Worst-Case Optimal Algorithm for XPath Evaluation over XML Streams

Prakash Ramanan
EECS Department
Wichita State University
Wichita, KS 67260–0083
ramanan@cs.wichita.edu

Abstract

We consider the XPath evaluation problem: Evaluate an XPath query $Q$ on a streaming XML document $D$; i.e., determine the set $Q(D)$ of document elements selected by $Q$. We mainly consider Conjunctive XPath (CXPath) queries (a subclass of XPath 1.0 [12]) that involve only the child and descendant axes. Previously known in-memory algorithms for this problem use $O(|D|)$ space and $O(|Q||D|)$ time. Several previously known algorithms for the streaming version use $\Omega(d^n)$ space and $\Omega(d^n|D|)$ time in the worst case; $d$ denotes the depth of $D$, and $n$ denotes the number of location steps in $Q$. Their exponential space requirement could well exceed the $O(|D|)$ space used by the in-memory algorithms. We present an efficient algorithm that uses $O(d|Q| + nc)$ space and $O((|Q| + dn)|D|)$ time in the worst case; $c$ denotes the maximum number of elements of $D$ that can be candidates for output, at any one instant. For some worst case $Q$ and $D$, the memory space used by our algorithm matches our lower bound proved in a different paper; so, our algorithm uses optimal memory space in the worst case.

keywords. XML, XPath, query evaluation, stream processing

1 Introduction

We consider the XPath evaluation problem: Evaluate an XPath query $Q$ on a streaming XML document $D$; i.e., determine the set $Q(D)$ of document elements selected by $Q$. We mainly consider Conjunctive XPath (CXPath) queries (a subclass of XPath 1.0 [12]) that involve only the child and descendant axes. We present an efficient algorithm that uses $O(d|Q| + nc)$ space and $O((|Q| + dn)|D|)$ time in the worst case; $d$ denotes the depth of $D$, and $n$ denotes the number of location steps in $Q$. $c$ denotes the maximum number of elements of $D$ that can be candidates for output, at any one instant. As we point out later, our algorithm can be extended to queries whose predicates involve or and not, some XPath library functions such as aggregation and position, and the preceding and preceding-sibling axes, without increasing the memory space or runtime.
In line with most of the theoretical papers in this area, the memory space bounds that we quote in this paper do not include the space used to buffer the contents of the candidate elements. This latter space requirement is discussed separately, in Section 3.

There are many previously-known results concerning the XPath evaluation problem. Unless mentioned otherwise, the following results pertain to CXPath queries. First, consider results pertaining to non-streaming $D$. Gottlob et al. [15] and Ramanan [27] presented in-memory algorithms that use $O(|D|)$ space and $O(|Q||D|)$ time. Gottlob et al. [16] proved some abstract complexity results for different fragments of XPath.

From now onwards, consider results pertaining to streaming $D$. If $Q$ does not have predicates, the evaluation problem is easy: An element $e$ 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. [9] can be adapted to solve the evaluation problem. The resulting algorithm uses $O(dn + c)$ space and $O(n|D|)$ time.

When $Q$ has predicates, the evaluation problem is more complicated. At the time the start tag of an element $e$ 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 depends on the yet-to-be-seen part of $D$. For example, for the query $/a[b]/c$, the $c$ children of an $a$ element will be candidates until a $b$ child of the $a$ element is seen (when the $c$ children qualify for output), or until the $a$ element closes without any $b$ children (when the $c$ children can be discarded). So, an algorithm for this problem would typically store, at any instant, the candidates $e$ as well as some information about their ancestors $f$ that could enable the candidates to qualify for output.

There are several previously known algorithms for this problem. The XSQ algorithm of Peng and Chawathe [26], and the SPEX system of Olteanu et al. [25] require $\Omega(d^n)$ space and $\Omega(d^n|D|)$ time,
in the worst case. The quantity \(d^n\) represents the number of different paths in \(D\) that a candidate \(e\) could take to qualify for output; all these paths are embedded on the path from the root of \(D\) to \(e\). This space could well exceed the \(O(|D|)\) space used by the in-memory algorithms [15, 27] mentioned above. Also, the XSQ algorithm uses an additional \(\Theta(2^n|Q|)\) space and time, in the best case, for a pushDown transducer.

The recursion depth of \(D\), denoted below by \(r\), is the maximum number of elements with the same tagname on any root-to-leaf path in \(D\); note that \(1 \leq r \leq d\). We say that \(D\) is nonrecursive with respect to \(Q\) (or \((Q,D)\) is nonrecursive) if the following holds: For any path (starting at root\((Q)\)) in \(Q\), and any path (starting at root\((D)\)) in \(D\), there is at most one embedding of the former in the latter. \((Q,D)\) is recursive if it is not nonrecursive.

Bar-Yossef et al. [7] presented an algorithm for nonrecursive \((Q,D)\) that uses \(O(|Q| \log |D| + c)\) space and \(O(|Q||D|)\) time; they also proved an \(\Omega(c)\) space lower bound, for each instance \((Q,D)\). Josifovski et al. [21] outlined an algorithm for the general case, but no explicit complexity bounds were presented. Chen et al. [11] presented an algorithm that uses \(O(|Q||D|(|Q|+dc))\) time; no memory space bound was given. Olteanu et al. [24] presented an algorithm that they claim uses \(O(d^2|Q| + c)\) space and \(O(d|Q||D|)\) time. Recently, Gou et al. [17] presented an algorithm that they claim uses \(O(r|Q| + c)\) space and \(O(|Q||D|)\) time. The latter two algorithms use only \(O(c)\) space for storing information that might qualify the candidates for output. We [30] proved that any algorithm must use \(\Omega(nc)\) space to store such information, for some worst case \(Q\) and \(D\). So, the algorithms in [24, 17] are incorrect. Our algorithm presented in this paper is from [28]; it is among the first correct algorithms known for the streaming version that also have a polynomial bound on the memory space and runtime.

When all the location steps in \(Q\) have the descendant axis (outside the predicates), our algorithm uses \(O(d|Q| + c)\) space and \(O(|Q||D|)\) time. When \(Q\) has a mix of child and descendant axis steps, our algorithm uses \(O(d|Q| + nc)\) space and \(O((|Q| + dn)|D|)\) time, in the worst case. For some worst case \(Q\) and \(D\), this space requirement matches our lower bound in [30]; so, our algorithm uses optimal memory space in the worst case.

For the general case, since \(n \leq depth(Q)\), our worst case runtime of \(O((|Q| + dn)|D|)\) is very
competitive with the \(O(|Q||D|)\) runtime of the in-memory algorithms [15, 27]; also, our algorithm uses much less memory space.

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, 10, 13, 18, 19] presented algorithms for various versions of this problem; of these, the XPush machine [19] is the only algorithm that works for general CXPath queries. All these algorithms require space and time exponential in \(|Q|\).

Consider the filtering problem, for a single XPath query. Bar-Yossef et al. [6] presented an algorithm that uses \(O(r|Q|(\log |Q| + \log d))\) bits of space and \(O(r|Q||D|)\) time. They also presented an \(\Omega(r + \log d)\) space lower bound, for each instance \((Q, D)\). We present an algorithm (Section 5) that uses \(O(d|Q|)\) bits of space and \(O(|Q||D|)\) time; as per our lower bound in [30], this algorithm uses optimal space for some worst case queries. Recently, Gou et al. [17] presented a similar algorithm that uses \(O(r|Q|\log d)\) bits of space and \(O(|Q||D|)\) time.

Barton et al. [5] presented an algorithm for evaluating XPath queries that also have backward axes (e.g., parent and ancestor). Florescu et al. [14], Josifovski et al. [21], Koch et al. [22] and Ludascher et al. [23] presented systems for evaluating different subclasses of XQuery queries on streaming XML documents.

Bar-Yossef et al. [6, 7] presented space lower bounds for the query filtering and evaluation problems, respectively, for nonrecursive \((Q, D)\); Ramanan [30] presented lower bounds for recursive \((Q, D)\).

There have been several results concerning algorithms for various problems in the data stream model.

**Outline of the paper.** In Section 2, we define our fragment of XPath, called Conjunctive XPath, and discuss query evaluation. In Section 3, we describe SAX events, and discuss the buffering of elements for output. In Sections 4, we give a brief outline of our algorithm, and describe one of the three components of our algorithm, namely, path stacks. In Section 5, we describe another component, namely, the predicate checker. In Section 6, we describe our algorithm when all the location steps in $Q$ have the descendant axis. In Section 7, we present the modifications to our algorithm, when some location steps in $Q$ have the child axis. In Section 8, we show how to extend our algorithm to queries whose predicates involve or, not, some XPath library functions, and the preceding and preceding-sibling axes. In Section 9, we present our conclusions.

## 2 Class of Queries and Query Evaluation

In this section, we define a fragment of XPath, called Conjunctive XPath. We also define embeddings and the output of a query on an XML document.

We follow the *XPath* 1.0 data model [12]. An XML document $D$ is represented as a tree. Each element, attribute or text content is represented by a node. For an element or attribute node $x \in D$, $\tau(x)$ denotes its tagname. $\text{Root}(D)$ is a special node that does not correspond to any element in $D$; it is the parent of the node that corresponds to the root element of $D$; $\tau(\text{root}(D)) = /$.

We consider *XPath* 1.0 [12] queries that involve only the child and descendant axes. Let *Conjunctive XPath* *(CXPath)* be the subclass of XPath 1.0, consisting of queries of the form $L_1L_2\ldots L_n$. Each location step $L_i$ is of the form $<axis> <node_test> <predicates>$. Axis is either / or //, corresponding to child and descendant axis, respectively. In node tests, attributes are treated similar to subelements. Each predicate is either an and of predicates, a relative query, or a comparison between the value of a node matching a relative query and a string value. This class of queries is defined by the following grammar:

$$
<query> ::= <loc_step> | <loc_step> <query>
$$
\textbf{Definition 2.1.} \([\text{trunk}_i(Q), \text{trunk}(Q) \text{ and } \text{opv}(Q)]\) For 1 \(\leq i \leq n\), \(\text{trunk}_i(Q)\) denotes the path \((v_1, v_2, \ldots, v_i)\). \(\text{Trunk}(Q)\) denotes \(\text{trunk}_n(Q) = (v_1, v_2, \ldots, v_n)\). \(v_n\) is called the output vertex of \(Q\), and is denoted by \(\text{opv}(Q)\). \(\Diamond\)

\(\text{opv}(Q)\) is marked by a \$ sign in the figures.

\textbf{Example 2.1.} Consider the \textit{CXPath} query \(Q = //a[./b\text{ and } ./c]/*[./d\text{ and } ./b > 2]\). It consists of the two location steps \(L_1 = //a[./b\text{ and } ./c]\), and \(L_2 = /*[./d\text{ and } ./b > 2]\). \(\text{Predicate}(L_1) = [./b\text{ and } ./c]\), and \(\text{predicate}(L_2) = [./d\text{ and } ./b > 2]\). Figure 1a shows \(\text{tree}(Q); \text{trunk}(Q) = (v_1, v_2)\) and \(\text{opv}(Q) = v_2). \(\Diamond\)
Figure 1: Examples 2.1 and 2.2: (a) Tree \( T \) for \( Q \), (b) \( T_1 = \text{tree}(\text{predicate}(L_1)) \) and \( T_2 = \text{tree}(\text{predicate}(L_2)) \)

In general, \(|\text{tree}(Q)|\) is linear in \(|Q|\). From now onwards, we will not distinguish between \( Q \) and \( \text{tree}(Q) \). To minimize confusion, we will use the terms vertices and arcs while referring to the components of \( Q \); nodes and edges refer to the corresponding components of \( D \). For a vertex \( u \in Q \), let \( Q_u \) denote the subtree of \( Q \) that is rooted at \( u \). For a node \( e \in D \), let \( D_e \) denote the subtree of \( D \) rooted at \( e \).

**Definition 2.2.** [Embedding] An embedding \( \Gamma \) of \( Q_u \) in \( D_e \) is a mapping from the vertices of \( Q_u \) to the nodes of \( D_e \), that satisfies the following conditions:

- **Preserve vertex tagnames:** For each vertex \( v \) in \( Q_u \):
  
  - If \( \tau(v) = / \), then \( \Gamma(v) = \text{root}(D) \). In this case, \( v = u = \text{root}(Q) \) and \( e = \text{root}(D) \).
  
  - If \( \tau(v) \in \Sigma \), then \( \tau(\Gamma(v)) = \tau(v) \).

  In addition, \( \Gamma(v) \) satisfies any “<relOp> const” condition associated with \( v \) (e.g. “> 2” at vertex 6 in Figure 1a).

- **Preserve arc types:**
  
  - For each \( c \text{-arc} (v, v') \) in \( Q_u \): \( \Gamma(v') \) is a child of \( \Gamma(v) \) in \( D \).
  
  - For each \( d \text{-arc} (v, v') \) in \( Q_u \): \( \Gamma(v') \) is a descendant of \( \Gamma(v) \) in \( D \).

**Definition 2.3.** [Output of \( Q \) on \( D \)] The output of \( Q \) on \( D \) is

\[
Q(D) = \{ \Gamma(\text{opv}(Q)) \mid \Gamma \text{ is an embedding of } Q \text{ in } D \}.
\]

For a node \( e \in D \), let \( \text{path}(e) \) denote the path from \( \text{root}(D) \) to \( e \). An embedding of \( \text{trunk}_1(Q) \) in \( \text{path}(e) \) is an embedding as defined above, but with its domain being \( \text{trunk}_1(Q) \) and its target set being...
<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CXPath</strong></td>
<td>Conjunctive XPath with only / and // axes</td>
</tr>
<tr>
<td>$L_i$ ($1 \leq i \leq n$)</td>
<td>$i^{th}$ location step in $Q$</td>
</tr>
<tr>
<td>$tree(Q)$</td>
<td>tree representing $Q$</td>
</tr>
<tr>
<td>$v_i$</td>
<td>vertex in $tree(Q)$ that corresponds to nodeTest($L_i$)</td>
</tr>
<tr>
<td>$trunk_i(Q)$</td>
<td>path $(v_1, v_2, \ldots, v_i)$ in $tree(Q)$</td>
</tr>
<tr>
<td>$opv(Q) = v_n$</td>
<td>output vertex of $Q = \text{last vertex in } trunk(Q)$</td>
</tr>
<tr>
<td>$\tau()$</td>
<td>tagname associated with a vertex in $tree(Q)$ or node in $D$</td>
</tr>
<tr>
<td>$Q_u$</td>
<td>subtree of $Q$ rooted at vertex $u$</td>
</tr>
<tr>
<td>$D_e$</td>
<td>document subtree of $D$ rooted at node $e$</td>
</tr>
<tr>
<td>$Q(D)$</td>
<td>output of $Q$ on document $D$</td>
</tr>
<tr>
<td>$path(e)$</td>
<td>path from root($D$) to node $e$</td>
</tr>
<tr>
<td>$embeddings_i(e)$</td>
<td>embeddings of $trunk_i(Q)$ in $path(e)$, with $\Gamma(v_i) = e$</td>
</tr>
<tr>
<td>$embeddings_i(\leq e)$</td>
<td>$\cup_{e'}embeddings_i(e')$, union over $e$ and its ancestors in $D$</td>
</tr>
<tr>
<td>$embeddings_i(all)$</td>
<td>$\cup_{e \in D}embeddings_i(e)$, union over $e$ and its ancestors in $D$</td>
</tr>
<tr>
<td>$T_i$</td>
<td>$tree(predicate(L_i))$</td>
</tr>
</tbody>
</table>

Table 2: Notations from Section 2

path($e$).

**Definition 2.4.** [embeddings$_i(e| \leq e| all)$] For $1 \leq i \leq n$, and node $e \in D$, embedings$_i(e)$ denotes

$$\{\Gamma | \Gamma \text{ is an embedding of } trunk_i(Q) \text{ in } path(e), \text{ with } \Gamma(v_i) = e\}.$$  

$Embeddings_i(\leq e)$ denotes $\cup_{e'}embeddings_i(e')$, where the union is over $e$ and all its ancestors in $D$.  

$Embeddings_i(all)$ is the set of all embeddings of $trunk_i(Q)$ in $D$; i.e., $embeddings_i(all) = \cup_{e \in D}embeddings_i(e)$. ◦

**Definition 2.5.** [$T_i = tree(predicate(L_i))$] For $1 \leq i \leq n$, $T_i = tree(predicate(L_i))$ is the subtree obtained from $Q_{v_i}$ as follows: Change $\tau(v_i)$ to $*$; further, if $i < n$, delete the arc $(v_i, v_{i+1})$ and the subtree $Q_{v_{i+1}}$. ◦

**Example 2.2.** Figure 1b shows $T_1 = tree(predicate(L_1))$ and $T_2 = tree(predicate(L_2))$, for the query $Q$ in Example 2.1. ◦

**Definition 2.6.** [Node Satisfying a Predicate] A node $e \in D$ satisfies $predicate(L_i)$ if there exists an embedding $\Gamma'$ of $T_i = tree(predicate(L_i))$ in $D$ such that $\Gamma'(\text{root}(T_i)) = e$. ◦
An embedding $\Gamma$ of $\text{trunk}(Q)$ in $D$ can be extended to an embedding of $Q$ in $D$ iff, for each $i$ ($1 \leq i \leq n$), $\Gamma(v_i)$ satisfies $\text{predicate}(L_i)$. So, we have the following.

**Fact 2.1.** $Q(D) = \{ e \in D \mid \exists \Gamma \in \text{embeddings}_n(e) \text{ such that } \Gamma(v_i) \text{ satisfies } \text{predicate}(L_i) \text{ for } 1 \leq i \leq n \}$. ♦

### 3 SAX Events and the Element Buffers

In this section, we first describe SAX events, and then discuss the buffering of elements (i.e., their contents) that might need to be output.

We assume that the input XML document $D$ is presented as a stream of SAX events [8] of five types:

- `startDocument()`
- `startElement(a)`
- `text(s)`
- `endElement(a)`
- `endDocument()`

We treat attributes similarly to elements; so, the tagname $a$ above might be an element or an attribute tagname. $s$ is a data (string) value. For example, the document

```
<a b="101"><c>201</c></a>
```

leads to the following sequence of events:

```
startDocument(), startElement(a), startElement(@b),
text("101"), endElement(@b), startElement(c), text("201"),
endElement(c), endElement(a), endDocument().
```

These events can be numbered, starting at one. So, the above sequence consists of events one through ten.

**Definition 3.1.** [Open, Closed and Current Nodes] An element node (in a document) opens when its `startElement` is seen in the input stream; it stays open until its `endElement` is seen, at which point it becomes closed. A node is current if it is open, but none of its descendants is open. ♦

Note that a node becomes current when it opens, and stays current until one of its children opens; it becomes current again when that child closes. The currentness of a node is not affected by `text` events.

**Definition 3.2.** [Current Path] The current path is the path from $\text{root}(D)$ to the current node; it contains all (and only) the open nodes. ♦
Now, let us consider the form of output required from an algorithm for the XPath evaluation problem. As per the XPath 1.0 specification [12], an algorithm should output the elements in $Q(D)$ in document order. But the following example shows that, in the stream model, the order in which these elements are found to belong to the output (based on the document prefix seen so far) might not match the document order.

**Example 3.1.** Consider the result of the query $Q = //a [b]$ on the three XML documents in Figure 2. For all three documents, the output consists of the elements represented by nodes 1–4. As per the XPath 1.0 specification [12], these four elements 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 elements are found to belong to the output might not match the document order. For Figures 2a, 2b and 2c, the elements would be found to belong to the output in the order 1234, 4321 and 1342, respectively.

To output the elements in document order, we have to buffer each output element, until none of its preceding elements or ancestors is a candidate. This could result in the buffering of many elements that have already been found to belong to the output.

In general, an algorithm that outputs the element contents in document order must maintain two buffers:

1. A *candidate buffer* that maintains the contents of candidate elements. For an element $e$ in this buffer, there are two possibilities:
• We find that $e \notin Q(D)$. Then $e$ is discarded.
• We find that $e \in Q(D)$. Then $e$ is moved to the output buffer.

2. An output buffer that maintains the contents of those elements that are known to be in $Q(D)$ but have not yet been output. An element $e$ from this buffer is output when the following holds:

• The complete content of $e$ is known (i.e., $e$ is closed).
• No predecessor or ancestor of $e$ is either a candidate, or is in the output buffer.

Let $C$ denote the maximum (over all time instants) space used by both these buffers. Any algorithm for the XPath evaluation problem must use this $O(C)$ space for buffering element content. In our memory bounds, we do not include this space.

In the rest of this paper, when we say “output” an element $e$, we mean the following:

• We have determined that $e \in Q(D)$.
• Move the element content of $e$ (seen so far) from the candidate buffer to the output buffer.

4 Algorithm Outline and Path Stacks

Our algorithm consists of three main components:

• Path stacks (described in this section) to compactly represent all embeddings of $\text{trunk}(Q)$ in the current path in $D$. This is a more elaborate version of the path stacks of [9].
• The predicate checker (Section 5) determines which predicates $\text{predicate}(L_i)$ ($1 \leq i \leq n$) in $Q$ are satisfied/failed at each node in the path stacks.
• Candidate stacks (Sections 6 and 7) to maintain the candidates and move them along to the output or trash.

In this section, we describe our path stacks. There are $n$ path stacks: For $1 \leq i \leq n$, path stack $S_i$ corresponds to location step $L_i$. $S_i$ contains those open nodes $e$ for which $\text{embeddings}_i(e) \neq \emptyset$. 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. $S_0$ denotes the
**Table 3: Notations from Section 4**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_i$ ($1 \leq i \leq n$)</td>
<td>path stack associated with step $L_i$</td>
</tr>
<tr>
<td>$top(S_i)$</td>
<td>top record in $S_i$</td>
</tr>
<tr>
<td>$t_i$</td>
<td>pointer to $top(S_i)$</td>
</tr>
<tr>
<td>$nempty$</td>
<td>only $S_1, S_2, \ldots, S_{nempty}$ are nonempty</td>
</tr>
<tr>
<td>$R_i(e)$</td>
<td>record for an open node $e$ in $S_i$</td>
</tr>
<tr>
<td>$R_i(e) = (\tau(e), event#, predStatus, leftPtr)$</td>
<td>all embeddings of trunk$_i(Q)$ in the current path</td>
</tr>
<tr>
<td>$embeddings_i()$</td>
<td>all paths of length $i$ ending in $e \in S_i$, represented by $S_1 \ldots S_i$</td>
</tr>
<tr>
<td>$Paths_i(e)$</td>
<td>$Paths_i(\leq e)$</td>
</tr>
<tr>
<td>$Paths_i(\leq \text{top}(S_i))$</td>
<td>all paths of length $i$ repstd by $S_1 \ldots S_i$</td>
</tr>
</tbody>
</table>

*imaginary* path stack that corresponds to root($Q$); it contains only root($D$); recall that $\tau(root(Q)) = \tau(root(D)) = /$.

Let $top(S_i)$ denote the top record in $S_i$, and let $t_i$ denote the pointer to $top(S_i)$. At any instant, the variable $nempty$ keeps track of the last nonempty path stack: $S_1, S_2, \ldots, S_{nempty}$ are all nonempty.

A document node $e$ is pushed into a path stack only when the most recent SAX event is the startElement event for $e$. $e$ is pushed into $S_i$ iff it satisfies three conditions:

1. $e$ passes $\text{nodeTest}(L_i)$.

2. Consider $\text{axis}(L_i)$.

   - $\text{Axis}(L_i) = \text{descendant}$. Then, $e$ needs to have 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., $i \leq nempty + 1$.

   - $\text{Axis}(L_i) = \text{child}$. Then, $e$’s parent needs to be in $S_{i-1}$. If we proceed in decreasing order of $i$, then this condition is satisfied iff $i \leq nempty + 1$, and $top(S_{i-1})$ is the parent of $e$.

3. $e$ is not redundant in $S_i$. This is an optimization measure that will be explained in Sections 6 and 7; for now, ignore this requirement.

When we push $e$ into $S_i$, we actually push the record

$$R_i(e) = (\tau(e), event\#, predStatus, leftPtr).$$
**Event#** is the SAX event number for the **startElement** event for *e*. **PredStatus** is either **True** or **Unknown**, indicating whether node *e* has already-satisfied/not-yet-satisfied **predicate**(*L*), respectively; *R*(*e*) is popped from *S* when *e* fails **predicate**(*L*), or when *e* closes. **LeftPtr** is the value of *t* when *R*(*e*) is pushed into *S*, and after that its value stays constant; it points to the top most element of *S* that is a proper ancestor of *e*. In the rest of this paper, when we say “element *e* in *S*”, we actually mean the record *R*(*e*).

Recall the definitions of **embeddings**(*i*)(*e*) and **embeddings**(all), from Section 2. Now, we have the following.

**Definition 4.1.** [**embeddings**(i)] For 1 ≤ *i* ≤ *n*, **embeddings**(*i*)(e) denotes all embeddings of **trunk**(i)(*Q*) in the current path in *D*; i.e., **embeddings**(*i*)(e) = ∪ e **embeddings**(e), where the union is over all the nodes *e* in the current path in *D*.

For 1 ≤ *i* ≤ *n*, (*S*1, *S*2, . . . , *S*) together maintain a compact representation of a set of paths Paths(*i*) = ∪ e Paths(*i*)(e). Each path *P* ∈ Paths(*i*)(e) consists of a sequence *P*(1)*P*(2) · · · *P*(i) of *i* nodes; it is a subsequence of the nodes on the current path. *P*(i) = *e*; for 1 ≤ *j* < *i*, *P*(j) is specified in decreasing order of *j*, as follows. Suppose that *P*(*j* + 1) = *e*; choices for *P*(j) depend on axis(*L*). If axis(*L*) = child, then *P*(j) is the element *f* in *S* where *R*(*j* + 1)(c).leftPtr points to; if axis(*L*) = descendant, then *P*(j) is either *f* or one of the elements below *f* in *S*. We let Paths(*i*)(≤ *e*) = ∪ e Paths(*i*)(e), where the union is over *e* and all the nodes below it in *S*.

**Example 4.1.** Let *Q* = //a//b//c//d; see Figure 3a. Consider the current path *a*c0b0a2 b1c1d1b2c2d2 (Figure 3c); we have used subscripts to distinguish between different nodes with the same tagname. The path stacks are shown in Figure 3b; Paths(*i*) is shown in Figure 3d. Note that, since axis(*L*) = descendant, a path in Paths(*i*)(d2) could contain either *c*2 or *c*1; but, since axis(*L*) = child, *c*2 can only be used with *b*2 (not with *b*0 or *b*1).

If we change axis(*L*) to descendant, the contents of the path stacks remain the same (for this example current path). But Paths(*i*) now consists of the six paths shown in Figure 3d, and the five additional ones *a*2b1c2d2, *a*1b1c2d2, *a*1b0c2d2, *a*1b0c1d2 and *a*1b0c1d1.
Note that a path \( P \in \text{Paths}_i(e) \) represents an embedding \( \Gamma \in \text{embeddings}_i(e) \); for \( 1 \leq j \leq i \), \( \Gamma(v_j) = P(j) \). From now onwards, we will not distinguish between \( P \) and the embedding it represents.

We have the following.

**Fact 4.1.** Suppose that we ignore \( \text{predStatus} \), and item 3) above, among the conditions for pushing an element into \( S_i \). Then, for \( 1 \leq i < n \) and \( e \in S_i \), \( \text{Paths}_i(e) = \text{embeddings}_i(e) \).

Note that at any particular time instant, the paths in \( \text{Paths}_i(e) \) only contain open nodes. So, no such path represents any embedding in \( \text{embeddings}_i(\text{all}) - \text{embeddings}_i() \).

Now, we give an informal description of our algorithm, based on Example 4.1. Suppose that in the example, the location steps in \( Q \) contain predicates (not shown). Elements \( e \) such that \( \text{embeddings}_n(e) \neq \emptyset \) are candidates for output. For \( 1 \leq i < n \), candidate stack \( C_i \) is associated with path stack \( S_i \); they are used to store the candidates. If candidate \( d_2 \) fails \( \text{predicate}(L_4) \), it is no longer a candidate; it is removed from \( S_4 \) and discarded. Suppose that \( d_2 \) satisfies \( \text{predicate}(L_4) \); when \( d_2 \) closes, it is moved to candidate stack \( C_3 \) and is made to point to \( c_2 \in S_3 \). Consider the following possibilities.

- \( c_2 \) fails \( \text{predicate}(L_3) \). \( c_2 \) is removed from \( S_3 \). Since \( \text{axis}(L_4) = \text{descendant} \), \( d_2 \) is still a candidate, based on an alternate embedding consisting of \( c_1 \); so, \( d_2 \in C_3 \) is made to point to \( c_1 \in S_3 \).

- \( c_2 \) satisfies \( \text{predicate}(L_3) \). When \( c_2 \) closes, \( d_2 \) is moved to \( C_2 \) and is made to point to \( b_2 \in S_2 \).

Consider the following possibilities for \( b_2 \).
- $b_2$ satisfies $\text{predicate}(L_2)$. When $b_2$ closes, $d_2$ is moved to $C_1$ and made to point to $a_2 \in S_1$.

- $b_2$ fails $\text{predicate}(L_2)$. Since axis($L_3$) = \text{child}, $d_2$ should be moved to $C_3$ and made to point to $c_1 \in S_3$; $c_1$ could provide alternate embeddings for $d_2$ to qualify for output. This complication in the presence of \text{child} axes is handled in Section 7. 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 \text{child} axes steps, we need to store additional information with each candidate to allow this backtracking. The easy case is when all the axes in $Q$ (outside the predicates) are \text{descendant} axes; this case is handled in Section 6.

5 The Predicate Checker

The predicate checker determines which predicates $\text{predicate}(L_i)$ ($1 \leq i \leq n$) in $Q$ are satisfied/failed at each node in the path stacks. It uses only $O(d|Q|)$ bits of space and $O(|Q||D|)$ time. It is essentially the same as the bottom-up transducer in [29].

As we saw in Section 2, each predicate $\text{predicate}(L_i)$ is represented by a tree $T_i = \text{tree}(\text{predicate}(L_i))$; $\tau(\text{root}(T_i)) = \ast$. The predicate checker $M$ is constructed from the forest $F = \{T_i \mid 1 \leq i \leq n\}$. 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 $\text{record}(e) = (\tau(e), \text{event}\#_e, \text{self}_e, \text{child}_e, \text{desc}_e)$. $\text{self}_e$, $\text{child}_e$ and $\text{desc}_e$ are three boolean arrays indexed from 1 to $m$. Recall that $D_e$ denotes the document subtree rooted at $e$. For $1 \leq j \leq m$, let $T'_j$ denote the subtree rooted at vertex $j$ in $F$. The arrays are defined as follows: At any instant,

S1 $\text{self}_e[j] = 1$ iff there exists an embedding of $T'_j$ in the part of $D_e$ seen so far, with $j$ mapped to $e$.

C1 $\text{child}_e[j] = 1$ iff there exists an embedding of $T'_j$ in $D$, with $j$ mapped to some child of $e$ seen so far.

D1 $\text{desc}_e[j] = 1$ iff there exists an embedding of $T'_j$ in $D$, with $j$ mapped to some descendant of $e$ seen so far.
**Example 5.1.** For the predicate \( P = [./a[./b and ./c]/]* [./a and ./b] \), \( \text{tree}(P) \) is shown in Figure 4a. Consider the evaluation of \( P \) on the document fragment shown in Figure 4b. The vertices in \( F = \{\text{tree}(P)\} \) are numbered 1 to 7 (i.e., \( m = 7 \)). After document node 3 closes, we have \( \text{self}_1 = 0010010, \text{child}_1 = 0000110 \) and \( \text{desc}_1 = 0001111 \). For example, \( \text{self}_1[3] = 1 \) because there exists an embedding of \( T'_3 \) such that vertex 3 is mapped to node 1. \( \text{child}_1[5] = 1 \) because there exists an embedding of \( T'_5 \) such that vertex 5 is mapped to node 3 (a child of node 1). \( \text{desc}_1[7] = 1 \) because there exists an embedding of \( T'_7 \) such that vertex 7 is mapped to node 5 (a descendant of node 1). \( \diamond \)

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 \( \text{self} = \text{child} = \text{desc} = \vec{0} \).

**startElement:** The record for the current element is pushed into \( S \). The new element \( e \) becomes the new current element, with \( \text{self}_e = \text{child}_e = \text{desc}_e = \vec{0} \). For each leaf vertex \( j \in F \) such that \( \tau(e) \) matches \( \tau(j) \), and there is no “\(<\text{relOp}>\text{const}\)” condition associated with \( j \), set \( \text{self}_e[j] = 1 \). For each \( \text{predicate}(L_i) \) that is empty, output that \( e \) has satisfied \( \text{predicate}(L_i) \).

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

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

- \(child_{e'}[j] = child_{e'}[j] \lor sel_{e}[j]\)
- \(desc_{e'}[j] = desc_{e'}[j] \lor sel_{e}[j] \lor desc_{e}[j]\)
- If \(sel_{e'}[j] = 0\) then set it to 1 if:
  - \(\tau(e')\) 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 \(predicate(L_i)\).

Discard \(record(e)\).

\text{endDocument:} \text{Current element must be ‘/ ’, and S must be empty. Discard \(record(‘/’)\).}

\textbf{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 6 and 7. Recall that \(XPath\) queries do not contain or and not in their predicates. After processing a \texttt{startElement} or \texttt{text} SAX event, \(M\) outputs a set \(L^T\) for the current element \(e\). After processing the \texttt{endElement} SAX event for \(e\), \(M\) outputs an \(L^F\) set for \(e\), and an \(L^T\) set for its parent \(e'\).

\textbf{Resource requirements of M:} First, consider memory space. The forest \(F\) takes space \(O(|Q|)\). The stack \(S\) contains exactly the open document nodes. Consider each record \(record(e)\) in \(S\); if the alphabet \(\Sigma\) is large, then instead of storing \(\tau(e)\), we can store a boolean array of size \(m\), indicating which \(\tau(j)\) \((j \in F)\) are matched by \(\tau(e)\). So, each record needs \(O(|Q|)\) bits; the total space is \(O(d|Q|)\) bits.

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 \texttt{endElement} events; it is 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 \(sel_{e'}[j]\) takes time proportional to \(\text{outdegree}(j)\) in \(F\); over all \(j \in F\), this takes
\[ \sum_{j \in F} \text{outdegree}(j) = O(|Q|) \text{ time.} \] So, \( M \) spends \( O(|Q|) \) time for each endElement event. Since there are totally \( O(|D|) \) events, runtime is \( O(|Q||D|) \).

**Theorem 5.1.** The predicate checker \( M \) correctly determines when the current element in \( D \) has satisfied/failed each predicate in \( Q \). It uses \( O(d|Q|) \) bits of space and \( O(|Q||D|) \) time.

**Proof.** Note that \( M \) updates the arrays \( \text{self}_e, \text{child}_e \) and \( \text{desc}_e \) 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 resource analysis appears above. \( \diamond \)

The XPush machine [19] can be used in place of our predicate checker, but it would need memory space and runtime exponential in \( |Q| \). Also, unlike the XPush machine, our predicate checker can be extended to more general predicates (see Section 8). 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 \( \text{self}[v] = 1 \) 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. As per our lower bound in [30], the memory space used by this algorithm is optimal for some worst case queries. Bar-Yossef et al. [6] presented a filtering algorithm that uses \( O(r|Q|(|Q|+\log d)) \) bits of space and \( O(r|Q||D|) \) time. Recently, Gou et al. [17] presented an algorithm similar to ours that uses \( O(r|Q|\log d) \) bits of space and \( O(|Q||D|) \) time. Unlike our algorithm, it uses one stack for each nonleaf vertex in \( Q \).

### 6 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 \( Q \) (outside the predicates) have the descendant axis; Section 7 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 5 handles those axes.

Detailed pseudo code for the algorithm is given in the Appendix. In each line, the character
sequence “//” precedes comments. Procedures \texttt{PStartDocument}, \texttt{PStartElement}, \texttt{PText} and \texttt{PEndElement} are called in response to SAX events \texttt{startDocument}, \texttt{startElement}, \texttt{text} and \texttt{endElement}, respectively, after the predicate checker (Section 5) has processed the event.

Our algorithm uses path stacks and candidate stacks. Procedure \texttt{PStartDocument} initializes the path stacks, candidate stacks, their top pointers, and the variables \texttt{nempty} (defined in Section 4), and \texttt{satUpto} (defined below).

Path stacks \(S_i, 1 \leq i \leq n\), were described in Section 4. Recall that, for \(e \in S_i\), they maintain a compact representation of the set \(\text{Paths}_i(e)\) of paths. Each path \(P \in \text{Paths}_i(e)\) represents an embedding \(\Gamma \in \text{embeddings}_i(e)\); no such path represents any embedding in \(\text{embeddings}_i(\text{all}) - \text{embeddings}_i()\). We will not distinguish between a path \(P\) and the embedding it represents. Our statements below that pertain to embeddings also apply to the paths (if any) that represent them. We have the following.

\begin{definition}
[Satisfying Embedding] An embedding \(\Gamma \in \text{embeddings}_i(\text{all})\) \((1 \leq i \leq n)\) is called a \textbf{satisfying embedding} if \(\Gamma(v_j)\) satisfies \(\text{predicate}(L_j)\), for \(1 \leq j \leq i\).
\end{definition}

Note that whether an embedding \(\Gamma \in \text{embeddings}_i(\text{all})\) is a satisfying embedding might depend on the yet to be seen part of \(D\). We want to distinguish this from the following.

\begin{definition}
[Satisfied Embedding] At a particular time instant, an embedding \(\Gamma \in \text{embeddings}_i(\text{all})\) \((1 \leq i \leq n)\) is called a \textbf{satisfied embedding} if \(\Gamma(v_j)\) has already satisfied \(\text{predicate}(L_j)\), for \(1 \leq j \leq i\), based on the input stream seen up to that instant.
\end{definition}

By Fact 2.1, we have the following.

\begin{fact}
\(Q(D) = \{e \in D \mid \exists \text{ a satisfying embedding } \Gamma \in \text{embeddings}_n(e)\}\).
\end{fact}

\begin{definition}
[Distinct Embeddings] Two embeddings \(\Gamma, \Gamma' \in \text{embeddings}_i(\text{all})\) \((1 \leq i \leq n)\) are \textbf{distinct} if \(\Gamma(v_j) \neq \Gamma'(v_j)\), for some \(j, 1 \leq j \leq i\).
\end{definition}

\begin{definition}
[One Embedding Below Another] Consider two distinct embeddings \(\Gamma, \Gamma' \in \text{embeddings}_i(\text{all})\) \((1 \leq i \leq n)\). We say that \(\Gamma\) is \textbf{below} \(\Gamma'\), denoted by \(\Gamma \prec \Gamma'\), if \(\Gamma(v_j)\) is a (not necessarily proper) ancestor of \(\Gamma'(v_j)\), for \(1 \leq j \leq i\).
\end{definition}

Definition 6.4 applies to two distinct paths \(P, P' \in \text{Paths}_i()\) as follows: \(P \prec P'\) if \(P(j)\) is either
same as \(P'(j)\) or is below \(P'(j)\) in \(S_j\), for \(1 \leq j \leq i\).

We describe our algorithm in the following subsections.

### 6.1 Avoiding Redundant Nodes in the Path Stacks

This subsection and the next one deal with the procedure \(PStartElement\); this procedure processes a new element \(e\). In Section 4, we specified three conditions that \(e\) must satisfy, in order for it to be pushed into a path stack \(S_i\). Here, we consider the third of those conditions: \(e\) is not redundant in \(S_i\).

We need the following definitions.

**Definition 6.5.** [Redundant Embedding] An embedding \(\Gamma' \in \text{embeddings}_i(\text{all})\) \((1 \leq i \leq n)\) is redundant if there exists another embedding \(\Gamma \in \text{embeddings}_i(\text{all})\) such that the following hold:

- If \(i = n\), then \(\Gamma(v_n) = \Gamma'(v_n)\).
- \(\Gamma \prec \Gamma'\).
- \(\Gamma'\) cannot be a satisfying embedding unless \(\Gamma\) is. ◦

Pertaining to the third item above, note that we do not consider interrelationships between \(\text{predicate}(L_i)\), \(1 \leq i \leq n\). We treat the \(n\) predicates as independent black boxes: Any node in \(D\) can satisfy/fail a predicate, independent of the satisfaction/failure of any predicates, at any other node.

**Definition 6.6.** [Redundant Node] An open node \(e\) is redundant in \(S_i\) \((1 \leq i < n)\) if all the embeddings in \(\text{embeddings}_i(e)\) are redundant. ◦

Note that redundancy does not apply to \(S_n\): \(S_n\) contains candidates (see Subsection 6.2), and candidates can not be redundant. Redundant nodes in path stacks waste the space they occupy. In addition, they delay the output of elements that have already qualified for output (based on the stream seen so far), thereby wasting even more space, as seen from the following example.

**Example 6.1.** Suppose that, in Figure 3, we change \(\text{axis}(L_3)\) from \text{child} to \text{descendant}. As seen in Example 4.1, \(\text{Paths}_4()\) consists of eleven paths.

The path \(P_1 = a_2b_1c_1d_2\) is below \(P_2 = a_2b_2c_2d_2\). Consider the instant when \(d_2\) satisfies \(\text{predicate}(L_4)\). Suppose that \(P_1\) is a satisfied embedding; then \(P_2\) is redundant. Also, \(P_2\) hides \(P_1\), thereby delaying the output of \(d_2\). To prevent this, we need to avoid pushing \(b_2\) and \(c_2\).
Because of the way our predicate checker works, we update \( R_i(e) \) only when \( e \) is the current node (so, \( \text{top}(S_i) = R_i(e) \)). So, when element \( b_2 \) became open, we must have had \( R_1(a_2).\text{predStatus} = R_2(b_1).\text{predStatus} = \text{True} \); i.e., \( a_2b_1 \) is a satisfied embedding; this is enough to infer that \( b_2 \) is redundant in \( S_2 \). Similarly, when \( c_2 \) became open, we must have further had \( R_3(c_1).\text{predStatus} = \text{True} \); so, \( a_2b_1c_1 \) is a satisfied embedding; this is enough to infer that \( c_2 \) is redundant in \( S_3 \).

Now consider the case that \( a_2b_1 \) is not a satisfied embedding, when \( b_2 \) became open. Suppose that \( R_1(a_1).\text{predStatus} = R_1(a_2).\text{predStatus} = \text{Unknown} \) but \( R_2(b_1).\text{predStatus} = \text{True} \). Then \( b_2 \) is redundant in \( S_2 \) for the following reason: An embedding \( \Gamma' \in \text{embeddings}_2(b_2) \) can become a satisfied embedding only if \( a_1 \) or \( a_2 \) satisfies \( \text{predicate}(L_1) \); but then an embedding \( \Gamma \in \text{embeddings}_2(b_1) \) (\( \Gamma \prec \Gamma' \)) would become a satisfied embedding. ⚫

The following Lemma characterizes redundant nodes.

**Lemma 6.1.** A new element \( e \) is redundant in \( S_i \) \((1 \leq i < n)\) if the following holds:

\[
\text{top}(S_i).\text{predStatus} = \text{True} \quad \text{and} \quad \text{top}(S_i).\text{leftPtr} = t_{i-1}. \tag{**}
\]

Also, suppose that we consistently avoid pushing any element into any path stack for which (**) holds. Then a new element \( e \) is redundant in \( S_i \) only if (**) holds.

**Proof.** First consider the “if” part. Suppose that \( \text{top}(S_i) = e' \) satisfies (**). Consider any path \( P \in \text{Paths}_i(e) \). Let \( P' \) be the path obtained from \( P \) by replacing the last node \( e \) with \( e' \). Since \( R_i(e').\text{leftPtr} = t_{i-1} = R_i(e).\text{leftPtr} \), \( P' \in \text{Paths}_i(e') \); also, \( P' \prec P \). Since \( R_i(e').\text{predStatus} = \text{True} \), \( P \) is a satisfying path only if \( P' \) is. So, \( e \) is redundant in \( S_i \).

Now, consider the “only if” part. Suppose that \( \text{top}(S_i) = e' \) either does not satisfy \( \text{predicate}(L_i) \) or does not point to \( \text{top}(S_{i-1}) \). Consider the consistent behavior referred to in the lemma. The top most path in \( \text{Paths}_i(e) \) can be a satisfying path, even if no path in \( \text{Paths}_i(\leq e') \) is. So, \( e \) is not redundant in \( S_i \). ⚫

Our algorithm consistently avoids pushing any element into any path stack for which (**) holds. For the sake of efficient implementation, we introduce the variable \( \text{satUpto} \) (abbreviation for “satisfied upto”). It is the largest integer less than \( n \) such that the following holds: For \( 1 \leq i \leq \text{satUpto} \), \( \text{top}(S_i) \)
satisfies (**) above. Then, the top most path in $Paths_{satUpto}$ is a satisfied path. So, any new element is redundant in any $S_i$, for all $i \leq satUpto$. Hence, while considering a new element $e$, we only need to consider $S_i$, for $i > satUpto$.

Procedure $PStartDocument$ initializes $satUpto$ to 0; the other procedures update its value. Procedure $PStartElement$ considers pushing an element $e$ into $S_i$, only for $i > satUpto$. Procedure $notRedPush(i, R)$ pushes a new element’s record into $S_i$ only if $S_i$ does not satisfy (**).

**Lemma 6.2.** Consider path stacks $S_i$, $i < n$. Procedure $PStartElement$ pushes a new element $e$ into $S_i$, iff $embeddings_i(e) \neq \emptyset$ and $e$ is not redundant in $S_i$.

**Proof.** The procedure pushes $e$ into $S_i$ only if $e$ matches $nodeTest(L_i)$ and $i \leq nempty + 1$. This equals the condition that $embeddings_i(e) \neq \emptyset$. Restricting $i$ to be greater than $satUpto$, and using the procedure $notRedPush$ equals the condition that $e$ is not redundant in $S_i$. ∎

### 6.2 Path Stack $S_n$

In Section 4, we specified three conditions that a new element $e$ must satisfy, in order for it to be pushed into a path stack $S_i$. Path stack $S_n$ is special, because there is no concept of redundancy in $S_n$. So, $e$ qualifies for $S_n$ if it meets the first two conditions. Such elements are potentially candidates for output.

**Fact 6.2.** An element $e \in S_n$ qualifies for output iff (and when) the following conditions are met:

1. $e$ satisfies $predicate(L_n)$.

2. Any one path in $Paths_{n-1}(\leq (*R_n(e).leftPtr))$ becomes a satisfied path

(*$x$ denotes the dereferencing of a pointer $x$). ∎

Candidates start their candidacy in $S_n$. 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 6.4 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 6.2.** Consider Figure 3 with $axis(L_3) = \text{descendant}$. In Figure 3b, consider the following scenarios:

- $d_2$ fails $\text{predicate}(L_4)$: $d_2$’s candidacy dies; it is popped from $S_4$ and discarded.
- $\text{satUpto} = 3$ when $d_2$ satisfies $\text{predicate}(L_4)$: $a_2b_2c_2d_2$ is a satisfied path. $d_2$ is popped from $S_4$ and output.
- $\text{satUpto} < 3$ when $d_2$ satisfies $\text{predicate}(L_4)$: No path in $\text{Paths}_4(d_2)$ is currently a satisfied path (based on the part of $D$ seen so far). Then, none of these paths can become a satisfied path until $d_2$ closes. $d_2$ is kept in $S_4$ until $d_2$ closes; then it is moved to candidate stack $C_3$ associated with $S_3$, and made to point to $c_2$. ◇

**Lemma 6.3.** Procedure $\text{PStartElement}$ considers a new element $e$ for $S_n$ iff $\text{embeddings}_n(e) \neq \emptyset$. It outputs $e$ iff $e$ already qualifies for output based on the stream seen so far; else it pushes $e$ into $S_n$.

**Proof.** The procedure considers $e$ for $S_n$ only if $e$ matches $\text{nodeTest}(L_n)$ and $n \leq \text{nempty} + 1$. This equals the condition that $\text{embeddings}_n(e) \neq \emptyset$. It outputs $e$ if $\text{predicate}(L_n) = \text{nil}$ and $\text{satUpto} = n - 1$; this equals the condition that $e$ already qualifies for output. ◇

### 6.3 Candidate Stacks and Their Interaction with Path Stacks

For $1 \leq i < n$, $C_i$ is the candidate stack associated with $S_i$. The $C_i$s together contain all closed elements that are candidates (open candidates are in $S_n$). Also, the $C_i$s are disjoint: No candidate appears in two or more $C_i$s simultaneously. An element $e$ in $C_i$ is represented by the record $R_i'(e) = (\tau(e), \text{event#}, \text{pathPtr})$. $\text{pathPtr}$ always points to a node in $S_i$; when $e$ is pushed into $C_i$, its $\text{pathPtr}$ gets the then value of $t_i$; whenever $\text{pathPtr}$ changes, it always gets the then value of $t_i$. We group all the elements in $C_i$ that have the same value for $\text{pathPtr}$ (they must be contiguous in $C_i$) into a bunch with a single $\text{pathPtr}$.

**Fact 6.3.** Let $\text{Bunch}$ be a bunch of candidates in $C_i$ that have the same value for $\text{pathPtr}$. All the candidates in $\text{Bunch}$ qualify for output iff (and when) some path in $\text{Paths}_i(\leq (*\text{pathPtr}))$ becomes
a satisfied path. When $pathPtr = t_i$, some path in $Paths_i (\leq \ast pathPtr)$ is a satisfied path iff $satUpto = i (\ast x$ denotes the dereferencing of a pointer $x$). ◊

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 6.4.** Whenever we push/pop/access/modify/delete the record pertaining to an element $e$, in any path stack $S_i$ ($1 \leq i < n$), $e$ must be the current element. None of $e$’s descendants is open. So, $e$ must be the top element in $S_i$; also, no element from $S_{i+1}$ can point to it (through $leftPtr$). ◊

As explained in Section 5, $L^T$ and $L^F$ are sets of location steps whose predicates become $True$ and $False$, respectively, at current element $e$, as a result of 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 $deleteFalse$. $e$ should be popped from $S_i$. $TopBunch(C_i)$, if pointing to $e$ through its $pathPtr$, 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 $pathPtr$ (see procedure $pushBunch$). 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 a satisfied path; 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 \(\text{topBunch}(C_i).\text{pathPtr}\) was pointing to \(e\), then (since \(R_i(e).\text{predStatus} = \text{True}\)) we move this bunch from \(C_i\) to \(C_{i-1}\). Decrement \(\text{satUpto}\), if necessary, to reflect the deletion of \(e\).

**Example 6.3.** Continuation of Example 6.2 (third scenario). \(d_2\) has been moved to \(C_3\), with its \(\text{pathPtr}\) pointing to \(c_2\). Consider three cases for \(c_2\):

- \(c_2\) fails \(\text{predicate}(L_3)\): \(c_2\) is popped from \(S_3\). \(d_2\) in \(C_3\) now points to \(c_1\). \(b_2\) closes with or without satisfying \(\text{predicate}(L_2)\), and is popped from \(S_2\). If \(d_1\) fails \(\text{predicate}(L_4)\), it is popped from \(S_4\) and discarded. Suppose that \(d_1\) passes \(\text{predicate}(L_4)\). When \(d_1\) closes, it is moved to \(C_3\), forming a single bunch with \(d_2\), with their \(\text{pathPtr}\) pointing to \(c_1\).

- \(\text{satUpto} = 2\) and \(c_2\) passes \(\text{predicate}(L_3)\): \(\text{satUpto}\) is incremented to 3; this indicates that \(\text{top}(S_1)\text{top}(S_2)\text{top}(S_3) = a_2b_2c_2\) is a satisfied path. Since \(R_3'(d_2).\text{pathPtr} = t_3\), \(d_2\) is output.

- \(\text{satUpto} < 2\) and \(c_2\) passes \(\text{predicate}(L_3)\): \(R_3(c_2).\text{predStatus}\) 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 \(\text{pathPtr}\) pointing to \(b_2\).

**Lemma 6.4.** Procedures \(\text{deleteFalse}, \text{setTrue}\) and \(\text{deleteTrue}\) correctly handle the three cases itemized prior to Example 6.3, respectively.

Following a text event, the predicate checker returns a pair \((L^T, L^F)\) for the current element \(e\) (see Section 5). Then, procedure \(P\text{Text}\) processes the changes at \(e\) using procedures \(\text{deleteFalse}\) and \(\text{setTrue}\). 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 5). Procedure \(P\text{EndElement}\) first processes the changes at \(e\), using procedures \(\text{deleteFalse}, \text{setTrue}\) and \(\text{deleteTrue}\). Then, it process the changes at \(e'\) using procedures \(\text{deleteFalse}\) and \(\text{setTrue}\).

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

**Lemma 6.5.** Procedures \(P\text{Text}\) and \(P\text{EndElement}\) correctly process the changes resulting from a text event and an endElement event, respectively.
6.4 Resource Requirements of Our Algorithm

We need to add up the resource requirements of the various components: Predicate checker, path stacks and candidate stacks.

First, consider memory space. The predicate checker requires $O(d|Q|)$ space (Theorem 5.1). In the worst case, each open node could be in each path stack; space required for path stacks is $O(nd)$. The candidate stacks use $O(1)$ space for each candidate; worst case space required is $O(c)$. So, the overall memory space required is $O(d|Q| + c)$.

Now, consider runtime. The predicate checker uses $O(|Q||D|)$ time (Theorem 5.1). We spend $O(1)$ time for each element $e$, in each path stack $S_i$ (i.e., each record $R_i(e)$), for each of the following events: $e$ is pushed into $S_i$, $\text{predicate}(L_i)$ evaluates to True/False at $e$, and $e$ closes and is deleted from $S_i$. So, overall, we spend $O(1)$ time for each record $R_i(e)$; this includes the time spent on the candidate stacks, as a result of changes to $R_i(e)$. Hence, the worst case time spent on path and candidate stacks is $O(n|D|)$. So, the overall worst case runtime is $O(|Q||D|)$. This is same as the runtime of the best in-memory algorithms [15, 27] that use $\Theta(|D|)$ memory space.

**Theorem 6.1.** The algorithm described in this section correctly evaluates a CXPath query $Q$ on a streaming XML document $D$, when all the location steps in $Q$ (outside the predicates) have the descendant axis. The algorithm uses $O(d|Q| + c)$ space and $O(|Q||D|)$ time, where $d$ is the depth of $D$, and $c$ is the maximum number of candidate nodes at any one time.

**Proof.** The correctness proof follows from Facts 6.1 to 6.4 and Lemmas 6.1 to 6.5. The resource analysis appears above. ◦

7 Modifications When There Are Child Axes

In this section, we consider the modifications needed in our algorithm, for general CXPath queries $Q$: Some location steps in $Q$ might have the child axis. All of our statements (including Definitions, Facts and Lemmas) prior to Section 6.1 apply to general queries. In the following Subsections 7.1 to 7.4, we consider the modifications pertaining to Subsections 6.1 to 6.4, respectively.
7.1 Avoiding Redundant Nodes in the Path Stacks

Recall that, for the case when all the axes in $Q$ are descendant axes, Definition 6.5 defined redundant embeddings in $\text{embeddings}_i(\text{all})$. For the general case, we define redundancy only for embeddings in $\text{embeddings}_n(\text{all})$; i.e., for $i = n$. The following definition is a specialization of Definition 6.5, for $i = n$.

**Definition 7.1.** [Redundant Embedding] An embedding $\Gamma' \in \text{embeddings}_n(f)$ is redundant if there exists another embedding $\Gamma \in \text{embeddings}_n(f)$ such that the following hold:

- $\Gamma \prec \Gamma'$.
- $\Gamma'$ cannot be a satisfying embedding unless $\Gamma$ is. ⋄

The following definition replaces Definition 6.6. For the special case considered in Section 6, the two definitions are equivalent.

**Definition 7.2.** [Redundant Node] An open node $e$ is redundant in $S_i$ ($1 \leq i < n$) if all the embeddings $\Gamma' \in \text{embeddings}_n(\text{all})$ (over all possible documents $D$ that are consistent with the stream seen so far) with $\Gamma'(v_i) = e$ are redundant. ⋄

To characterize redundant nodes, we need the following definitions.

**Definition 7.3.** [Trunk Segment, Length] A segment of trunk($Q$) is a chain $(v_l, v_{l+1}, \ldots, v_k)$ of consecutive vertices in trunk($Q$), such that the following hold:

- Either $l = 1$ or $\text{axis}(L_l) = \text{descendant}$.
- $k < n$ and $\text{axis}(L_{k+1}) = \text{descendant}$.
- For all $i, l < i \leq k$, $\text{axis}(L_i) = \text{child}$. ⋄

The length of a segment is the number of vertices in it. ⋄

**Definition 7.4.** [Satisfied Segment] At any particular time instant, a trunk segment $(v_l, v_{l+1}, \ldots, v_k)$ is satisfied if for all $j, l \leq j \leq k$, there exists a node $e_j$ in $S_j$, such that the following hold:

- $R_j(e_j).\text{predStatus} = \text{True}$.
- $R_l(e_l).\text{leftPtr} = t_{l-1}$.
• For \( l < j \leq k \): \( R_j(e_j).leftPtr \) points to \( R_{j-1}(e_{j-1}) \); i.e., \( e_{j-1} \) is the parent of \( e_j \) in \( D \).

Note that whether a segment is satisfied changes with time. A satisfied segment could become unsatisfied as one or more \( e_j \)'s in Definition 7.4 close.

**Example 7.1.** Consider the query \( Q = //a//b//c//d \) of Example 4.1. It contains the two segments \((v_1)\) and \((v_2, v_3)\). If we changed \( axis(L_1) \) to \( \text{child} \), the query would still contain the same two segments.

Consider the path stacks shown in Figure 3. Suppose that \( Q \) contains predicates (not shown). Note that 
\[
R_2(b_1).leftPtr = R_2(b_2).leftPtr = t_1.
\]
If either
\[
R_2(b_1).predStatus = R_3(c_1).predStatus = \text{True},
\]
or 
\[
R_2(b_2).predStatus = R_3(c_2).predStatus = \text{True},
\]
then \((v_2, v_3)\) would be a satisfied segment.

The following lemma characterizes redundant nodes.

**Lemma 7.1.** A new element \( e \) is redundant in \( S_i \) \((1 \leq i < n)\) iff the following hold:

- There is a trunk segment \((v_l, v_{l+1}, \ldots, v_k)\), such that \( l \leq i \leq k \).
- \((v_l, v_{l+1}, \ldots, v_k)\) is a satisfied segment.

**Proof.** First consider the “if” part. Consider any embedding \( \Gamma' \in \text{embeddings}_n(f) \), for some future node \( f \), with \( \Gamma'(v_i) = e \). Let \( \Gamma \in \text{embeddings}_n(f) \) be the embedding obtained from \( \Gamma' \), by replacing \( \Gamma'(v_l) \Gamma'(v_{l+1}) \cdots \Gamma'(v_k) \) with \( e_l e_{l+1} \cdots e_k \) (specified in Definition 7.4). \( \Gamma \) and \( \Gamma' \) satisfy the two conditions in Definition 7.1; so \( \Gamma' \) is redundant. By Definition 7.2, \( e \) is redundant in \( S_i \).

Now consider the “only if” part. Suppose that the first condition in the lemma fails. Then \( axis(L_{i+1}) = axis(L_{i+2}) = \cdots = axis(L_n) = \text{child} \). If any one embedding in \( \text{embeddings}_n(f) \) (for some future node \( f \)), maps \( v_i \) to \( e \), then every embedding in \( \text{embeddings}_n(f) \) must map \( v_i \) to \( e \). So, \( e \) is not redundant in \( S_i \).

Now, suppose that the first condition in the lemma holds, but the second condition fails. By controlling the remaining part of the input stream, we can ensure that, for some node future \( f \), any satisfying embedding in \( \text{embeddings}_n(f) \) must map \( v_i \) to \( e \). So, \( e \) is not redundant in \( S_i \).

Now, consider the modifications to our algorithm. For each trunk segment 
\((v_l, v_{l+1}, \ldots, v_k)\), our algorithm maintains a boolean variable \( \text{satisfied} \), indicating whether or not the
segment is currently satisfied. We think of this variable as being associated with the path stack $S_k$. If $S_k.satisfied = True$, then a new element is redundant in $S_i$, for all $i, l \leq i \leq k$. If $S_k.satisfied = False$, we allow pushing new elements into $S_i$ ($l \leq i \leq k$); so, for $i < k$, a node could get pushed into $S_i$, even if $Paths_i()$ already contains a satisfied path. So, a satisfied path could get hidden underneath a not-yet-satisfied path. To recognize this when it happens, 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 satisfied path in $Paths_i(e)$; inductively,

$$R_i(e).actualMatch = \ast(R_i(e).leftPtr).actualMatch \land R_i(e).predStatus.$$ 

If $R_i(e).actualMatch = True$, then any candidate in $C_i$ whose $pathPtr$ is pointing to $R_i(e)$ can be output immediately; this will be of use in Subsection 7.3.

The variable `satUpto` is redefined as the largest index $k < n$ that satisfies the following conditions:

- $top(S_k).actualMatch = True$
- $axis(L_{k+1}) = descendant$.

Equivalently, `satUpto` is the right end point of the rightmost segment for which the following holds: That segment and all segments to its left are satisfied.

Our algorithm considers pushing a new element $e$ into $S_i$, only for $i > satUpto$. It pushes $e$ into $S_i$ iff the following hold:

- $embeddings_i(e) \neq \emptyset$.
- Either $v_i$ does not belong to any segment, or the segment containing $v_i$ is not satisfied.

Because of the second condition in the above definition of `satUpto`, `satUpto` moves in quantum jumps. Unlike in Section 6, there could be elements that were pushed into $S_i$, for $i < satUpto$, on top of a node belonging to a satisfied path in $Paths_i()$, before `satUpto` jumped to its current value; such elements are not redundant, as seen from the following example. Same applies to elements that were pushed into $S_i$, for $i > satUpto$, before the segment containing $v_i$ became satisfied.

**Example 7.2.** Let $Q = //a/*/*/*//*///*$, where predicates are present but not shown. $Trunk(Q)$ contains only the one segment $(v_1, v_2, \ldots, v_5)$. In Figure 5, we show three different embeddings of
$v_1 v_2 \cdots v_5$ in the current path to node $I$ ($v_6$ is irrelevant to our discussion here). Numbers 1 through 5 denote indices of the vertices $v_1$ through $v_5$, and upper case letters $A$ through $I$ are document node ids.

Since $axis(L_6) = \text{descendant}$, the contents of $S_1, S_2, \ldots, S_5$ could be as shown only if

$$R_5(G).\text{actualMatch} = R_5(E).\text{actualMatch} = False.$$  

Recall that new nodes are considered for insertion in $S_i$, in decreasing order of $i$. Suppose that $R_5(G).\text{actualMatch}$ is $True$ when $G$ is pushed into $S_5$. Then, $CDEFG$ is a satisfied path; $\text{satUpto}$ jumps from 0 to 5. Then we would not push $G$ into $S_3$, as it is redundant; consequently, we would not push $H$ and $I$ into $S_4$ and $S_5$, respectively.

But $R_1(E)$ and $R_2(F)$ were pushed before $\text{satUpto}$ 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 $\text{nodeTest}(L_5)$ or $\text{predicate}(L_5)$, and so candidates that are descendants of $G'$ can not use a path with the prefix $CDEFG'$, to qualify for output. But $G'$ might pass $\text{nodeTest}(L_3)$ and get pushed into $S_3$; so, some of those candidates could rely on a possible path with the prefix $EFG'$.

\[ \diamond \]

### 7.2 Path Stack $S_n$

For the general case, Fact 6.2 is modified as follows.

**Fact 7.1.** An element $e \in S_n$ qualifies for output iff (and when) the following conditions are met:
1. $e$ satisfies $\text{predicate}(L_n)$.

2. Consider $\text{axis}(L_n)$.

   (a) $\text{Axis}(L_n) = \text{descendant}$. Any one path in $\text{Paths}_{n-1}(\leq *(R_n(e).leftPtr))$ becomes a satisfied path.

   (b) $\text{Axis}(L_n) = \text{child}$. Any one path in $\text{Paths}_{n-1}(*(R_n(e).leftPtr))$ becomes a satisfied path. ◦

The only difference in our algorithm (with respect to Section 6) is in how we determine whether condition 2) in the Fact is met. For both 2a) and 2b), the condition is met iff $*(R_n(e).leftPtr).\text{actualMatch} = \text{True}$. For 2b), this is obvious; for 2a), this is due to the following. If $R_{n-1}(e').\text{actualMatch}$ was True for some ancestor $e'$ of $e$, the segment with right endpoint $n - 1$ would become satisfied; so, we would not have pushed any elements on top of $e'$ in $S_{n-1}$; consequently, $R_n(e).leftPtr$ must point to $e'$.

7.3 Candidate Stacks and Their Interaction with Path Stacks

Now, let us consider the modifications pertaining to candidate stacks. As in Section 6, each closed candidate $e$ appears in a unique $C_i$. In Section 6, $e$ was represented by the record $R'_i(e) = (\tau(e), \text{event} \#, \text{pathPtr})$.

Now, we add the additional field $\text{stackSeq}$ to $R'_i(e)$. This field and its use constitute the intellectually hardest part of this paper. If there is no segment containing $v_i$, then $R'_i(e).\text{stackSeq} = (i)$. Now, let $(v_i, v_{i+1}, \ldots, v_k)$ be the segment containing $v_i$. $\text{StackSeq}$ is a variable length sequence of some stack indices $j$, $i \leq j \leq k$; it keeps track of possible alternate paths for $e$ to qualify for output. Whenever we process $R'_i(e)$, the top elements of all the path stacks whose indices are in $R'_i(e).\text{stackSeq}$ are the same document element. We will first explain this with an example, before giving a formal description.

Example 7.3. Continuing with Example 7.2, suppose that $\text{satUpto}$ stays at 0, and at some point we have three embeddings of $v_1v_2\cdots v_5$ in the current path to node $I$, as shown in Figure 5. Recall that $\text{axis}(L_6) = \text{descendant}$. Since $\text{satUpto} = 0$, the $\text{actualMatch}$ field must be $\text{False}$ for all three records in $S_5$; otherwise, $\text{satUpto}$ would be 5.

For any element in $C_5$, its $\text{stackSeq}$ is $(5)$. Suppose that for the top element $e$ in $C_5$, $R'_5(e).\text{pathPtr} = t_5$; i.e., $R'_5(e)$ is pointing to $R_5(I)$. Consider the following two alternatives for $R_5(I)$:
• $I$ fails $\text{predicate}(L_5)$. Since $axis(L_6) = \text{descendant}$, $e$ could try the path ending in $R_5(G)$; so, we will pop $R_5(I)$ and set $R_5'(e).\text{pathPtr}$ to the new value of $t_5$, just as in Section 6.

• $I$ satisfies $\text{predicate}(L_5)$. As in Section 6, $e$ will stay in $C_5$ until $I$ closes, and then $e$ would be moved to $C_4$, with its $\text{pathPtr}$ pointing to $R_4(H)$ and $\text{stackSeq} = (4)$.

Let us continue with the second alternative above. Consider the following two alternatives for $R_4(H)$:

• $H$ fails $\text{predicate}(L_4)$. $H$ would be popped from $S_4$. Instead of just changing $R_4'(e).\text{pathPtr}$ to the new value of $t_4$ as done in Section 6, we have to move $e$ back to $C_5$ with $\text{pathPtr}$ pointing to $\text{top}(S_5) = R_5(G)$, and $\text{stackSeq} = (5)$.

• $H$ satisfies $\text{predicate}(L_4)$. As in Section 6, $e$ will stay in $C_4$ until $H$ closes, and then $e$ would be moved to $C_3$, with $\text{pathPtr}$ pointing to $R_3(G)$, and $\text{stackSeq} = (3)$. Since the top element in $S_3$ is same as that in $S_5$, we would append 5 to the $\text{stackSeq}$ field: $R_3'(e).\text{stackSeq} = (3,5)$. This signifies that $e$ could try an alternate path ending with $R_5(G)$; i.e., $e$ would be output iff there exists a satisfying path in $\text{Paths}_3(G)$ or $\text{Paths}_5(\leq G)$.

Let us continue with the second alternative above. Note that $G$ might not stay as the top element in $S_5$, as for example, if we next push a sibling $H'$ of $H$ into $S_4$, and then push a child $I'$ of $H'$ into $S_5$; we still do not need to store a pointer to $R_5(G)$ in $R_4'(e)$. The reason for this: Next time we want to process $R_4'(e)$, $G$ will again be the current element, and be the top element in $S_3$ and $S_5$ (Fact 6.4).

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

• $G$ satisfies $\text{predicate}(L_5)$ but does not yet fail $\text{predicate}(L_3)$. Set $R_5(G).\text{predStatus} = \text{True}$; if $G$ also satisfies $\text{predicate}(L_3)$ then set $R_3(G).\text{predStatus} = \text{True}$. If $R_5(G).\text{actualMatch}$ becomes $\text{True}$, increment $\text{satUpto}$ to 5, pop and output $R_3'(e)$, and pop $R_3(G)$ as it is redundant. Else if $R_3(G).\text{actualMatch}$ becomes $\text{True}$, then pop and output $R_3'(e)$.

• $G$ satisfies $\text{predicate}(L_5)$ but fails $\text{predicate}(L_3)$. Set $R_5(G).\text{predStatus} = \text{True}$ and pop $R_3(G)$. Pop $e$ from $C_3$ and push it into $C_5$, with $\text{stackSeq} = (5)$. If $R_5(G).\text{actualMatch}$ becomes $\text{True}$, increment $\text{satUpto}$ to 5, pop and output $e$.

• $G$ fails $\text{predicate}(L_5)$ but does not yet fail $\text{predicate}(L_3)$. Pop $R_5(G)$; if $\text{topBunch}(C_5)$ was pointing to it, make it point to $R_5(E)$. Delete 5 from $R_3'(e).\text{stackSeq}$; it becomes (3). If $G$
passes \textit{predicate}(L_3), set \(R_3(G).\text{predStatus} = \text{True}; \) if \(R_3(G).\text{actualMatch}\) becomes \text{True}, then pop and output \(e\).

- \(G\) fails \textit{predicate}(L_5) and \textit{predicate}(L_3). Pop \(R_3(G)\) and \(R_5(G)\).

If \(\text{topBunch}(C_5)\) was pointing to \(R_5(G)\), then make it point to \(R_5(E)\). Move \(e\) from \(C_3\) to \(C_5\), with \(\text{pathPtr}\) pointing to \(R_5(E)\) and \(\text{stackSeq} = (5)\).

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

- Both \textit{predicate}(L_5) and \textit{predicate}(L_3) are \text{True}. \(e\) is moved from \(C_3\) to \(C_2\), with \(\text{pathPtr}\) pointing to \(R_2(F)\) and \(\text{stackSeq} = (2, 4)\) (i.e., pointing to \(R_2(F)\) and \(R_4(F)\)).

- Only \textit{predicate}(L_5) is \text{True}. \(e\) is moved from \(C_5\) to \(C_4\), with \(\text{pathPtr}\) pointing to \(R_4(F)\) and \(\text{stackSeq} = (4)\).

- Only \textit{predicate}(L_3) is \text{True}. \(e\) is moved from \(C_3\) to \(C_2\), with \(\text{pathPtr}\) pointing to \(R_2(F)\) and \(\text{stackSeq} = (2)\).

In all the three cases above, if the new \(\text{top}(S_5)\) is \(F\) (in our example, it is \(E\)), then 5 would be appended to \(R^t(e).\text{stackSeq}\).

As in Section 6, we group all elements in \(C_i\) that have the same value for \(\text{pathPtr}\) and \(\text{stackSeq}\) (they must be contiguous in \(C_i\)) into a \textit{bunch} with a single \(\text{pathPtr}\) and \(\text{stackSeq}\). All elements in a bunch are relying on the same set of paths to qualify for output. If any one of these paths becomes satisfied, then the entire bunch is output.

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

Now, we give a general description of how to handle bunches and their \(\text{stackSeq}\). Consider a bunch in \(C_i\). If there is no segment containing \(v_i\), then its \(\text{stackSeq}\) is just \((i)\). Now, let \((v_i, v_{i+1}, \ldots, v_k)\)
be the segment containing $v_i$. For any bunch in $C_k$, its stackSeq is (re)initalized to $(k)$. Consider a bunch in $C_i$, where $l \leq i < k$. Its stackSeq is an increasing sequence of some integers $j$, $i \leq j \leq k$; also, $i$ is the first (smallest) element of this sequence (so, for the special case considered in Section 6, stackSeq = $(i)$ always). 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 as follows:
  - 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. If $m = l - 1$, reinitialize stackSeq to $(m)$.

We have the following analogue of Fact 6.3.

**Fact 7.2.** Let $Bunch$ be a bunch of candidates in $C_i$ that have the same value for pathPtr and stackSeq; let the common pathPtr point to a node $e$ in $S_i$. Consider two cases:

1. There is no segment containing $v_i$. In this case, the common stackSeq must be $(i)$. All the candidates in $Bunch$ qualify for output iff (and when) some path in $\bigcup_{j \in b} Paths_i(e)$ becomes a satisfied path.

2. There is a segment containing $v_i$. Let $(v_l, v_{l+1}, \ldots, v_k)$ be that segment. All the candidates in $Bunch$ qualify for output iff (and when) some path in $\bigcup_{j \in stackSeq} Paths_j(e) \cup (\bigcup_{f \in ancestors(e)} Paths_k(f))$ becomes a satisfied path. ◢
7.4 Resource Requirements of Our Algorithm

Compared to the analysis in Subsection 6.4, the main change here pertains to candidate stacks, for maintaining the stackSeqs. The length of any stackSeq is bounded by the length of the longest segment. Using the trivial upper bound of \( n - 1 \) for this, the space needed for candidate stacks is \( O(nc) \). Maintaining the stackSeq for any one candidate \( e \) (or bunch), over the lifetime of that candidate, takes \( O(dn) \) time; this is because there could be up to \( (d - m + 1) \) embeddings of a segment of length \( m \) in the current path, and testing if any one of them is satisfied takes \( O(m) \) time (excluding the time taken by the predicate checker). So, the total time spent on candidate stacks is \( O(dn|D|) \). So, the overall memory space and runtime required, in the worst case, are \( O(d|Q| + nc) \) and \( O((|Q| + dn)|D|) \), respectively.

**Theorem 7.1.** The algorithm described in this section correctly evaluates a \( CXPath \) query \( Q \) on a streaming XML document \( D \). The algorithm uses \( O(d|Q| + nc) \) space and \( O((|Q| + dn)|D|) \) time; \( d \) is the depth of \( D \), \( n \) is the number of location steps in \( Q \), and \( c \) is the maximum number of candidate nodes at any one time. ⋆

8 Algorithm Extension

In this section, we show how to extended our algorithm to queries with more complex predicates, without increasing the memory space or runtime. In Subsection 8.1, we consider predicates that involve \( or \) and \( not \). In Subsection 8.2, we consider predicates that involve the preceding and preceding-sibling axes.

8.1 Predicates with \( or \) and \( not \)

Consider a predicate \( P \) that contains the boolean operators \( and \), \( or \) and \( not \). In [27], we showed how \( P \) 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 tag \( \tau(v) \), and a boolean operator \( boolean(v) \) associated with it. \( \tau(v) \in \Sigma \cup \{\ast\} \) is the element type of \( v \). \( Bool(v) \in \{\text{and, or, not}\} \). Each arc \( r \in A \) has an axis \( axis(r) \) associated with it; \( axis(r) \in \{\text{self, child, descendant}\} \). As in Section 2, a leaf vertex could have a “\( <\text{relOp}> \text{const} \)” condition associated with it.
Figure 6: Example 8.1: $P = [.//a/.b > 2 \text{ and not } .//c]/*[./a \text{ or } ./b]]$

Now, the predicate checker must consider the tagnames $\text{bool}(v)$ for each vertex $v$, and $\text{axis}(r)$ for each arc $r$; the extension is tedious but straightforward. We will illustrate this using an example.

**Example 8.1.** Consider the predicate $P = [.//a/.b > 2 \text{ and not } .//c]/*[./a \text{ or } ./b]]$. $\text{Tree}(P)$ is shown in Figure 6. For each vertex $v$, the pair $(\tau(v), \text{bool}(v))$ is shown next to $v$; if $\text{bool}(v)$ is not specified, it should be taken to be $\text{and}$. For each arc $r$, $\text{axis}(r)$ is shown next to it: $s$, $c$ and $d$ stand for $\text{self}$, $\text{child}$ and $\text{descendant}$, respectively.

Consider the $\text{not}$ operator (vertex 6) in Figure 6. For a document node $e$, we have $\text{self}_e[6] = 1$ iff $\text{self}_e[7] = 0$; so, $\text{self}_e[7]$ should be computed before $\text{self}_e[6]$, because of the $\text{self}$ arc from vertex 6 to vertex 7. Now, consider the $\text{or}$ operator (vertex 9). For a document node $e$, we have $\text{self}_e[9] = 1$ iff $\text{self}_e[10] = 1$ or $\text{self}_e[11] = 1$. ◦

Our predicate checker can also be extended to predicates that contain certain XPath library functions, such as aggregation and $\text{position}$. For an example involving aggregation, see [29].

**8.2 Predicates with preceding and preceding-sibling axes**

Predicates containing the preceding and preceding-sibling axes can also be represented as tree patterns; see [27]. For instance, consider the predicate

$$P' = [.//a/.b > 2 \text{ and not } .//c]/*[./a \text{ or } ./\text{preceding-sibling::b}],$$

obtained by modifying the predicate $P$ of Figure 6. $\text{Tree}(P')$ is obtained from $\text{tree}(P)$ by replacing the axis associated with the arc $(12, 13)$ by $\text{preceding-sibling}$.
For our predicate checker in Section 5, we only need that \( self_e \) be known by the time \( e \) closes; this requirement is met for predicates containing \( \text{preceding} \) and \( \text{preceding-sibling} \) axes. From Section 5, recall that \( F \) is the forest representing the predicates; vertices in \( F \) are indexed from 1 to \( m \); for \( 1 \leq j \leq m \), \( T'_j \) is the subtree rooted at vertex \( j \). For each element \( e \) in \( D \), we define two more boolean arrays \( \text{prec}_e \) and \( \text{precsib}_e \) (in addition to \( \text{self}_e, \text{child}_e \) and \( \text{desc}_e \)) indexed from 1 to \( m \), as follows: For \( 1 \leq j \leq m \),

\[
P_1 \quad \text{prec}_e[j] = 1 \text{ iff there is an embedding of } T'_j \text{ in } D, \text{ with } j \text{ mapped to an element preceding } e.
\]

\[
\text{PS1} \quad \text{precsib}_e[j] = 1 \text{ iff there is an embedding of } T'_j \text{ in } D, \text{ with } j \text{ mapped to a preceding sibling of } e.
\]

Let \( e' \) be the parent of node \( e \) in \( D \). Note that, at any instant in the computation of the predicate checker, while \( e \) is open, we have \( \text{prec}_e[j] = \text{prec}_{e'}[j] \lor \text{desc}_{e'}[j] \), and \( \text{precsib}_e[j] = \text{child}_{e'}[j] \); this is because, when \( e \) is open, the effect of the subtree rooted at \( e \) is not reflected in \( \text{child}_{e'} \) and \( \text{desc}_{e'} \). So, the predicate checker can easily maintain \( \text{prec}_e \) and \( \text{precsib}_e \), without increasing the memory space or runtime.

\section{Conclusions}

We presented an efficient algorithm for evaluating an XPath 1.0 query \( Q \) (involving only \( \text{child} \) and \( \text{descendant} \) 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. It is among the first correct algorithms known for the streaming version that also have a polynomial bound on the memory space and runtime. Also, for some worst case \( Q \) and \( D \), the memory space used by our algorithm matches our lower bound proved in [30]; so, our algorithm uses optimal memory space in the worst case. Also, our algorithm is runtime competitive with the in-memory algorithms, while using much less memory space.

Our algorithm has one drawback, resulting from its use of the predicate checker and stacks: lazy evaluation of predicates, and the consequent delay in outputting/discardng candidates. Our predicate checker \( M \) determines the satisfaction/failure of the predicates at an element \( e' \) only when \( e' \) is the current element. If \( e' \) satisfied/failed a predicate through one of its children, this information would be
reflected at $e'$ only after that child closed. For example, consider the predicate $[b]$. $M$ would determine that $e'$ has satisfied this predicate only after its $b$ child closes, not when the $b$ child opens. Our Last-In-First-Out (LIFO) path stacks use Fact 6.4 that is a consequence of this lazy evaluation. Without Fact 6.4, our path and candidate stacks would not be stacks at all.

The laziness of $M$ can be reduced to some extent as follows: On a \texttt{startElement} event for $e$, $M$ should compute a $(L^T, L^F)$ pair for the current element $e'$ (parent of $e$), based on this event, before pushing $e'$ into the stack $S$ and making $e$ the new current element. Procedure $PStartElement$ should process this $(L^T, L^F)$ pair for $e'$, before pushing $e$ into the appropriate path stacks. Fact 6.4 would still hold in this case. The space and runtime requirements of the algorithm are unchanged. With this modification, $M$ would determine that $e'$ has satisfied the predicate $[b]$ as soon as the $b$ child opens. Now consider the predicate $[b/d]$: $M$ would determine that $e'$ has satisfied this predicate only when its $b$ child (that in turn has a $d$ child) closes, not when the $d$ grandchild opens. Stretching the above modification one step further becomes complicated, as $e'$ is not the current element when its $d$ grandchild opens, and is possibly buried under the $b$ child in some path stacks; so, Fact 6.4 would not hold.
References


APPENDIX

For the case when \( \text{axis}(L_i) = \text{descendant} \), for all steps \( L_i \)

procedure PStartDocument()

Initialize path stacks \( S_1, S_2, \ldots, S_n \) to be empty;
initialize their top pointers \( t_1, t_2, \ldots, t_n \) to nil.

Initialize candidate stacks \( C_1, C_2, \ldots, C_{n-1} \) to empty;
initialize their top pointers \( t'_1, t'_2, \ldots, t'_{n-1} \) to nil.

\( nempty = 0; \) /* \( S_1, \ldots, S_{nempty} \) are the only nonempty path stacks.

\( \text{satUpto} = 0; \) /* We will not show the updating of \( nempty \).

\( \text{satUpto} = 0; \) /* For \( 1 \leq i \leq \text{satUpto} \), \( (\text{top}(S_i).\text{predStatus} = T) \land (\text{top}(S_i).\text{leftPtr} = t_{i-1}) \).

procedure PStartElement(a, event#)

/** Push the new element into appropriate path stacks.

for each \( i \) (\( \text{satUpto} < i \leq \text{nempty} + 1 \)), such that \( a \) matches \( \text{nodeTest}(L_i) \),

in \( \downarrow \) order of \( i \), do

/** No descendant of this new element has opened.

/** So, currently, no element from either \( S_{i+1} \) or \( C_i \) needs to point to this element.

if \( (i = n) \)
then if \( \text{predicate}(L_n) = \text{nil} \)
then if \( \text{satUpto} = n - 1 \)
then add the element \( (a, \text{event#}) \) to the output buffer
else push(\( S_n, (a, \text{event#}, \text{True}, t_{n-1}) \))
else push(\( S_n, (a, \text{event#}, \text{Unknown}, t_{n-1}) \))
else if \( \text{predicate}(L_i) = \text{nil} \)
then notRedPush(\( i, (a, \text{event#}, \text{True}, t_{i-1}) \))
if \( i = \text{satUpto} + 1 \) then \( \text{satUpto} + +; \)
else notRedPush(\( i, (a, \text{event#}, \text{Unknown}, t_{i-1}) \))

procedure notRedPush(i, R)

/** Push record \( R \) into path stack \( S_i \) iff it is not redundant.

if \( \text{not} ((\text{top}(S_i).\text{predStatus} = T) \land (\text{top}(S_i).\text{leftPtr} = t_{i-1})) \)
then push(\( S_i, R \))

procedure PText(s)

/** Process a \text{t}ext event

\( e \leftarrow \text{current element} \) /* Parent of the \text{t}ext \text{node}

Obtain \( (L^T, L^F) \) from the predicate checker, for \( e \)

/** Modify corresponding path and candidate stacks:
\( \text{deleteFalse}(e, L^F); \quad \text{setTrue}(e, L^T) \)
procedure deleteFalse(e, \(L^F\))  
/** e fails the predicates in steps \(L_i \in L^F\); 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 ≤ nempty and top(S_i).predStatus = Unknown.  
  if ((i < n) and (topBunch(C_i).pathPtr = \(t_i\))) then bunch ← popBunch(C_i)  
  else bunch ← \(\phi\);  
  pop(S_i)  
  if ((\(t_i \neq nil\)) and (bunch \(\neq \phi\))) then pushBunch(i, bunch)  

procedure pushBunch(i, bunch)  
/** bunch is a set of candidates for output;  
/** push bunch on \(C_i\) after, if necessary, merging with topBunch(C_i).  
  if topBunch(C_i).pathPtr = \(t_i\) then bunch′ ← popBunch(C_i)  
  else bunch′ ← \(\phi\);  
  newbunch ← bunch \(\cup\) bunch′  
  newbunch.pathPtr = \(t_i\);  
  push(C_i, newbunch)  

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 \(top(S_i) = R_i(e)\), in ↑ order of i, do  
/** satUpto < i ≤ nempty and top(S_i).predStatus = Unknown.  
  top(S_i).predStatus ← True  
  if ((i = n) and (satUpto = n − 1) and (top(S_n).leftPtr = \(t_{n-1}\))) then pop(top(S_n)); output the corresponding element  
  else if ((i = satUpto + 1) ∧ (top(S_i).leftPtr = \(t_{i-1}\))) then satUpto++;  
      if topBunch(C_i).pathPtr = \(t_i\) then pop topBunch(C_i);  
      output the corresponding elements  

procedure deleteTrue(e)  
/** e is closing; delete e (with predStatus = True) from all path stacks  
for each i such that \(top(S_i) = R_i(e)\), in ↑ order of i, do  
/** 1 ≤ i ≤ 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, (\(\tau(e)\), event#, \(t_{n-1}\)))  
  else if topBunch(C_i).pathPtr = \(t_i\) then bunch ← popBunch(C_i); pushBunch(i − 1, bunch)  
  pop(S_i);  
  if i = satUpto then satUpto − −
procedure PEndElement(a)
/** Process an endElement event

  e ← document node whose endElement event was seen

  Obtain \((L^T, L^F)\) from the predicate checker, for \(e\)

  /** \(e\) is closing; \(L^T \cup L^F\) contains all loc. steps for which predStatus was Unknown

  /** Modify corresponding path and candidate stacks:
  deleteFalse\((e, L^F)\); setTrue\((e, L^T)\); deleteTrue\((e)\)

  \(e' ← parent(e)\)

  /** \(e'\) is obtained by popping the stack of the predicate checker;
  /** it becomes the new current element.

  if \(\tau(e') \neq /\) then /* not reached document end

    /** The endElement event for \(e\) makes some predicates \(T/F\) at \(e'\).
    Obtain \((L^T, L^F)\) from predicate checker, for \(e'\)

    /** Modify corresponding path and candidate stacks:
    deleteFalse\((e', L^F)\); setTrue\((e', L^T)\)