Evaluating an XPath Query on a Streaming XML Document

Prakash Ramanan

Department of Computer Science
Wichita State University
Wichita, KS 67260–0083
ramanan@cs.wichita.edu

Abstract

We present an efficient algorithm for evaluating an XPath query Q (involving only `child` and `descendant` axes) on a streaming XML document D. Previously known in-memory algorithms for XPath evaluation use $O(|D|)$ space and $O(|Q||D|)$ time. Several previous algorithms for the streaming version use $\Theta(d^n + c)$ space and $\Theta(d^n|D|)$ time in the worst case; d is the depth of D, n is the number of location steps in Q, and c is the maximum number of *candidate* elements for output at any one time. In the worst case, the exponential $\Theta(d^n)$ space alone could well exceed the $O(|D|)$ space used by the in-memory algorithms. Our algorithm uses $O(d|Q|+cn)$ space and $O((|Q|+d+n^2)|D|)$ time in the worst case. So, our algorithm is runtime competitive with the in-memory algorithms, while using much less memory space.

1 Introduction

We consider the XPath *evaluation* problem: Evaluate an XPath query [5] Q on a streaming XML document D. Let d denote the depth of D, and n denote the number of location steps in Q.

We mostly consider queries that involve only the `child` and `descendant` axes. As shown in [15], queries involving `parent` and `ancestor` axes can be rewritten to involve only `child` and `descendant` axes. Also, as we point out later, our algorithm can be extended to queries that also involve the `preceding` and `preceding-sibling` axes *inside* the predicates, without increasing the memory space or runtime. This extension is an unique feature of our algorithm.

First, consider the case when D is fully available (or can be stored) in the memory. Gottlob et al. [9] and Ramanan [17] presented evaluation algorithms that use $O(|D|)$ space and $O(|Q||D|)$ time.

International Conference on Management of Data
COMAD 2005b, Hyderabad, India, December 20–22, 2005

From now onwards, let D be available only in streaming form. If Q does not have predicates, the evaluation problem is easy: A document node e should be output iff e and some of its ancestors (in D) match the location steps in Q; this can be completely determined when the start tag of e is seen. For this case, the *path stacks* of Bruno et al. [6] can be adapted to solve the evaluation problem. The resulting algorithm uses $O(dn)$ space and $O(n|D|)$ time.

When Q has predicates, the evaluation problem is more complicated. At the time a document node e is seen, we may or may not know if e should be output. As before, e should be output iff e and some of its ancestors f (in D) match the location steps in Q. Whether f satisfies the predicate in some location step might depend on some yet-to-be-seen descendants of f that are descendants/successors of e. So, in general, an algorithm, after seeing part of a streaming D, would have output some elements. There would be other (partly/fully) seen elements, called *candidates* for output: These are elements whose membership in the output can not be determined based on the part of D seen so far. So, any algorithm for this problem must store, at any instant, the candidates e as well as some of their ancestors f that could enable the candidates to qualify for the output.

There are two previously known algorithms for this problem: The XSQ algorithm of Peng and Chawathe [16], and the SPEX system of Olteanu et al. [14]. In the worst case, both these algorithms require $\Theta(d^n)$ space and $\Theta(d^n|D|)$ time just to handle the paths. The quantity d^n represents the number of different paths in D that a candidate e could take to reach the output; all these paths are embedded on the path from the root of D to e. This space alone could well exceed the $O(|D|)$ space used by the in-memory algorithms [9, 17] mentioned above.

We avoid using exponential space or time, by using efficient algorithms for three main components:

- *Path stacks* to compactly represent the paths that candidates could take to reach the output.

This is a much more elaborate version of the path stacks of [6].

- A *predicate checker* to determine when a node in the path stacks has satisfied/failed each predicate in Q. It scans the document bottom-up (while the nodes are streamed in document order), and determines which predicates in Q are satisfied/failed at each node.
- *Candidate stacks* to maintain the candidates and move them along to the output or trash.

We believe that our predicate checker would be of general use, in many other applications involving XPath.

For each path in the path stacks, the nodes on the path might satisfy their corresponding predicates in random order. A candidate should be output as soon as *any one of its paths* satisfies all the predicates. If we store all the paths as done in [6], then some redundant paths (that haven't yet satisfied the predicates) at the top of the stack might hide a "good" path (i.e., one that has satisfied the predicates) below them. This would result in delaying the output of a candidate, thereby costing memory space. We present a mechanism to avoid storing redundant paths, and to quickly determine when a candidate should be output. Maintenance of the path and candidate stacks constitute the most intricate parts of our algorithm.

Related to the evaluation problem studied here is the XPath *filtering* problem that arises in document dissemination: Given a set of XPath queries, determine which of those queries have a nonempty output on a given streaming XML document. [1, 7, 8, 10, 11] presented algorithms for various versions of this problem. Of these, the XPush machine [11] is the only algorithm that allows general predicates in the queries.

This paper is an abridged version of [18]. About the same time, [3] presented an algorithm for *filtering* a streaming XML document with respect to a single XPath query. In Section 4, we show that our predicate checker can be used for filtering, with memory and runtime requirements competitive with theirs. [12] presented an XPath *evaluation* algorithm; they claim that their algorithm runs in polynomial space and time, but no explicit complexity bounds are presented. Recently, [13] presented an evaluation algorithm that uses $O(d^2|Q| + |D|)$ space and $O(d|Q||D|)$ time. This is slightly worse than our space and time requirements given below. Also, it is not clear if their algorithm can handle nested predicates, and predicates containing `not` and library functions, like aggregations and `position` (our algorithm can).

The easiest case for our algorithm arises when all the location steps in Q have the `descendant` axis (outside the predicates). In this case, our algorithm uses $O(d|Q| + c)$ space and $O((|Q| + d)|D|)$ time; c is the maximum number of candidates stored at any one time. The technically hardest part of our algorithm arises when some location steps in Q have the `child` axes. For each candidate, we need to store *just enough* information to backtrack and try alternate paths, when one path fails; some of the nodes for an alternate path would have closed (i.e., we would have seen the ending tag) by then (see Section 6). In this case, our algorithm uses $O(d|Q| + cn)$ space and $O((|Q| + d + n^2)|D|)$ time in the worst case. These compare very favorably with the exponential $O(d^n + c)$ space and $O(d^n|D|)$ time requirements of [14, 16].

Most recently, [4] presented an XPath evaluation algorithm for *nonrecursive* documents: For any path P_Q in Q and any path P_D in D, there exists at most one embedding of P_Q in P_D. Their algorithm uses $O(|Q| \log |D| + C)$ bits of space and $O(|Q||D|)$ time, where C denotes the space required to store the candidates. For this special case, our algorithm uses $O(d|Q| + d \log |D| + C)$ bits of space and $O((|Q| + d)|D|)$ time. Since most real-life documents have small depth d, the resource requirements of the two algorithms are comparable. Our algorithm is much more general, and was not designed to give optimal performance for this very special case.

For the general case, since $n \leq depth(Q)$, our worst case runtime of $O((|Q| + d + n^2)|D|)$ is very competitive with the $O(|Q||D|)$ runtime of the in-memory algorithms [9, 17]; also, our algorithm uses much less memory space.

Other desirable features of our algorithm:

- Unlike in [16], candidates are not duplicated; so, there is no need to perform duplicate avoidance in the output.
- The kinds of predicates we allow in Q are more general compared to [14, 16]; in particular, we allow boolean combinations of predicates involving the operators `or` and `not`, nesting of predicates to arbitrary depths, and library functions like aggregations and the `position` function.
- Our algorithm can be extended to queries that involve the `preceding` and `preceding-sibling` axes inside the predicates, without increasing the memory space or runtime. This is an unique feature of our algorithm.

Our algorithm and those of [14, 16] can be used if D, instead of being streamed, is stored on a disk. When there are element-tag/atomic-value indexes on D, the $|D|$ in the space/runtime analyses denotes the sum of sizes of the streams returned by the indexes (after eliminating duplicates) for the vertices in Q.

Section 2 contains the preliminary definitions and notations we need. In Sections 3 and 4, we briefly describe the path stacks and our predicate checker, respectively. In Section 5, we describe our algorithm when all the location steps in Q have the `descendant` axis. In Section 6, we present the modifications to our algorithm, when some location steps in Q have the `child` axis. In Section 7, we present our conclusions.

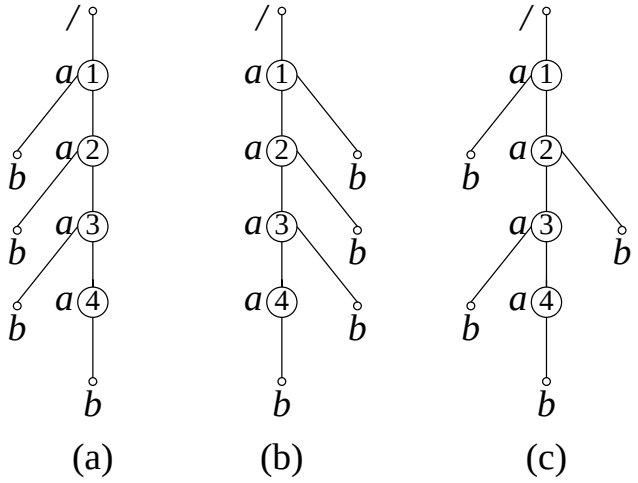

Figure 1: Example 2.1: Three different documents

2 Class of Queries, Output Order and SAX Events

An XPath query Q that we consider is of the form $L_1 L_2 \ldots L_n$, where each *location step* L_i is of the form <*axis*> <*node_test*> <*predicates*>. *Axis* is either `/` (for `child`) or `//`(`descendant`). Attributes are treated similar to elements. This class of queries is defined by the following grammar:

<*query*> ::= <*loc_step*> | <*loc_step*><*query*>
<*loc_step*> ::= <*axis*><*node_test*><*predicates*>
<*axis*> ::= / | //
<*node_test*> ::= `elem_label` | ∗ | `attr_label` | @∗
<*predicates*> ::= ϵ | [<*predicate*>]
<*predicate*> ::= <*predicate*> `and` <*predicate*>
| <*predicate*> `or` <*predicate*>
| `not` <*predicate*> | . <*query*>
| . <*query*> <*relOp*> `const`
<*relOp*> ::= < | ≤ | > | ≥ | = | ≠

. <*query*> indicates a relative query. Let $axis(L_i)$, $nodeTest(L_i)$ and $predicate(L_i)$ denote the axis, node test and predicate in step L_i, respectively.

There is a problem in evaluating queries on XML streams. In the usual model, the output of an XPath query on an XML document consists of the document nodes that match the last location step L_n; these nodes are to be output in document order. But in the stream model, the order in which the nodes are found to belong to the output (based on the document piece seen so far) might not match the document order.

Example 2.1. Consider the result of the query $Q = //a[b]$ on the three XML documents in Figure 1. For all three documents, the output consists of nodes 1–4; typically, these four nodes should be output in the document order $1, 2, 3, 4$. But when the documents are presented in streaming form, the order in which the four nodes are found to belong to the output might not match the document order. For Figures 1a, 1b and 1c, the nodes would be found to belong to the output in the order 1234, 4321 and 1342, respectively.

If we insist that the nodes be output in document order, then we have to buffer each output node, until its SAX *event#* (defined below) becomes the smallest among all candidates. This would result in the buffering of a large number of nodes that have already been found to belong to the output.

In practice, the desired output might be some function of the output nodes, such as an aggregate function (`sum, count`, etc), or an `attribute` value or `text` value of these nodes. For the reason given above, we assume that this function is *order invariant*; i.e., its value does not depend on the order in which its input arguments are presented.

The output of our algorithm is a sequence of pairs $(label(e), event\#)$, where $label(e)$ is the label of an output element e, and $event\#$ is the SAX event number for the `start` tag of e. For each document element found to belong to the output, we will immediately output the corresponding pair. This can be easily modified, to compute any order invariant output function or to output entire elements.

As in [11], we use a SAX parser to read the XML document, and generate a stream of events of five types:
`startDocument()`, `startElement(a, SAX event#)`, `text(s)`, `endElement(a)`, `endDocument()`.
Event# is the SAX event number, and `s` is a data (string) value. We treat attributes similarly to elements; so, the label `a` above might be an element label or an attribute label. For example, the XML document <*a b* = "101"><*c*> 201 </*c*></*a*> leads to the following sequence of SAX events:
`startDocument()`, `startElement(a,1)`, `startElement(@b,2)`, `text(101)`, `endElement(@b)`, `startElement(c,5)`, `text("201")`, `endElement(c)`, `endElement(a)`, `endDocument()`.

A document node (element or attribute) is *born* when its `startElement` is seen; it stays *open* until its `endElement` is seen, at which point it *closes*. A node is *current* if it is open, but none of its descendants is open; this is not affected by `text` events. All the open nodes lie on the unique path from the document root to the current node; this path is the *current path*.

3 Path Stacks

In this section, we describe the path stacks of [6]. Consider a query Q with no predicates. They use path stacks to compactly represent all possible embeddings of Q in the current path in D.

Example 3.1. Let $Q = //a//b/c//d$; in Figure 2a, `child` and `descendant` axes are represented by light and dark edges. Consider the current path $a_1 c_0 b_0 a_2$ $b_1 c_1 d_1 b_2 c_2 d_2$ (Figure 2c); we have used subscripts to distinguish between different elements with the same

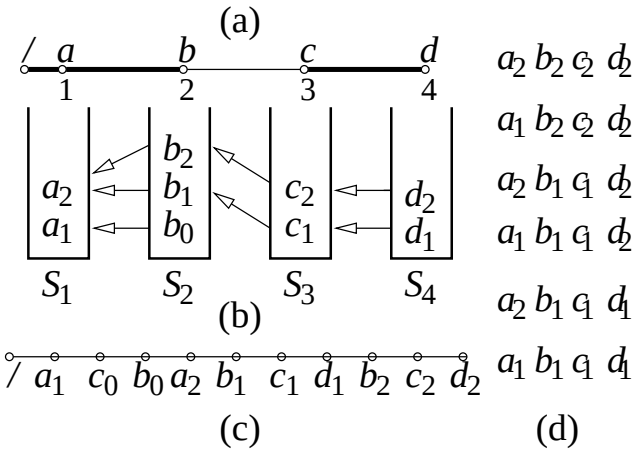

Figure 2: Example 3.1: (a). Query Q, (b). Path Stacks, (c). Current path in D, (d). The Embeddings

label. For $1 \leq i \leq 4$, S_i is the path stack for storing elements that are matches for location step L_i. The path stacks and the embeddings they represent are shown in the figure. Note that, since $axis(L_4) = //$, an embedding could use d_2 with either c_2 or c_1; but, since $axis(L_3) = /$, c_2 can only be used with b_2 (not with b_0 or b_1). ∘

How our use of path stacks differs from that in [6]:

- Our location steps have predicates. So, we need to keep track of whether an element in a path stack S_i has satisfied/not-yet-satisfied $predicate(L_i)$; if the element fails $predicate(L_i)$, it is removed from S_i. We use the predicate checker (Sec 4) to determine when an element satisfies/fails a predicate.
- We are not interested in outputting the embeddings themselves, but only the nodes that are matches for the last location step; in Example 3.1, these are the d nodes that are matches for L_4. So, we do not need to maintain all possible paths. If a_1 satisfied $predicate(L_1)$ at the time a_2 was born, a_2 should not be pushed onto S_1; since $axis(L_2) = //$, any path that uses a_2 could instead use a_1. Pushing a_2 would hide a_1, thereby delaying the output of some d elements, and costing space.
- Elements that are possible matches for L_n are candidates for output. In Example 3.1, suppose that candidate d_2 satisfies $predicate(L_4)$. If element c_2 fails $predicate(L_3)$, then it is removed from S_3; since $axis(L_4) = //$, d_2 could try an alternate path consisting of c_1; so, d_2 is made to point to c_1. Suppose that c_2 satisfies $predicate(L_3)$. Then, when c_2 closes, d_2 should be moved to a different stack (candidate stack C_2 attached to S_2) from which it can directly point to b_2. Since $axis(L_3) = /$, if b_2 fails $predicate(L_2)$, d_2 should be moved to a different stack (candidate stack C_3 attached to S_3) and made to point to c_1; c_1 could provide alternate paths for d_2 to reach the output. This complication in the presence of `child` axes is handled in Section 6. It involves backtracking (ex. moving d_2 from C_2 to C_3), and is the intellectually hardest part of the paper: When Q contains a sequence of `child` axes steps, we need to store additional information with each candidate to allow backtracking.

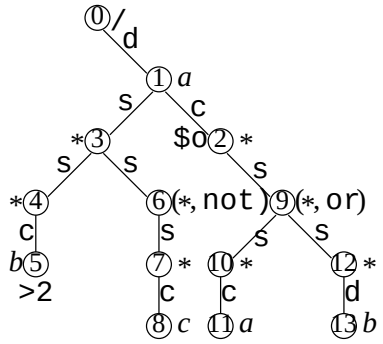

Figure 3: $P = [.//a[b[> 2]$ `and not` $c]/ * [a$ `or` $.//b]]$

4 The Predicate Checker

The *predicate checker* determines when a node in the path stacks has satisfied/failed each predicate in Q, using only $O(d|Q|)$ space and $O(|Q||D|)$ time. This predicate checker is similar to the *bottom-up transducer* in [19].

As per our grammar given in Section 2, a `predicate` could contain the boolean operators `and`, `or` and `not`. Such a predicate P can be represented by a tree $tree(P) = (V, A)$ where V is a set of vertices, and A is a set of arcs. Each vertex $v \in V$ has a type $\tau(v)$, and a boolean operator $bool(v)$ associated with it. $\tau(v) \in \Sigma \cup \{*\}$ is the element type of v. $Bool(v) \in \{$`and`, `or`, `not`$\}$. Each arc $r \in A$ has an axis $axis(r)$ associated with it; $axis(r) \in \{$`self`, `child`, `descendant`$\}$. If v is a leaf vertex, optionally, there could be a "<*relOp*> `const`" condition associated with v.

Example 4.1. For predicate
$P = [.//a[b[> 2]$ `and not` $c] / * [a$ `or` $.//b]]$,
$tree(P)$ is shown in Figure 3. For each vertex v, the pair $(\tau(v), bool(v))$ is shown next to v; if $bool(v)$ is not specified, it should be taken to be `and`. For each arc r, $axis(r)$ is shown next to it: `s`, `c` and `d` stand for `self`, `child` and `descendant`, respectively. ∘

The predicate checker must consider the labels $bool(v)$ for each vertex v, and $axis(r)$ for each arc r. To avoid cluttering our description with these details, we describe our predicate checker only for predicates that do not contain the operators `or` and `not`; the extension to general predicates is tedious but straight forward. Some pointers are given later for `not`. Then, $bool(v) =$ `and` for each vertex; so it can be left out. Also, $axis(r)$ is either `child` or `descendant` for each arc. Let r be the arc from vertex j to vertex j'. If

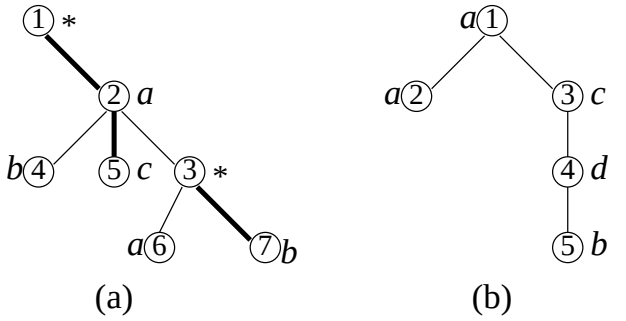

Figure 4: Examples 4.2–4.3: (a). $P = [.//a[b \texttt{ and } .//c] / * [a \texttt{ and } .//b]]$, (b). a document fragment

$axis(r) = \texttt{child}$, we say that j' is a c-child of j; if $axis(r) = \texttt{descendant}$, we say that j' is a d-child of j.

Example 4.2. For $P = [.//a[b \texttt{ and } .//c]/*[a \texttt{ and } .//b]]$, $tree(P)$ is shown in Figure 4a. ∘

In general, $|tree(P)|$ is linear in $|P|$. From now onwards, we will not distinguish between P and $tree(P)$. To minimize confusion, we will use the terms *vertices* and *arcs* while referring to the components of P; *nodes* and *edges* refer to the components of D.

The predicate checker M is constructed from the forest $F = \{tree(predicate(L_i)) \mid 1 \le i \le n\}$. Let the vertices in F be numbered consecutively from 1 to m; $m = O(|Q|)$. M maintains a stack S of records. The records in S, from bottom to top, correspond to the (open) elements on the current path in D. Each open element e is represented by the record $record(e) = (label(e), event\#, self_e, child_e, desc_e)$. $Event\#$ is the SAX event number for the `startElement` event for e. $Self_e$, $child_e$ and $desc_e$ are three boolean arrays indexed from 1 to m. Let D_e denote the document subtree rooted at e. For $1 \le j \le m$, let P_j denote the subtree rooted at vertex j in F. The arrays are defined as follows: At any instant,

S1 $self_e[j] = 1$ iff there exists an embedding of P_j in the part of D_e seen so far, with j mapped to e.

C1 $child_e[j] = 1$ iff there exists an embedding of P_j in D, with j mapped to some child of e seen.

D1 $desc_e[j] = 1$ iff there exists an embedding of P_j in D, with j mapped to some descendant of e seen so far.

Example 4.3. Consider the evaluation of the predicate P shown in Figure 4a, on the document fragment shown in Figure 4b. The vertices in $F = \{tree(P)\}$ are numbered 1 to 7 (i.e., $m = 7$). After document node 3 closes, we have $self_1 = 0010010$, $child_1 = 0000110$ and $desc_1 = 0001111$. For example, $self_1[3] = 1$ because there exists an embedding of P_3 such that vertex 3 is mapped to node 1. $child_1[5] = 1$ because there exists an embedding of P_5 such that vertex 5 is mapped to node 3 (a child of node 1). $desc_1[7] = 1$ because there exists an embedding of P_7 such that vertex 7 is mapped to node 5 (a descendant of node 1). ∘

Let us see how M operates on each of the five kinds of SAX events. After M processes each SAX event, the invariants S1, C1 and D1 stated above would hold at the current element.

`startDocument`: S is initialized to empty. Current element is `/` , with $event\# = 0$, $self = child = desc = \vec{0}$.

`startElement`: The record for the current element is pushed onto S. The newborn element e becomes the new current element, with $self = child = desc = \vec{0}$. For each leaf vertex $j \in F$ such that $label(e)$ matches $\tau(j)$, and there is no "$<relOp>$ `const`" condition associated with j, set $self[j] = 1$.

`text` : Let e be the current element. Consider a leaf $j \in F$ such that $label(e)$ matches $\tau(j)$, and there is a "$<relOp>$ `const`" condition associated with j. If the string value in the text event satisfies the condition at j, set $self[j] = 1$; additionally, if j is the root of some $predicate(L_i)$, output that e has satisfied $predicate(L_i)$.

`endElement`: Let e be the current element that is closing. For each $predicate(L_i)$ whose root vertex $r \in F$ has $self_e[r] = 0$, output that e has failed $predicate(L_i)$. Pop the top record from S; let it be $record(e')$; e' becomes the new current element. Update $child_{e'}$, $desc_{e'}$ and *then* $self_{e'}$ as follows:

- $child_{e'}[j] = child_{e'}[j] \ \vee \ self_e[j]$
- $desc_{e'}[j] = desc_{e'}[j] \ \vee \ self_e[j] \ \vee \ desc_e[j]$
- If $self_{e'}[j] = 0$ then set it to 1 if:
  - $label(e')$ matches $\tau(j)$,
  - for each c-child j' of j, $child_{e'}[j'] = 1$, and
  - for each d-child j' of j, $desc_{e'}[j'] = 1$.

  Also, if j is the root of some $predicate(L_i)$, output that e' has satisfied $predicate(L_i)$.

`endDocument`: Current element must be '`/` ', and S must be empty. Discard $record('/')$.

Now, let us see how to handle the `not` operator. Consider the predicate P in Figure 3. For a document node e, we have $self_e[6] = 1$ iff $self_e[7] = 0$; so, $self_e[7]$ should be computed before $self_e[6]$, because of the `self` arc from vertex 6 to vertex 7.

A note about the output of M: For a SAX event, when M "outputs" that some element e has satisfied or failed a predicate L_i, it adds L_i to set L^T or L^F for e, respectively. L^T and L^F are sets of locations steps whose predicates are found to be true and false, respectively, at e, as a result of *this* SAX event. The pair (L^T, L^F) is returned to the stream processing algorithm described in Sections 5 and 6. For a general predicate (containing `or` and `not`), after processing a `text` SAX event, M outputs a pair (L^T, L^F) for the current element e. For the `endElement` SAX event for e, M produces two (L^T, L^F) pairs: one for e, and another for its parent e'.

Resource requirements of M: The forest F takes space $O(|Q|)$. The stack S contains exactly the open document nodes. Each record in the stack needs space $O(|Q|)$. So, the total space is $O(d|Q|)$. Now, consider runtime. Setting up the forest F takes time $O(|Q|)$. In the operation of M, each event takes $O(|Q|)$ time. We show this for `endElement` events; trivial for others. M spends $O(|Q|)$ time on the closing element e. On the new current element e', updating *child* and *desc* takes $O(|Q|)$ time. Updating $self_{e'}[j]$ takes time proportional to $outdegree(j)$ in F; over all $j \in F$, this takes $\sum_{j \in F} outdegree(j) = O(|Q|)$ time. So, M spends $O(|Q|)$ time for each `endElement` event. Since there are totally $O(|D|)$ events, runtime is $O(|Q||D|)$.

Theorem 4.1. The predicate checker M correctly determines when the current element in D has satisfied/failed each predicate in Q. It uses $O(d|Q|)$ space and $O(|Q||D|)$ time.

Proof. Note that M updates the arrays $self_e$, $child_e$ and $desc_e$ only when e is the current element. Using induction on time, we can prove that invariants S1, C1 and D1 above hold for e, after M processes each SAX event. Correctness of M follows from S1. □

The XPush machine [11] can be extended and used in place of our BUT, but it would need memory space and runtime exponential in $|Q|$. Also, unlike the XPush machine, our predicate checker can be extended to predicates that involve some XPath library functions, including aggregation and `position`; the contents of arrays $self$, *child* and *desc* will be more general (not boolean). It can also be extended to predicates that involve `preceding` and `preceding-sibling` axes. We believe that our predicate checker would be of use in other XML applications.

Our predicate checker can also be used to *filter* an XML document with respect to an XPath query. For filtering, an XPath query is just a predicate on the document root: For example, $/a[b] \equiv /[a[b]]$, and $//a[b] \equiv /[.//a[b]]$. So, an XML document D passes the filter iff $self[v] = 1$ at the document root, where v is the root of the query tree. Our predicate checker can determine this using $O(d|Q|)$ bits of space and $O(|Q||D|)$ time. [3] considered the class of *univariate XPath* queries; this is same as the class of queries we consider in this paper (see Section 2).They presented a filtering algorithm that uses $O(r|Q|(\log|Q| + \log d))$ space and $O(r|Q||D|)$ time; r denotes the "recursion depth" of D with respect to Q, $1 \leq r \leq d$. Our algorithm is faster by a factor of $\Theta(r)$, allowing faster streaming. For memory space, there are instances where our algorithm is better, and vice versa. It is well-known that most real life XML documents have small depth. So, our algorithm is definitely competitive with theirs, in terms of memory space, in most cases of interest. Also, our algorithm is lot simpler.

5 Our Algorithm When There Are No Child Axes

In this section, we present our algorithm for evaluating Q on D, when all the location steps in Q have the `descendant` axis; Section 6 contains modifications for handling `child` axis steps. Note that we are not concerned about the axes inside the predicates attached to these location steps; the predicate checker in Section 4 handles those axes.

Detailed pseudo code for our algorithm is given in [18]; important parts are given below. In each line, the character sequence "/**" precedes comments. Procedures *PStartDocument*, *PStartElement*, *PText* and *PEndElement* are called in response to SAX events `startDocument`, `startElement`, `text` and `endElement`, after the predicate checker (Section 4) has processed the event.

Our algorithm uses path stacks and candidate stacks. Candidate stacks contain *candidate* nodes for output; these are elements that are possible matches for L_n. The path stacks together maintain a compact representation of the *possible* paths, consisting only of open nodes, that a candidate could take to reach the output (or be discarded along the way if all relevant paths fail). Each such path of length i is a possible match for $L_1 L_2 \cdots L_i$, subject only to the satisfaction of the predicates in each location step.

We first describe the path stacks. There are n path stacks: For $1 \leq i \leq n$, path stack S_i corresponds to location step L_i; it contains open nodes that are possible matches for L_i. The nodes in S_i, from the bottom of the stack to the top, lie on the current path. The same open node might be in several path stacks, and some open nodes might not be in any path stack.

Let $top(S_i)$ denote the top record in S_i, and let T_i denote the pointer to $top(S_i)$. The variable *nempty* keeps track of the last nonempty path stack: $S_1, S_2, \ldots, S_{nempty}$ are all nonempty. Procedure *PStartDocument* initializes the path stacks, candidate stacks, their top pointers, and the variables *nempty* and *satUpto* (defined later).

A document node e is pushed onto a path stack only when e is *newborn*; i.e., the most recent SAX event is the `startElement` event for e. Procedure *PStartElement* pushes e onto S_i iff it satisfies three conditions:

1. e passes $nodeTest(L_i)$
2. e has an ancestor element in S_{i-1}. If we proceed in decreasing order of i, then this condition is satisfied iff S_{i-1} is nonempty; i.e., $i \leq nempty + 1$.
3. e is not *redundant* in S_i. This is an optimization measure that will be explained later.

When the procedure pushes e onto S_i, it actually pushes the record
$R_i(e) = (label(e), event\#, predStatus, leftPtr)$.
$PredStatus$ is either $True$ or $Unknown$, indicating whether node e has already-satisfied/not-yet-satisfied

predicate(L_i), respectively; $R_i(e)$ is popped from S_i when e fails *predicate*(L_i), or when e closes. $LeftPtr$ is the value of T_{i-1} when $R_i(e)$ is pushed onto S_i, and after that its value stays constant; it points to the top most element of S_{i-1} that is an `ancestor` (not `self`) of e. In what follows, when we say "element e in S_i", we actually mean the record $R_i(e)$.

Let e be an element in S_i. $S_1, S_2, \ldots, S_i$ together provide a compact representation of a set $Paths(i, e)$ of paths that are *possible* matches for $L_1 L_2 \cdots L_i$. The last node on the path (i.e., *possible* match for L_i) is either e or one of the nodes below it in S_i; let it be some node e'. Then the second last node on the path (i.e., *possible* match for L_{i-1}) is either the element f in S_{i-1} that $R_i(e').leftPtr$ points to, or one of the elements below f in S_{i-1}; and so on. Some of these nodes already satisfy their corresponding predicates (i.e., have $predStatus = True$), while others have not (i.e., have $predStatus = Unknown$). If all the nodes on one of the paths in $Paths(i, e)$ satisfy their corresponding predicates, then that path becomes an *actual match* for $L_1 L_2 \cdots L_i$.

Let $P' = (e'_1, e'_2, \ldots, e'_i)$ and $P = (e_1, e_2, \ldots, e_i)$ be two paths in $Paths(i, e)$; for $1 \leq j \leq i$, $e_j, e'_j \in S_j$. We say that P' is below P (denoted $P' \preceq P$) if e'_j is either same as e_j or is below e_j in S_j, for $1 \leq j \leq i$; i.e., each element e'_j in P' is an `ancestor-or-self` of the corresponding element e_j in P. The following fact characterizes redundant paths.

Fact 5.1. If $P' \preceq P$ and P' is an actual match for $L_1 L_2 \cdots L_i$, $i < n$, then P is redundant: Any candidate that could use P to reach the output can instead use P' and reach the output right away.

Example 5.1. Suppose that, in Figure 2, we change $axis(L_3)$ from `child` to `descendant`; so, all the four location steps have the descendant axis. The contents of path stacks remain as before. But $Paths(4, d_2)$ now consists of the six paths shown in Figure 2d, and the five additional paths $a_2 b_1 c_2 d_2$, $a_1 b_1 c_2 d_2$, $a_1 b_0 c_2 d_2$, $a_1 b_0 c_1 d_2$ and $a_1 b_0 c_1 d_1$.

$a_2 b_1 c_1$ is below $a_2 b_2 c_2$. So, if $a_2 b_1 c_1$ is an actual match for $L_1 L_2 L_3$, then $a_2 b_2 c_2$ is redundant: Any candidate, such as d_2, that is waiting on the paths in $Paths(3, c_2)$ to reach the output, can use $a_2 b_1 c_1$ and reach the output right away. ∘

To avoid storing redundant paths, and to recognize an actual match when there is one, we introduce the variable *satUpto* (abbreviation for "satisfied upto"). It is the largest integer such that: For $1 \leq i \leq satUpto$, $top(S_i)$ satisfies *predicate*(L_i) and points to $top(S_{i-1})$.

Fact 5.2. The top elements of stacks $S_1, \ldots, S_{satUpto}$ form an actual match for $L_1 L_2 \cdots L_{satUpto}$. There are two consequences: For $1 \leq i \leq satUpto$,

- Any candidate that could use any of the paths in $Paths(i, top(S_i))$ to reach the output can go to the output right away.
- No need to push any element on S_i.

The second item in Fact 5.2 deals with avoiding redundancy in the prefix $S_1, S_2, \ldots, S_{satUpto}$. This does not apply to S_n, as candidates can not be redundant.

Redundancy can also arise at S_i, $satUpto < i < n$, as follows. Suppose that $R_i(e').pred\ Status = True$ and $R_i(e').leftPtr = T_{i-1}$. Then pushing an element e above e', in S_i, is redundant; in an actual match that uses e, e can be replaced by e'.

Example 5.2. Consider Figure 2 (with $axis(L_3) = //$). First, let us consider avoiding redundancy in $S_1 S_2 \cdots S_{satUpto}$. Suppose that at the time b_2 is born, a_2 and b_1 satisfy *predicate*(L_1) and *predicate*(L_2), respectively; so $satUpto \geq 2$. Then b_2 is redundant in S_2, and should not be pushed onto S_2.

Now, consider avoiding redundancy past $S_{satUpto}$. At the time b_2 is born, suppose that a_1 and a_2 have not yet satisfied *predicate*(L_1); so $satUpto = 0$. Also, suppose that b_1 satisfies *predicate*(L_2). Then b_2 is redundant in S_2. ∘

For $satUpto < i < n$, redundancy in S_i is avoided by using the procedure $notRedPush(i, R)$ (code skipped). It does not push R onto S_i if $top(S_i)$. $predStatus = True$ and $top(S_i).leftPtr = T_{i-1}$.

Lemma 5.1. Procedure PStartElement pushes a newborn element e onto the appropriate path stacks S_i, while avoiding redundancy.

Proof. The procedure pushes e onto S_i for which the three conditions enumerated at the beginning of this section are met. For $i < n$, the third condition ("e is not redundant in S_i") translates to restricting $i > satUpto$, and using the procedure *notRedPush*. For $i = n$, there is no concept of redundancy: e is either output (if e is already found to belong to the output) or it is pushed onto S_n. □

Now, let us consider candidates and candidate stacks. Candidates start their candidacy in S_n: S_n contains open nodes that are possible matches for L_n, and these are all candidates for output. For an element e in S_n, its $leftPtr$ gives the set of paths $Paths(n-1, *(R_n(e).leftPtr))$ that e could take to reach the output ($*p$ denotes the dereferencing of pointer p).

Fact 5.3. An element e in S_n should eventually reach the output iff the following conditions are satisfied:

1. e satisfies *predicate*(L_n).
2. Any one path in $Paths(n-1, *(R_n(e).leftPtr))$ becomes an actual match for $L_1 L_2 \cdots L_{n-1}$.

If e in S_n fails condition 1), its candidacy dies, and it is popped from S_n and discarded. Let e meet condition 1). If, at the time condition 1) is met, condition 2) is also met (i.e., $satUpto = n-1$ and $R_n(e).leftPtr = T_{n-1}$), then e is output immediately (this results in failing the conditions to be met to increment *satUpto* to n; so, $satUpto < n$ always). If condition 2) is not

met at that time then, by Fact 5.5 below, condition 2) can not be met until e closes; e is kept in S_n until e closes, and then e is moved to candidate stack C_{n-1}.

Example 5.3. In Figure 2b, consider the following scenarios:

- d_2 fails $predicate(L_4)$: d_2's candidacy dies; it is popped from S_4 and discarded.
- $satUpto = 3$ when d_2 satisfies $predicate(L_4)$: $Top(S_1)top(S_2)top(S_3) = a_2b_2c_2$ is an actual match for $L_1L_2L_3$. d_2 is popped from S_4 and output.
- $satUpto < 3$ when d_2 satisfies $predicate(L_4)$: No path in $Paths(3, c_2)$ is currently known to be an actual match for $L_1L_2L_3$. Then, none of these paths can become an actual match for $L_1L_2L_3$ until d_2 closes. d_2 is kept in S_4 until d_2 closes, and then moved to candidate stack C_3 attached to S_3, with its $pathPtr$ pointing to c_2. ∘

For $1 \leq i < n$, C_i is the candidate stack attached to S_i. The C_is together contain all closed elements that are candidates (open candidates are in S_n). Also, the C_is are disjoint: No candidate appears in two or more C_is simultaneously. An element e in C_i is represented by the record $R'_i(e) = (label(e), event\#, pathPtr)$, where $pathPtr$ is the value of T_i when e is pushed onto C_i. In actuality, we group all the elements in C_i that have the same value for $pathPtr$ (they must be contiguous in C_i) into a *bunch* with a single $pathPtr$.

Fact 5.4. Let Bunch be a bunch of candidates in C_i that have the same value for $pathPtr$. All the candidates in Bunch are relying on the same set of paths $Paths(i, *pathPtr)$ to reach the output. All these candidates should be output iff (and when) some path in $Paths(i, *pathPtr)$ becomes an actual match for $L_1L_2 \cdots L_i$; this happens exactly when $satUpto = i$ and $pathPtr = T_i$.

Bunches can be combined/moved-to-C_{i-1}/output/discarded, but a bunch can never be split. Let $topBunch(C_i)$ denote the top bunch in C_i. Let us consider changes to $top(S_i)$ and $topBunch(C_i)$, based on the operations of the predicate checker, following a SAX event. Recall that the predicate checker always operates on the current node. In our path stacks also, we always operate only on the current node. We have the following.

Fact 5.5. Whenever we push/pop/access/modify /delete the record pertaining to an element e, in any path stack, e must be the current element. None of e's descendants is open. So, if e is in any path stack, it must be the top element of that stack; also, no element from the next path stack can point to it (thru $leftPtr$).

As explained in Section 4, L^T and L^F are sets of location steps whose predicates become $True$ and $False$, respectively, at current element e, as a result of

procedure PStartElement(a, event#)
/**Push new element onto appropriate path stacks
for each i, $satUpto < i \leq nempty + 1$,
 such that a matches $nodeTest(L_i)$, in ↓ order of i, do
 /**All descendants of this new element are unborn.
  if $(i = n)$
  then if $predicate(L_n) = nil$
       then if $satUpto = n - 1$
            then output $(a, event\#)$
            else $push(S_n, (a, event\#, True, T_{n-1}))$
       else $push(S_n, (a, event\#, Unknown, T_{n-1}))$
  else if $predicate(L_i) = nil$
      then $notRedPush(i, (a, event\#, True, T_{i-1}))$
           if $i = satUpto + 1$ then $satUpto++$;
      else $notRedPush(i, (a, event\#, Unknown, T_{i-1}))$

procedure deleteFalse(e, L^F)
/**e fails the predicates in the location steps $L_i \in L^F$;
/**so, delete e from corresponding path stacks.
 for each i such that $L_i \in L^F$ and $top(S_i) = R_i(e)$ do
  /**$satUpto < i \leq nempty$
  /**and $top(S_i).predStatus = Unknown$.
   if $((i < n)$ *and* $(topBunch(C_i).pathPtr = T_i))$
      then $bunch \leftarrow popBunch(C_i)$ else $bunch \leftarrow \phi$;
   $pop(S_i)$
   if $((T_i \neq nil)$ and $(bunch \neq \phi))$
                    then $pushBunch(i, bunch)$

procedure setTrue(e, L^T)
/**e satisfies the predicates in location steps $L_i \in L^T$;
/**so, update $R_i(e)$ and C_i.
 for each i such that $L_i \in L^T$ and $top(S_i) = R_i(e)$ do
  /**$satUpto < i \leq nempty$
  /**and $top(S_i).predStatus = Unknown$.
   $top(S_i).predStatus \leftarrow True$
   if $((i = n)$ *and* $(satUpto = n - 1)$ *and*
                    $(top(S_n).leftPtr = T_{n-1}))$
   then pop and output $top(S_n)$;
        go to the next pass of the *for* loop
   if $((i = satUpto + 1) \wedge (top(S_i).leftPtr = T_{i-1}))$
   then satUpto++;
        if $topBunch(C_i).pathPtr = T_i$
        then pop and output $topBunch(C_i)$

procedure deleteTrue(e)
/**e is closing;
/**delete e ($predStatus = True$) from all path stacks
 for each i s. t. $top(S_i) = R_i(e)$, in ↑ order of i, do
  /**$1 \leq i \leq nempty$ and $top(S_i).predStatus = T$.
  /**$Top(S_i).leftPtr = T_{i-1}$ (because of ↑ order of i).
    if $(i = n)$
    then $pop(S_n)$
         $pushBunch(n - 1, (label(e), event\#, T_{n-1}))$
    else if $topBunch(C_i).pathPtr = T_i$
         then $bunch \leftarrow popBunch(C_i)$
              $pushBunch(i - 1, bunch)$
         $pop(S_i)$; if $i = satUpto$ then $satUpto - -$

processing the most recent `text` or `endElement` SAX event. These sets are computed and returned by the predicate checker. Consequently, there are three possible cases pertaining to e in S_i:

- e fails $predicate(L_i)$. See procedure $deleteFalse$. e is no longer a possible match for L_i, and should be popped from S_i. $TopBunch(C_i)$, if pointing to e thru $pathPtr$, should have its $pathPtr$ set to the new value of T_i; this could result in merging this bunch with the one below it in C_i, if both these bunches have the same value for $pathPtr$. If S_i is empty after popping e, $topBunch(C_i)$ should be popped from C_i and discarded, emptying C_i.
- e satisfies $predicate(L_i)$. See procedure $setTrue$. Set $R_i(e).predStatus = True$. If $satUpto = i - 1$ and $R_i(e).leftPtr = T_{i-1}$, then increment $satUpto$; the top elements of $S_1S_2 \cdots S_i$ form an actual match for $L_1L_2 \cdots L_i$; if $topBunch(C_i).pathPtr = T_i$, then pop and output all the elements in that bunch. Else no change to $topBunch(C_i)$; in particular, we do not yet move this bunch to C_{i-1}, because it will not be the top bunch in C_{i-1} if e is also in S_{i-1}.
- e closes. We must have $R_i(e).predStatus = True$. See procedure $deleteTrue$. We delete e from S_i, in increasing order of i. If $topBunch(C_i).pathPtr$ was pointing to e, then, since e is a match for L_i, we move this bunch from C_i to C_{i-1}. Decrement $satUpto$, if necessary, to reflect the deletion of e.

Example 5.4. Continuation of Example 5.3 (third scenario). d_2 has been moved to C_3, with its $pathPtr$ pointing to c_2. Consider three cases for c_2:

- c_2 fails $predicate(L_3)$: c_2 is popped from S_3. d_2 in C_3 now points to c_1. b_2 closes with or without satisfying $predicate(L_2)$, and is popped from S_2. If d_1 fails $predicate(L_4)$, it is popped from S_4 and discarded. Suppose that d_1 passes $predicate(L_4)$. When d_1 closes, it is moved to C_3, forming a single bunch with d_2, with their $pathPtr$ pointing to c_1.
- $satUpto = 2$ and c_2 passes $predicate(L_3)$: $satUpto$ is incremented to 3; this indicates that $top(S_1)top(S_2)top(S_3) = a_2b_2c_2$ is an actual match for $L_1L_2L_3$. Since $d_2.pathPtr = T_3$, d_2 is output.
- $satUpto < 2$ and c_2 passes $predicate(L_3)$: $top(S_3).predStatus$ is set to true; no change to C_3 (i.e., d_2). When c_2 closes, it is popped from S_3. Only then d_2 is moved from C_3 to C_2, with its $pathPtr$ pointing to b_2. ∘

Lemma 5.2. Procedures $deleteFalse$, $setTrue$ and $deleteTrue$ correctly handle the three cases itemized prior to Example 5.4, respectively.

Following a `text` event, the predicate checker returns a pair (L^T, L^F) for the current element e (see Section 4). Then, procedure $PText$ processes the changes at e using procedures $deleteFalse$ and $setTrue$. Following an `endElement` event, the predicate checker returns a pair (L^T, L^F) for the closing element e, and another pair for its parent e' (see Section 4). Procedure $PEndElement$ first processes the changes at e, using procedures $deleteFalse$, $setTrue$ and $deleteTrue$. Then, it process the changes at e' using procedures $deleteFalse$ and $setTrue$.

Procedure $popBunch(C_i)$ (code not given) pops and returns $topBunch(C_i)$. Procedure $pushBunch(i, bunch)$ pushes $bunch$ on C_i after, if necessary, merging it with $topBunch(C_i)$.

Lemma 5.3. Procedures $PText$ and $PEndElement$ correctly process the changes resulting from a `text` event and an `endElement` event, respectively.

Resource requirements of our algorithm: We have already seen the requirements for the predicate checker. Now, consider path stacks and candidate stacks. First, consider memory space. In the worst case, each open node could be in each path stack; space required for path stacks is $O(nd)$. The candidate stacks contain one copy of each candidate; worst case space required is $O(c)$. So, the overall memory space required (including space for the predicate checker) is $O(d|Q| + c)$.

Now, consider runtime. We spend $O(1)$ time for each element e, in each path stack; the worst case time spent on path stacks is $O(n|D|)$. For each candidate e, time spent on candidate stacks is proportional to the number of different values its $pathPtr$ takes ($\leq d$); this includes the movement of e from C_i to C_{i-1}, as the target of its $pathPtr$ varies from S_i to S_{i-1}. So, the overall worst case runtime, including the runtime of the predicate checker, is $O((|Q| + d)|D|)$. It is not much worse than the runtime $O(|Q||D|)$ of the best in-memory algorithms [9, 17] that use $\Theta(|D|)$ memory space. So, our algorithm is runtime competitive with these algorithms, while using much less memory space.

Theorem 5.1. The algorithm described in this section correctly evaluates an XPath query Q on a streaming XML document D, when all the location steps in Q have the `descendant` axis. The algorithm uses $O(d|Q| + c)$ space and $O((|Q| + d)|D|)$ time.

Proof. The correctness proof follows from Facts 5.1 to 5.5 and Lemmas 5.1 to 5.3. □

6 When There Are Child Axes

In this section, we consider the modifications needed in our algorithm, when some location steps have the `child` axis. This is the intellectually hardest part.

First, consider the modifications pertaining to path stacks. In the previous section, we saw that a newborn element e is pushed onto a path stack S_i iff it satisfies three conditions. The second condition is modified as follows:

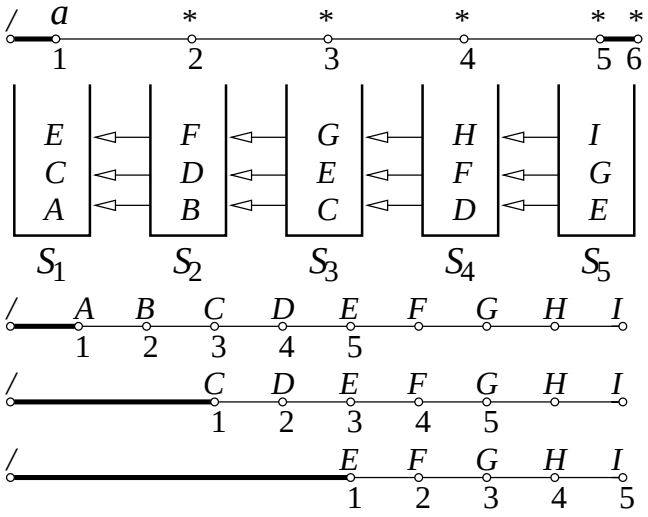

Figure 5: Example 6.1: (a). Query Q, (b). Path Stacks, and (c). Embeddings in the Current Path

> As before, S_{i-1} must be nonempty; i.e., $i \leq nempty + 1$. In addition, if $axis(L_i) = /$, then $top(S_{i-1})$ must be the parent of e.

Also, $R_i(e)$ will have one extra field $actualMatch$ discussed below.

For an element e in S_i, the set $Paths(i, e)$ of paths that are *possible* matches for $L_1L_2 \cdots L_i$ is redefined to account for / axes. For an example, see Figure 2.

The concept of redundancy is also tightened; i.e., fewer cases of redundancy. In the previous section, for two paths $P, P' \in Paths(i, e)$, if $P' \preceq P$ and P' is an actual match for $L_1L_2 \cdots L_i$, then P is redundant. Now, for P to be redundant, $axis(L_{i+1})$ must be //. So, we allow pushing elements (onto path stacks) above an actual match of length i iff $axis(L_{i+1}) = /$. To recognize actual matches for $L_1L_2 \cdots L_i$ (especially when $axis(L_{i+1}) = /$), we augment the record $R_i(e)$ in a path stack with the boolean field $actualMatch$: $R_i(e).actualMatch = True$ iff there exists a path ending in e in $Paths(i, e)$ that is an actual match for $L_1L_2 \cdots L_i$; inductively, $R_i(e).actualMatch = (*(R_i(e).leftPtr).actualMatch \; and \; R_i(e).predStatus)$. If $R_i(e).actualMatch = True$, then any candidate that could use any of the paths in $Paths(i, e)$ to reach the output can go to the output right away.

The variable $satUpto$ is redefined as the largest integer i that satisfies the following conditions:

- $top(S_i).actualMatch = True$
- $axis(L_{i+1}) = //$.

Because of the second condition, $satUpto$ moves in quantum jumps. As before, pushing an element onto a path stack S_i, for $i \leq satUpto$, is redundant. But there could be elements pushed above elements on an actual match, before $satUpto$ jumped to its current value; such elements are *not* redundant.

Example 6.1. Let $Q = //a/*/*/*/*//*$, where predicates are present but not shown. In Figure 5, we show three different embeddings of $L_1L_2 \cdots L_5$ in the current path to node I (L_6 is irrelevant to our discussion here). Numbers 1 thru 5 are indices of location steps L_1 thru L_5, and upper case letters A thru I are document node ids. Since $axis(L_6) = //$, the contents of path stacks $S_1, S_2, \ldots, S_5$ could be as shown only if $R_5(G).actualMatch = R_5(E).actualMatch = False$. Recall that newborn nodes are considered for insertion in S_i, in decreasing order of i. Suppose that $R_5(G).actualMatch$ is $True$ when G is pushed onto S_5. $CDEFG$ is an actual match for $L_1L_2 \cdots L_5$; $satUpto$ jumps from 0 to 5. Then we would not push G onto S_3, as it is redundant; consequently, we would not push H and I onto S_4 and S_5, respectively. But $R_1(E)$ and $R_2(F)$ were pushed (possibly with $predStatus = Unknown$) before $satUpto$ jumped from 0 to 5. They are not redundant for the following reason: When G closes, $R_5(G)$ is popped. The next new element G' we see could be a sibling of G. G' might fail $nodeTest(L_5)$ or $predicate(L_5)$, and so candidates that are descendants of G' can not use $CDEFG'$ as an actual match, to reach the output. But G' might pass $nodeTest(L_3)$ and get pushed onto S_3; so, some of those candidates could rely on a possible path consisting of EFG' (as match for $L_1L_2L_3$). ∘

Now, consider avoiding redundancy past $S_{satUpto}$ (i.e., in S_i, $satUpto < i < n$), using the procedure $notRedPush(i, R)$. At any time, at most one child node of an open node is open. So, the condition $top(S_i).leftPtr = T_{i-1}$ in the procedure can hold only if $axis(L_i) = //$; so, this procedure needs to be called only in this case.

Now, let us consider the modifications pertaining to candidate stacks. As before, a candidate e in S_n should reach the output iff (and when) the following two conditions are satisfied:

1. e satisfies $predicate(L_n)$.
2. Any one path in $Paths(n-1, *(R_n(e).leftPtr))$ becomes an actual match for $L_1L_2 \cdots L_{n-1}$.

The only difference is in how we test if condition 2) is met, at the time condition 1) is met: $*(R_n(e).leftPtr).actualMatch = True$.

In Section 5, an element e in C_i was represented by the record $R'_i(e) = (label(e), event\#, pathPtr)$. If there is no index $k \geq i$ such that $axis(L_{k+1}) = //$, then there is no change to this record. Else, let $k \geq i$ be the smallest such index. Now, we add the additional field $stackSeq$ to $R'_i(e)$. This field and its use constitute the technically hardest part of this paper. $StackSeq$ is a variable length sequence of some stack indices j, $i \leq j \leq k$; it keeps track of alternate possible paths for e to reach the output. Whenever we process $R'_i(e)$, the top elements of all the stacks whose indices are in its $stackSeq$ field are the same document element.

Example 6.2. Continuing with Example 6.1, suppose that *satUpto* stays at 0, and at some point we have three embeddings of $L_1L_2\cdots L_5$ in the current path to node I as shown in Figure 5. Recall that $axis(L_6) = //$. Since $satUpto = 0$, the $actualMatch$ field must be $False$ for all three records in S_5.

For any element in C_5, its *stackSeq* is (5). Suppose that for the top element e in C_5, $R'_5(e).pathPtr = T_5$ (i.e., $R'_5(e)$ is pointing to $R_5(I)$). If I fails $predicate(L_5)$ then, since $axis(L_6) = //$, e could try the path ending in $R_5(G)$; so, we will pop $R_5(I)$ and set $R'_5(e).pathPtr$ to the new value of T_5, just as in the previous section. Instead, suppose that I satisfies $predicate(L_5)$. e will stay in C_5 until I closes, and then e would be moved to C_4, with its $pathPtr$ pointing to $R_4(H)$ and $stackSeq = (4)$. If H fails $predicate(L_4)$, it would be popped from S_4; instead of just changing $R'_4(e).pathPtr$ to the new value of T_4 as done in the previous section, we have to move e to C_5 with $pathPtr$ pointing to $top(S_5) = R_5(G)$, and $stackSeq = (5)$. Instead, suppose that H satisfies $predicate(L_4)$. e will stay in C_4 until H closes, and then e would be moved to C_3, with $pathPtr$ pointing to $R_3(G)$, and $stackSeq = (3)$. Since the top element in S_3 is same as that in S_5, we would append 5 to the *stackSeq* field: $R'_3(e).stackSeq = (3, 5)$; this signifies that e could try an alternate path ending with $R_5(G)$.

Note that G might not stay as the top element in S_5, as for example, if we next push a sibling H' of H onto S_4, and then push a child I' of H' onto S_5; we still do not need to store a pointer to $R_5(G)$ in $R'_3(e)$. The reason for this: Next time we want to process $R'_3(e)$, G will again be the current element, and be the top element in S_3 and S_5 (Fact 5.5).

When G is the current element, consider the following possibilities after a SAX event:

- G satisfies $predicate(L_5)$ but does not yet fail $predicate(L_3)$. Set $R_5(G).predStatus = True$; if G also satisfies $predicate(L_3)$ then set $R_3(G).predStatus = True$. If $R_5(G).actualMatch$ becomes $True$, increment *satUpto* to 5, pop and output $R'_3(e)$, and pop $R_3(G)$ as it is redundant. Else if $R_3(G).actualMatch$ becomes $True$, then pop and output $R'_3(e)$.
- G satisfies $predicate(L_5)$ but fails $predicate(L_3)$. Set $R_5(G).predStatus = True$ and pop $R_3(G)$. Pop e from C_3 and push it onto C_5, with $stackSeq = (5)$. If $R_5(G).actualMatch$ becomes $True$, increment *satUpto* to 5, pop and output e.
- G fails $predicate(L_5)$ but does not yet fail $predicate(L_3)$. Pop $R_5(G)$; if $topBunch(C_5)$ was pointing to it, make it point to $R_5(E)$. Delete 5 from $R'_3(e).stackSeq$; it becomes (3). If G passes $predicate(L_3)$, set $R_3(G).predStatus = True$; if $R_3(G).actualMatch$ becomes $True$, then pop and output e.
- G fails $predicate(L_5)$ and $predicate(L_3)$. Pop $R_3(G)$ and $R_5(G)$. If $topBunch(C_5)$ was pointing to $R_5(G)$, then make it point to $R_5(E)$. Move e from C_3 to C_5, with $pathPtr$ pointing to $R_5(E)$ and $stackSeq = (5)$.

Now, consider the situation when G closes with at least one of $predicate(L_5)$ and $predicate(L_3)$ being $True$. We have the following cases:

- Both $predicate(L_5)$ and $predicate(L_3)$ are $True$. e is moved from C_3 to C_2, with $pathPtr$ pointing to $R_2(F)$ and $stackSeq = (2, 4)$ (i.e., pointing to $R_2(F)$ and $R_4(F)$).
- Only $predicate(L_5)$ is $True$. e is moved from C_5 to C_4, with $pathPtr$ pointing to $R_4(F)$ and $stackSeq = (4)$.
- Only $predicate(L_3)$ is $True$. e is moved from C_3 to C_2, with $pathPtr$ pointing to $R_2(F)$ and $stackSeq = (2)$.

In all the three cases above, if the new $top(S_5)$ is F (in our example, it is E), then 5 would be appended to $R'(e).stackSeq$. ∘

As in the previous section, we group all elements in C_i that have the same value for $pathPtr$ **and** $stackSeq$ (they must be contiguous in C_i) into a *bunch* with a single $pathPtr$ and $stackSeq$. All elements in a bunch are relying on the same set of paths to reach the output. If any one of these paths becomes an actual match, then the entire bunch is output.

Because of the second condition for bunching elements in C_i (namely, they must have the same $stackSeq$), there could be several contiguous bunches at the top of C_i with $pathPtr = T_i$. When there is a change in $top(S_i)$ due to a SAX event, each of these bunches must be handled separately, based on its $stackSeq$; bunches that end up with the same $pathPtr$ and $stackSeq$, after the SAX event, must be combined.

Now, we give a general description of how to handle bunches and their $stackSeq$. Each $stackSeq$ is kept sorted in increasing order. For any bunch in C_k where $axis(L_{k+1}) = //$, its $stackSeq$ is (re)initialized to (k). Consider a bunch in C_i where $axis(L_{i+1}) = /$, and k is the smallest integer greater than i such that $axis(L_{k+1}) = //$. Its $stackSeq$ is an increasing sequence of some integers j, $i \le j \le k$; also i is the first (smallest) element of this sequence. Whenever we are processing this bunch, $top(S_j)$, for all $j \in stackSeq$, correspond to the same document element, namely the current element, say X. If $top(S_j).actualMatch = True$ for any $j \in stackSeq$, then the bunch is popped and output; also if $k \in stackSeq$ and $top(S_k).actualMatch = True$, then increment *satUpto* to k and pop X from S_j for all $j < k$, as they are redundant. Else, when X closes, the bunch is updated:

- For each $j \in stackSeq$: If X satisfied $predicate(L_j)$, replace j by $j - 1$; else delete j.

- After the previous step: If *stackSeq* is empty, set $stackSeq = (k)$; if S_k is empty, discard bunch. Else (i.e., *stackSeq* is not empty) if $parent(X)$ is in $top(S_k)$, append k to *stackSeq*.
- Move bunch to candidate stack C_m, where m is the first element of the new *stackSeq*.

Resource requirements of the modified algorithm: The only change, compared to the analysis in the previous section, pertains to candidate stacks, for maintaining the *stackSeq*s. The length of any *stackSeq* is bounded by the maximum number of location steps (all with `child` axis) in between two successive location steps with `descendant` axis. Using the trivial upper bound of n for this, the space for candidate stacks is $O(cn)$, and the time spent on candidate stacks is $O((d + n^2)|D|)$. So, the overall memory space and runtime required, in the worst case, are $O(d|Q| + cn)$ and $O((|Q| + d + n^2)|D|)$, respectively.

Theorem 6.1. The algorithm described in this section correctly evaluates an XPath query Q on a streaming XML document D. The algorithm uses $O(d|Q| + cn)$ space and $O((|Q| + d + n^2)|D|)$ time in the worst case.

7 Conclusions

We presented an efficient algorithm for evaluating an XPath query Q (involving only `child` and `descendant` axes) on a streaming XML document D. Several previously known algorithms for this problem use exponential space and time, in the worst case. Our algorithm uses polynomial space and time. Also, our algorithm is runtime competitive with the in-memory algorithms, while using much less memory space.

We presented a novel predicate checker that would be of use in other XML applications involving XPath. It can be extended to predicates that also involve the `preceding` and `preceding-sibling` axes. Our XPath evaluation algorithm can also be extended to queries containing such predicates, without increasing the memory space or runtime.

References

[1] M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information, VLDB 2000, pp. 53–64.

[2] C. Barton, P. Charles, D. Goyal, M. Ragavachari, M. Fontoura and V. Josifovski. Streaming XPath Processing with Forward and Backward Axes, *Proc. ICDE*, 2003.

[3] Z. Bar-Yossef, M. Fontoura and V. Josifovski. On the Memory Requirements of XPath Evaluation Over XML Streams, PODS 2004, pp. 177–188.

[4] Z. Bar-Yossef, M. Fontoura and V. Josifovski. Buffering in Query Evaluation Over XML Streams, PODS 2005, pp. 216–227.

[5] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie and J. Simeon. XML Path Language (XPath) 2.0, `www.w3.org/TR/xpath20`.

[6] N. Bruno, N. Koudas and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching, SIGMOD 2002, pp. 310–321.

[7] C. Chan, P. Felber, M. Garofalakis and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions, ICDE 2002.

[8] Y. Diao, P. Fischer, M. Franklin and R. To. YFilter: Efficient and Scalable Filtering of XML Documents, ICDE 2002.

[9] G. Gottlob, C. Koch and R. Pichler. Efficient Algorithms for Processing XPath Queries, *Proc. Intl. Conf. Very Large Data Bases* (VLDB), 2002.

[10] T. J. Green, G. Miklau, M. Onizuka and D. Suciu. Processing XML Streams with Deterministic Automata, ICDT 2003, pp. 173–189.

[11] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates, SIGMOD 2003.

[12] V. Josifovski, M. Fontoura and A. Barta. Querying XML Streams, *VLDB J*, 2004.

[13] D. Olteanu, T. Furche, F. Bry. Evaluating Complex Queries against XML Streams with Polynomial Combined Complexity, BNCOD 2004.

[14] D. Olteanu, T. Kiesling and F. Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams, ICDE 2003.

[15] D. Olteanu, H. Meuss, T. Furche and F. Bry. XPath: Looking Forward, *Proc. Workshop on XML–Based Data Management*, 2002, pp. 109–127.

[16] F. Peng and S. Chawathe. XPath Queries on Streaming Data, SIGMOD 2003.

[17] P. Ramanan. Covering Indexes for XML Queries: Bisimulation − Simulation = Negation, VLDB 2003, pp. 165–176.

[18] P. Ramanan. Evaluating an XPath Query on a Streaming XML Document, Tech. Report WSU CS 03-4, CS Dept, Wichita State Univ, Nov 2003.

[19] P. Ramanan. Holistic Join for Generalized Tree Patterns, submitted.