tests. JVM maximum allocation pool was set to 1GB, so that virtual memory activity has no influence on the results.

We used three datasets in the experiments – SWISSPROT, TREEBANK[1] and XMARK[18]. SWISSPROT is a real world dataset. TREEBANK is a deeply recursive dataset containing English sentences tagged with parts of speech that was synthetically generated. XMARK is a benchmark dataset.

To make the comparisons uniform, we excluded the time needed for query pre-processing, stream parsing and result output from execution time.

Experiment 1: In this experiment, we tested, for each dataset, three types of queries – query *without* predicates and wild cards, query containing value predicates and wild cards and query that has to be represented as an OaT with closure-edges. The queries are shown in Table 1. Each query number is prefixed by the first letter of the dataset name for easy identification.

The results are shown in Figure 8. In all the cases, our algorithm (referred as FX from now) outperforms SPEX by wide margins. The performance gain is due to the 'awareness' of OaTs about order restrictions between nodes and effective use of this information during query processing. In particular, FX processes branches of the OaT's root in left-to-right order. A branch is *never* processed unless constraints specified

in the preceding branches are satisfied. And, once the algorithm finds that the stream satisfies branch, it flushes out all the frames of stacks along that branch of the OaT, thereby avoiding subsequent processing of those frames. Also, during *open-tag* event, no frame is pushed to the stacks of query nodes in branches that are already processed successfully (except the rightmost branch which contains the result node), which subsequently avoids popping out and processing of frames during corresponding *close-tag* events.

Note that, the second and third query in each set of queries (for instance, SQ2 and SQ3) take more time than the first query (SQ1). This is due to the presence wild cards and value predicates.

Experiment 2: Here we measured the variation of query processing time with document size (d) and query size (q – number of axis steps). TREEBANK dataset was used in these experiments. As the dataset is deep and highly recursive with majority of elements occur recursively along paths of the document tree, the probability that a randomly generated query gives empty result set is very low. We generated query sets containing fifteen queries using the set of element names in the dataset that are most recursive. During query generation, the axis and node test in each new axis step of the query was chosen randomly with uniform distribution. One query set was generated for each of the query sizes 5, 10, 15, 20 and 25. The data set was also split into chunks of increasing size each
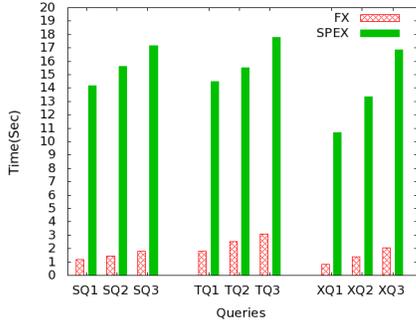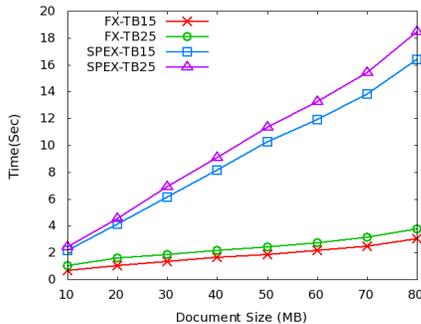
Figure 8: Query Processing Time (document size: Swissprot – 109M, Treebank – 82M, XMark – 109M)

(approximately) equal to multiple of 10M.

We tested how the average processing time of a query set varies with document size for query sizes 15 and 25. The plot of the results is shown in Figure 9(a). It can be seen that, FX scales much better than SPEX. The variation of processing time with query size for document sizes 40 and 80 was also measured and is shown in Figure 9(b). Here also performance of FX is better than that of SPEX.

Experiment 3: In this experiment, we measured the

(a) Document Size vs Time (q=15, 25)
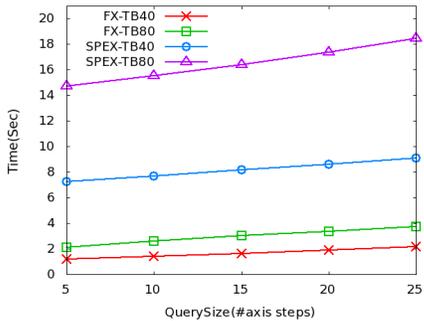
(b) Query Size vs Time (d=40, 80)

Figure 9: Scalability

effect of the number of *following* axis steps (f) and *following-sibling* axis steps (s) on query processing time using the TREEBANK dataset. To check the effect of *following* axis, we started with a path expres-

sion //NP//NP (denoted as FL0) and extended it by repeatedly adding the expression /fl::NP//NP in steps of 2 up to 10 (FL2, Fl4,FL6, FL8 and FL10). The element NP was selected as it occurs the maximum number of times in the dataset and is also deeply recursive and, hence, almost always produces non-empty result set. The resulting plot is shown in Figure 10(a). In case of FX, the time remains almost constant. This due to the left-to-right branch processing order and the avoidance of repeated processing of branches.

To check the effect of *following-sibling* axis, we started
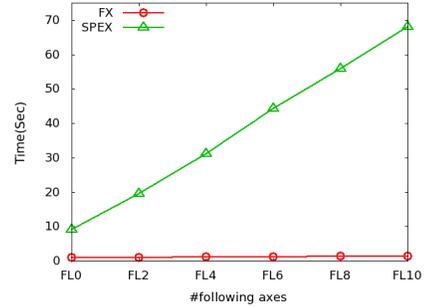
(a) Effect of fl axis (f=0–10 by 2)
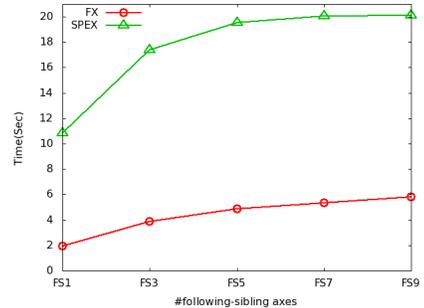
(b) Effect of fs axis (s=1–9 by 2)

Figure 10: Effect of Ordered Axes

with //NP/fs::* (denoted as FS1) and extended it by repeatedly adding //NP/fs::* in steps of 2 up to 9 (FS3, FS5, FS7 and FS9). Note that addition of each //NP/fs::* 'deepens' the OaT by adding a subtree rooted at a new closure node. We used * as the node-test for fs so that the result set will not be empty. Figure 10(b) shows the resulting plot. When there is only one *following-sibling* axis in the path expression, the results are identified by both the algorithms without processing deeper nodes in the document tree. As the number of *following-sibling* axes increases, both the algorithms examine deeper nodes also to identify the result. This is the reason for the initial slope of the graph. However, as the number of *following-sibling* axes increases further, the algorithms avoid processing of many elements in the stream. In case of FX, the conditions imposed at higher level nodes of the OaT fail and the algorithm avoids processing of deeper query nodes (remember that the depth of the OaT in-

creases with every //NP/fs::*). In case of SPEX, the conditions dictated by the earlier transducers in the transducer chain fail leading to less 'work-load 'on the transducers down the line. Hence the later part of the plots is almost parallel to the X-axis.

7 Conclusion

In this paper, we demonstrated how conventional twig structure can be extended by incorporating order information so that it can represent path XPath expressions with ordered axes. We also proposed an algorithm for processing XPath expressions with forward axes against streaming data. It was found that the algorithm is both efficient and scalable and outperforms currently available algorithm by sizeable margins. It would be interesting to investigate how the current algorithm can be extended to handle bigger XPath subsets with complex predicates and backward axes.

References

[1] XML Datasets. available at http://www.cs.washington.edu/ research/xmldatasets/.

[2] *XQuery 1: An XML Query Language.* W3C Recommendation 12 November 2003, available at http://www.w3.org/TR/xquery.

[3] *XML Path Language*, November 1999. available at http://www.w3.org/TR/xpath.

[4] Abdul Nizar M. and P. Sreenivasa Kumar. Order-aware Twigs: A Representation of XPath Expressions with Ordered Axes. Technical report, Department of CS & E, Indian Institute of Technology Madras, April 2008, Available at: *http://aidb.cs.iitm.ernet.in/tech-reports/tr-niz-04-08*.

[5] Aneesh Raj and P. Sreenivasa Kumar . Branch Sequencing Based XML Message Broker Architecture. In *ICDE*, pages 217–228, 2007.

[6] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath Processing with Forward and Backward Axes. In *ICDE*, pages 455–466, 2003.

[7] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. The xml stream query processor spex. In *ICDE*, pages 1120–1121, 2005.

[8] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering. In *VLDB Conference*, pages 559–570, 2006.

[9] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. pages 235–244, 2002.

[10] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. In *ICDE*, page 79, 2006.

[11] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Transactions on Database Systems*, 28(4):467–516, 2003.

[12] Feng Peng and Sudarshan S. Chawathe. XSQ: A streaming XPath engine. *ACM Trans. Database Systems*, 30(2):577–623, 2005.

[13] G. Gou and R. Chirkova. Efficient Algorithms for Evaluating XPath over Streams. In *SIGMOD Conference*, pages 269–280, 2007.

[14] A. K. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *SIGMOD Conference*, pages 419–430, 2003.

[15] S. Hou and H.-A. Jacobsen. Predicate-based Filtering of XPath Expressions. In *ICDE*, page 53, 2006.

[16] D. Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. Knowl. Data Eng.*, 19(7):934–949, 2007.

[17] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *EDBT Workshops*, pages 109–127, 2002.

[18] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB Conference*, pages 974–985, 2002.

[19] Vanja Josifovski and Marcus Fontoura and Attila Barta. Querying XML streams. *VLDB Journal*, 14(2):197–210, 2005.