Processing XPath queries with forward and downward axes over XML streams

Makoto Onizuka
NTT Cyber Space Laboratories, NTT Corporation
1-1 Hikari-no-oka, Yokosuka, Japan
onizuka.makoto@lab.ntt.co.jp

ABSTRACT

We propose an XPath processing algorithm that efficiently evaluates XPath queries in $XP^{\{\downarrow,\rightarrow,*,[]\}}$ over XML streams. An XPath query is expressed with axes, which are binary relations between nodes in XML streams: '↓' identifies the child/descendant axes and '→' indicates the following/following-sibling axes. The proposed algorithm evaluates XPath queries within one XML parsing pass and outputs the fragments found in XML streams as the query results. The difficulty of $XP^{\{\downarrow,\rightarrow,*,[]\}}$ evaluation lies in establishing dynamic scope control for the following/following-sibling axes. The algorithm uses double-layered non-deterministic finite automata (NFA) to resolve this issue. First layer NFA is compiled from XPath queries and is able to evaluate sub-queries in $XP^{\{\downarrow,\rightarrow,*\}}$. Second layer NFA handles predicate parts. It is dynamically maintained during XML parsing: a state is constructed from a pair of the corresponding state in the first layer automaton and the currently parsed node in the XML stream. Layered NFA achieves $O(|D||Q|)$ time complexity by introducing a *state sharing* technique, which avoids the exponential growth in the state size of Layered NFA by eliminating redundant transitions. We validate the efficiency of the algorithm through empirical experiments and show that Layered NFA is up to four times faster, and twice as fast on average, than existing algorithms.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

General Terms

Algorithms, Languages, Performance

Keywords

XML stream, XPath, Automata

1. INTRODUCTION

While databases process volatile queries over persistent data, the emerging technology of stream processing aims to handle persistent queries against incoming volatile data streams. Of particular interest, XML streams are expressed in a platform-neutral format, so they are suitable for data archiving or publication on the web. Typical application areas of XML streams are scientific data (astronomical data, protein sequence data), control streams (sensor stream, air traffic, ATM monitoring), and data archiving for analysis (click streams on the Internet, personal log, linguistic TreeBank database).

Several query evaluation algorithms for XPath queries over XML streams have been proposed for the above application areas, especially *full-fledged* query evaluation[1] aims to output stream elements selected by a query. Our goal is the efficient full-fledged evaluation of $XP^{\{\downarrow,\rightarrow,*,[]\}}$ which denotes the XPath fragment that includes downward axes (child, descendant) and forward axes (following, following-sibling), wildcard node-test, and optional predicates. The XML stream processing algorithms proposed in [15] have $O(|D||Q|)$ time complexity, and they use a multi-stack framework for $XP^{\{\downarrow,*,[]\}}$ evaluation, where $|D|$ is the XML stream size and $|Q|$ is the query size. XSQ [26] is based on hierarchical NFA augmented with buffers for $XP^{\{\downarrow,[]\}}$ evaluation. TwigM [8] efficiently evaluates $XP^{\{\downarrow,*,[]\}}$ queries by employing twigStack [4]. XAOS [3] evaluates $XP^{\{\downarrow,\uparrow,*,[]\}}$ queries by converting '↑' (parent/ancestor) axes into downward constraints so they can be handled in stream processing. xmltk [16] uses a DFA-based algorithm for $XP^{\{\downarrow,*\}}$.

SPEX [24] is a state of the art algorithm for evaluating XPath queries in $XP^{\{\downarrow,\rightarrow,*,[]\}}$; as already noted, other existing algorithms only support strict subclasses of $XP^{\{\downarrow,\rightarrow,*,[]\}}$. SPEX uses a network of independent transducers, each of which is compiled from a step in a query. A transducer accepts as input XML streams and outputs the XML streams annotated with marks on selected nodes. SPEX requires every transducer to react to SAX events in XML streams. A key problem of SPEX is its inefficient predicate evaluation. SPEX independently evaluates each predicate and the trunk part of the query, and then merges the intermediate results produced by single evaluations. Thus, the intermediate results output from the transducers tend to be so large that overall performance is degraded. Consider the query `/dblp/article[year>1990][Category='Algorithm']/title`. SPEX evaluates each predicate `[year>1990]`, `[Category='Algorithm']`, and trunk part `title` independently, and merges their results. Efficient predicate evaluation is crucial since many practical applications make extensive use of complex predicates.

For XPath queries in $XP^{\{\downarrow,\rightarrow,*,[]\}}$, we propose Layered NFA, an XPath processing algorithm that efficiently evaluates queries in

[1]In contrast, *filtering* is the function of outputting a bit indicating whether a query selects any nodes from the stream.

$O(|D||Q|)$ time. In contrast to SPEX, which processes SAX events in XML streams at every step (one step at a time) in a query, Layered NFA evaluates the whole query at every SAX event within one XML parsing pass (one SAX event at a time). The advantages of Layered NFA are 1) it generates smaller intermediate results than SPEX because the whole query is evaluated at every SAX event, and 2) it makes non-deterministic transitions for each incoming SAX event, fewer actions than required by the transducer network of SPEX.

The SAX-event-at-a-time approach is not new, since it was adopted in earlier algorithms [3, 15, 8, 26, 16] for $XP^{\{\downarrow,*,[]\}}$ queries. Here we have a question: can we simply extend those algorithms so as to support forward axes over XML streams? The answer is no. The problem raised by $XP^{\{\downarrow,\rightarrow,*,[]\}}$ evaluation lies in establishing dynamic scope control for queries using the following/following-sibling axes. Intuitively, a step scope covers all nodes that can satisfy the axis of the step being evaluated at the current node parsed in the XML stream. The scope of the child/descendant axes is static and symmetric with regard to the start and end points of the scope, however this doesn't hold for the following/following-sibling axes.

//inproceedings[section[title='Overview']/following::section]

Figure 1: XPath query example

```
 1:<dblp>
 2: <inproceedings mdate="2008-06-09">
 3:  ...
 4:  <title>Layered NFA</title>
 5:  <year>2008</year>
 6:  <section>
 7:   <title>Introduction</title>
 8:   ...
 9:  </section>
10:  <section>
11:   <title>Overview</title>
12:   ....
13:  </section>
14:  <section>
15:   <title>Algorithm</title>
16:   ....
17:  </section>
18:  ...
19: </inproceedings>
20: <article mdate="2002-01-23">
21:  ...
22:</dblp>
```

Figure 2: XML stream example

To illustrate the requirement of dynamic scope control, consider the XML stream in Fig.2 and the XPath query in Fig.1, which will be used as a running example hereafter. The query is intended to select inproceedings with overview section and following sections. The scope of step `title` (with implicit child axis) within the inner predicate starts after the opening tags of its context nodes `section` (lines 6,10,14 in Fig.2) and ends before the corresponding closing tags (lines 9,13,17). In contrast, the scope of `following::section` starts after the same opening tags but its end point depends on the predicate results of the step that matches a context node. In this example, `section` is the step that matches a context node of `following::section` and it has the predicate `[title='Overview']`. When it is satisfied after receiving the SAX event characters(`Overview`) (line 11), the scope, starting from the second `section` node, reaches the end of the stream. Otherwise, the scope ends before the closing tags of the context nodes `section` (lines 9, 17). This observation reveals that dynamic scope control is required for processing forward axes.

We can consider several approaches in designing efficient XPath processing algorithms over XML streams. We first considered the use of a fully dynamic data structure based on a data-driven query rewrite scheme against SAX events. The idea is that queries on the current node are converted into equivalent queries on the following nodes and this is repeated until the end of the stream[2]. This approach is concise and gives a fundamental background of evaluating XPath queries in $XP^{\{\downarrow,\rightarrow,*,[]\}}$, however its time complexity is high and preliminary experiments showed that it was indeed expensive. On the other hand, full compiler-based approaches are likely to suffer from exponential space complexity. As an example of this case, XSQ [26] requires $2^{|Q|-1}$ NFAs, since states are generated from all possible combinations of predicate results.

To overcome this problem, we introduce Layered NFA, an efficient XPath processing algorithm that balances the performance advantage of the compiler-based approach against the overhead of space complexity. Layered NFA uses a hybrid structure: a first layer NFA that is compiled from an XPath query and a second layer NFA that is dynamically maintained during XML parsing, see Fig.4 for an overview. The first layer NFA evaluates sub-queries in $XP^{\{\downarrow,\rightarrow,*\}}$ and the second layer NFA handles predicate parts. A state in the second layer NFA plays three roles. 1) a state is constructed from the corresponding state in the first layer NFA and represents which steps have been matched the received SAX events, 2) a state is also annotated with a node that matches with a step with predicates or the target step to be answered for the query. A node matched with a step with predicates records which predicates have been satisfied, whereas a node matched with the target step is buffered as a candidate node, and 3) a state keeps the scope status for dynamic scope control.

1.1 Contributions

Contributions are summarized as follows:

- We introduce Layered NFA, an efficient XML stream processing algorithm for queries in $XP^{\{\downarrow,\rightarrow,*,[]\}}$. The algorithm evaluates XPath queries within one XML parsing pass and outputs the matched fragments in the XML stream. Layered NFA uses double-layered NFA. First layer NFA is compiled from XPath queries and is able to evaluate sub-queries in $XP^{\{\downarrow,\rightarrow,*\}}$. Second layer NFA is used for predicate processing including dynamic scope control.
- Layered NFA holds time complexity to $O(|D||Q|)$ by using a *state sharing* technique, which avoids the exponential growth in the state size of Layered NFA by eliminating redundant transitions to the states that are constructed from the same state in the first layer NFA.
- Layered NFA prunes states in the second layer NFA for more efficient processing. When a predicate becomes satisfied, the related states can be removed because of the existential semantics of XPath predicates.

[2]In contrast, existing query rewrite schemes [14, 23, 27] are based on the query-driven approach for XML databases. For more detail see Section 6.

- We consider the problem of query rewrite for XML stream processing. In this query rewrite scheme, queries on the current node are converted into equivalent queries on the following nodes and this is repeated until the end of the stream.
- We describe experiments on various types of XPath queries over two real XML streams: Protein and TreeBank. The results show that Layered NFA is up to four times faster, and twice as fast on average, than SPEX, and is comparable to XSQ for $XP^{\{\downarrow,[]\}}$ queries.

The rest of this paper is as follows. Section 2 presents the XML data and query models. The query rewrite scheme for XML stream processing is introduced in Section 3. Layered NFA is introduced and analyzed in Section 4. Section 5 reports the results of experiments. Section 6 addresses related work and Section 7 concludes this paper.

2. DATA AND QUERY MODELS

XML Data model We model XML data as ordered trees with nodes named from an infinite alphabet Σ. The symbols in Σ represent the element names, attribute names, and text/attribute values that can occur in XML data. XML data is parsed in depth first order to output a sequence of SAX events; startDocument, endDocument, startElement(tag), endElement(tag), and characters(value) where tag is the name of the current element node being parsed, and value is the text value of the current text node. The sequence of SAX events is input to XPath algorithms for query evaluation.

XPath query $XP^{\{\downarrow,\rightarrow,*,[]\}}$ consists of expressions given by the following grammar:

$$\begin{array}{rcl} Q & ::= & /step\ (/step)^* \\ step & ::= & axis :: node{-}test\ ([predicate])^* \\ axis & ::= & \texttt{self} \mid \texttt{child} \mid \texttt{descendant} \mid \texttt{following} \mid \\ & & \texttt{following-sibling} \\ node{-}test & ::= & name \mid * \mid \texttt{text}() \\ predicate & ::= & Q \mid Q\ opr\ literal \mid func(Q, literal) \\ func & ::= & \texttt{starts-with} \mid \texttt{contains} \\ opr & ::= & > \mid >= \mid = \mid < \mid <= \mid != \end{array}$$

An XPath query is a sequence of steps, each of which consists of axis (binary relation on nodes in XML streams), node-test (wildcard '*' or node name), and optional predicates (boolean formula over paths). We restrict the grammar by preventing disjunctive predicates, because we can extend both the query rewrite scheme and Layered NFA easily to support them. / and // are abbreviated forms of /child and /descendant, respectively. *trunk part* denotes the query obtained by removing predicates from an original query and *trunk step* denotes a step in the *trunk part*. *target* denotes the last step in the *trunk part*. We don't treat the attribute axis explicitly since it can be handled in a way similar to that used for the child axis.

An overview of XPath query evaluation for XML streams is as follows. Let r be the root node of an XML stream, and Q be a query in the form of $/step_1/.../step_i/.../step_{|Q|}$. We use step evaluation function $eval_{step_i}(nodes)$ that outputs a set of nodes, each satisfying the axis, node-test, and optional predicates of $step_i$ for some node in $nodes$, namely context nodes. The XPath evaluation starts from $eval_{step_1}(\{r\})$ by treating r as the context node of $step_1$. Then, during XML parsing, the result nodes of $eval_{step_i}(X)$ for the context nodes X are used as the context nodes of the next step $step_{i+1}$. For the target step $step_{|Q|}$, $eval_{step_{|Q|}}(X)$ outputs the target nodes of the query. However, some steps may contain predicates whose results are determined after receiving the SAX events of the target nodes. This observation demands that we may buffer the target nodes[3] until we determine the predicate results. Moreover, as we saw in Section 1, the scope of the following/following-sibling axes dynamically changes.

Formally, we first define node *effectiveness*, and then introduce step/path scope. We define trunk($step_i$) as the trunk step of $step_i$, and preds($step_i$) as a set of all top level predicates of $step_i$[4]. For context node x, we also define $trunkEval(t, x)$ that outputs a set of nodes each satisfying trunk step t, and $prEval(p, x)$ that returns the result of predicate p.

Definition 2.1 (node effectiveness) *Let Q be an XPath query in the same form described above and x be a context node of $step_i$ in Q. $n \in trunkEval(trunk(step_i), x)$ is effective if:*
for i=|Q|, $\forall p \in preds(step_i)$. $prEval(p, n)$=true, and
for i<|Q|, $\forall p \in preds(step_i)$. $prEval(p, n)$=true and $\exists n' \in trunkEval(trunk(step_{i+1}), n)$ is effective.

Notice that, the effectiveness of node n for $step_i$ is determined by the predicate results of $step_i$ and the effectiveness of the nodes in $trunkEval(trunk(step_{i+1}), n)$, thus it is determined by propagating predicate results in a bottom up manner during stream processing. We also note that $prEval(p, n)$=true requires n to be effective for the query specified in predicate p[5].

The node effectiveness may be terminated by predicate failure as follows. In the following, we use the notation predsUp(Q) to indicate the set of all top predicates of the step in which Q is used in a top predicate. For simplicity, we also use $step_i(X)$ to indicate

$$\bigcup_{n \in X} trunkEval(trunk(step_i), n)$$

.

Definition 2.2 (effectiveness termination) *Let Q be an XPath query in the same form described above, x be a context node of $step_i$ in Q, and some $n \in trunkEval(trunk(step_i), x)$ be an effective node. The effectiveness of n is terminated if:*
for i>1, $\exists p \in preds(step_{i-1})$. $prEval(p, x)$=false, and
for i=1, $\exists p \in predsUp(Q)$. $prEval(p, x)$=false.
Accordingly, the effectiveness of all effective nodes $\in step_{i+1}(\{n\}) \cup step_{i+2}(step_{i+1}(\{n\})) \cup ... \cup step_{|Q|}(...step_{i+1}(\{n\})...)$ is terminated.

As an example, we return to the running example in Fig.1 and Fig.2. Let x be `inproceedings` node in the XML stream, which is a context node of $step_1$ in Q = `section[title='Overview']/following::section`, and n be the second `section` node in the XML stream. When we receive characters(`Overview`) event in line 11, n, which is in $trunkEval(trunk(step_1), x)$, becomes *effective*, because the only predicate `[title ='Overview']` in preds($step_1$) becomes satisfied and a node in $trunkEval(trunk$

[3]We call buffered target nodes candidate nodes.
[4]preds($step_i$) does not include nested predicates.
[5]This discussion does not hold if we extend the grammar to permit negation.

$$
\begin{array}{rcl}
\mathrm{S(x, "")} & = & \{\mathrm{x}\} \\
\mathrm{S(x, self :: n/p)} & = & \mathrm{if\ match(x, n)\ then\ S(x, p)\ else\ \{\}} \\
\mathrm{S(x, child :: n/p)} & = & \mathrm{S(first{-}child(x), self :: n/p \mid following{-}sibling :: n/p)} \\
\mathrm{S(x, descendant :: n/p)} & = & \mathrm{S(first{-}child(x), self :: n/p \mid descendant :: n/p \mid} \\
 & & \quad \mathrm{descendant{-}following{-}sibling :: n/p)} \\
\mathrm{S(x, following{-}sibling :: n/p)} & = & \mathrm{S(first{-}sibling(x), self :: n/p \mid following{-}sibling :: n/p)} \\
\mathrm{S(x, following :: n/p)} & = & \mathrm{S(first{-}following(x), self :: n/p \mid descendant :: n/p \mid following :: n/p)} \\
\mathrm{S(x, descendant{-}following{-}sibling :: n/p)} & = & \mathrm{S(first{-}sibling(x), self :: n/p \mid} \\
 & & \quad \mathrm{descendant :: n/p \mid descendant{-}following{-}sibling :: n/p)} \\
\mathrm{S(x, p1 \mid p2)} & = & \mathrm{S(x, p1) \cup S(x, p2)} \\
\mathrm{S(x, [p1]...[pn]/p)} & = & \mathrm{if\ S(x, p1) \neq \{\}\ and\ ...\ and\ S(x, pn) \neq \{\}\ then\ S(x, p)\ else\ \{\}}
\end{array}
$$

Figure 3: Query Rewrite for $XP^{\{\downarrow,\rightarrow,*,[]\}}$

$(step_2), n)$ is effective. In addition, the effectiveness of node n will not be terminated, since the only predicate `[section[title=' Overview']/following::section]` $\in$ predsUp(Q) is satisfied by the third `section` node in the XML stream.

Next, we define the step scope according to axis semantics as follows. Intuitively, the step scope for a context node is the period during XML parsing when the context node may become effective.

Definition 2.3 (step scope) *Let Q be an XPath query in the same form described above, x be a context node of $step_i$, parent(x) be the parent node of x, and scope($step_i$, x) be $step_i$'s scope for x denoted by {start, end}. scope($step_i$, x) is {startElement(x), endElement(x)}, if $step_i$'s axis is child/descendant or x is not effective. If x is effective, scope($step_i$, x) is {startElement(x), endElement(parent(x))} for the following-sibling axis, and {startElement (x), end of stream} for the following axis.*

In the running example, let n be the second `section` again. When we receive characters(`Overview`) event, n becomes *effective*, so scope(`following::section`, n) = {startElement(n), end of stream}.

Finally, the path scope is defined as follows.

Definition 2.4 (path scope) *Let Q be an XPath query in the same form described above, and x be a context node of $step_1$ in Q. The start point of Q's path scope for x is that of scope($step_1$, x). The end point of Q's path scope for x is the end of the stream if some $step_i$ in Q contains the following axis and x is effective, otherwise it is the end point of scope($step_1$, x).*

3. QUERY REWRITE

We introduce a query rewrite scheme for queries in $XP^{\{\downarrow,\rightarrow,*,[]\}}$ to give a fundamental background of query evaluation. We use $\mathrm{S}(x, Q)$ to denote a query evaluation function where x is a context node in the XML stream and Q is a query in $XP^{\{\downarrow,\rightarrow,*,[]\}}$. The query rewrite scheme in Fig.3 rewrites queries on the current node into queries on the following nodes; therefore, query rewrite can be continuously performed on a sequence of SAX events. A query is rewritten as follows. We start from $\mathrm{S}(r, Q)$ where r is the root node of the XML stream. First, we rewrite Q and assign the rewritten queries to the following nodes. Then, during XML parsing, for a startElement event, we collect the rewritten queries assigned to current node x from the preceding nodes and apply continuously the query rewrite scheme to those queries. Finally, we complete the evaluation at the end of the stream and obtain the target nodes.

In the query rewrite scheme in Fig.3, we write first-child(x), first-sibling(x), first-following(x) for the first child node of x, the first sibling node of x, and the first following node of x, respectively. We introduce an additional axis, descendant-following-sibling(x) that specifies the descendant nodes of the following sibling nodes of x. Due to space limitations, we only comment on the query rewrite scheme for the following axis. If the following axis is to be evaluated on context node x, the `title` element in line 7 in Fig.2, it is rewritten to a union of self, descendant, and following axes on first-following(x), which is the `section` element in line 10.

A query with predicates is handled as depicted in the last line in Fig.3. Intuitively, if all predicates p1,...,pn match some nodes, the result of trunk part p becomes the result of the query. However, this query rewrite is different from others: the condition can be evaluated at following nodes of x. In contrast, the condition of the query rewrite in the second line can be evaluated at the current context node, x. Here we face an issue of predicate processing: until when do we have to keep on checking if a predicate is satisfied? The path scope helps determine when predicate fails. As an example, if the end point of a predicate path scope is that of its first step and the axis of the step is child, we can determine the predicate result at the endElement event of the context node of the step.

Despite the conciseness of the query rewrite scheme, there are two problems in this approach. First, its time complexity is high, because the number of intermediate queries divided by $|$ linearly increases with query size. Moreover preliminary experiments showed that the cost of the query rewrite scheme was too expensive even for queries without predicates. Second, it is not obvious how to implement scope control, especially for queries with predicates. These problems motivated the concept of Layered NFA. Against the first problem, Layered NFA uses a hybrid structure: a first layer NFA that is compiled from an XPath query and a second layer NFA that is dynamically maintained during XML parsing. To deal with the second problem, Layered NFA dynamically controls the step/path scope at the second layer NFA.

4. LAYERED NFA

Layered NFA consists of four data structures: the query tree, two NFAs, and the context node tree. An example is shown in Fig.4, and is described in the following sections. A given query is parsed to form a query tree, and then compiled into an NFA in the first layer. The first layer NFA is able to evaluate sub-queries in $XP^{\{\downarrow,\rightarrow,*\}}$,

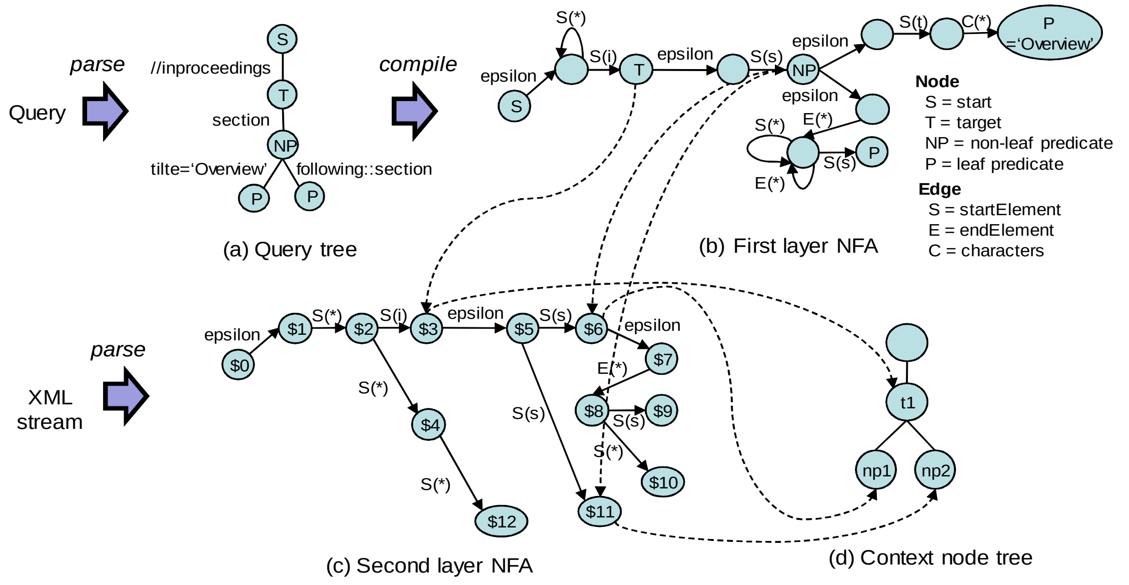

Figure 4: Layered NFA of the XPath query in Fig.1 : (Only the first letter of the element names is used in SAX events. The snapshot shown is that after receiving startElement of the 3rd `section` in Fig.2)

and it is used by the second layer NFA. The second layer NFA handles predicate parts. It is dynamically maintained during XML parsing: a state refers to both the corresponding state in the first layer NFA and a matched node in the XML stream. A referred state in the first layer NFA is used for $XP^{\{\downarrow,\rightarrow,*\}}$ processing and the matched node is used as the context node for predicate processing or is buffered as a candidate node. The context node tree of a query is a tree of nodes that are matched with steps with predicates or the target step in the query. The tree is maintained while the second layer NFA is being processed.

4.1 Query tree

In the query tree, an edge represents a sub-query in $XP^{\{\downarrow,\rightarrow,*\}}$ and a node represents a branch connecting the sub-queries of predicate/trunk parts: a node is labeled with its incoming edge type, which may be either target (T), non-leaf predicate (NP), or leaf predicate (P). The root node is labeled with start (S).

Fig.4 (a) shows the query tree for the query in Fig.1. The query is decomposed into sub-queries, `//inproceedings`, `section`, `title='Overview'`, and `following::section` to form the tree.

4.2 First layer NFA

The first layer NFA is compiled from queries by the NFA encoding rules in Fig.5. The label `s` indicates the initial state and `t` indicates the terminal state. S(`a`), E(`a`), C(*) are abbreviations of startElement(`a`), endElement(`a`), and characters(*), respectively, where `a` is node name and * is wildcard.

- The self axis is encoded by an epsilon transition.
- We follow a popular method for converting /`a`, //`a` into the NFA [9, 16], see (a), (b) in Fig.5.
- following-sibling::`a` is defined to select the following sibling nodes of the context node; they are tagged as '`a`'. It is encoded into the NFA in Fig.5 (c). First, we move up to the

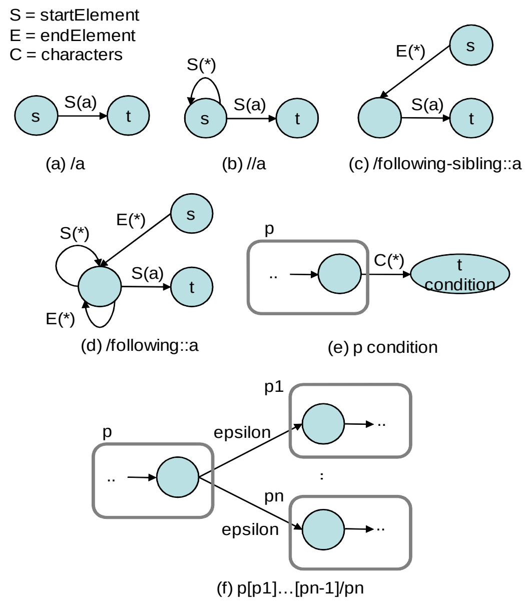

Figure 5: NFA encoding rules

parent node of the context node by E(*) transition, and then select nodes tagged as '`a`' by S(`a`) transition.

- following::`a` is defined to select all nodes tagged as '`a`' that appear after the context node except the descendant nodes. It is encoded into the NFA in Fig.5 (d). The difference from the following-sibling::`a` encoding rule is that two transitions are added: E(*) is used to locate all ancestor nodes and S(*) is used to select all of their descendant nodes.
- The condition used in a predicate is encoded into the NFA in Fig.5 (e). If `condition` contains operator/function and literal value, they are stored in the terminal state and checked after the transition to the state by characters(value) event.

Algorithm 1 startElement(*tag*)

```
Require: the set of current states currentStateSet, the state stack
    stateStack, input SAX event startElement(tag)
Ensure: updated currentStateSet, stateStack, the context node tree
 1: hitNodes = {}; nextStateSet = {};
 2: for all current $\in$ currentStateSet do
 3:     (state, context) = current;
        // current is a state in the 2nd layer NFA
        // state is a state in the 1st layer NFA
 4:     if noOutEdges(state) or noSuccessfulTransition(state) or
        isSinkState(current) then
 5:         sinkDown(current);
 6:         nextStateSet += current; continue;
 7:     end if
 8:     for all nextState $\in \delta$(state, startElement(tag)) do
 9:         if label(state)==('T' or 'NP') then
10:             context = new contextNode(tag);
11:             addChildInContextTree(context);
12:         else if label(state)==('P') then
13:             setPredicateResult(context);
14:             hitNodes += propagate&getHitNode();
15:         end if
16:         next=(nextState, context);
            // nextState is a state in the 1st layer NFA
            // next is a state in the 2nd layer NFA
17:         nextStateSet += next;
18:     end for
19: end for
20: stateStack.push(currentStateSet);
21: currentStateSet = nextStateSet;
22: flush(hitNodes);
23: removeRelatedStatesFrom(hitNodes, currentStateSet);
```

- When a query contains predicates, `p[p1]...[pn-1]/pn`, there needs to be epsilon transitions between the last state of the NFA obtained from path `p` and the first states of NFAs obtained from the predicates `p1,...,pn-1` and the trunk `pn`. Fig.5 (f) shows this encoding rule.

Fig.4 (b) shows the first layer NFA for the query in Fig.1. The states are labeled with the same name as the corresponding nodes in the query tree, start (S), target (T), non-leaf predicate (NP) and leaf predicate (P).

4.3 Second layer NFA & Context node tree

The transitions in the second layer NFA, which are based on the first layer NFA, drive the maintenance of the context node tree: node construction/destruction, predicate result propagation, candidate node buffering, and dynamic scope control.

Alg.1 and Alg.2 show the algorithms of startElement and endElement events, respectively. Layered NFA uses the state stack (*stateStack*) to maintain the current states (*currentStateSet*) for startElement and endElement events throughout the XML stream. It flushes out the buffered target nodes (*hitNodes*) to the user when they match a given query; the effectiveness of their context nodes is determined not to be terminated. The implementation of characters event is similar to the combination of startElement and endElement accompanied with condition evaluation, so we omit its description. We will refer to the corresponding parts of Alg.1 and Alg.2 in the following descriptions.

Algorithm 2 endElement(*tag*)

```
Require: the set of current states currentStateSet, the state stack
    stateStack, input SAX event endElement(tag)
Ensure: updated currentStateSet, stateStack, the context node tree
 1: hitNodes = {}; nextStateSet = {};
 2: for all current $\in$ currentStateSet do
 3:     (state, context) = current;
        // current is a state in the 2nd layer NFA
        // state is a state in the 1st layer NFA
 4:     if isSinkState(current) then
 5:         sinkUp(current);
 6:         nextStateSet += current; continue;
 7:     end if
 8:     if label(state)==('T') then
 9:         hitNodes += propagate&getHitNode();
10:     end if
11:     if isEffectivenessTerminate() then
12:         remove(context, current);
13:     end if
14:     for all nextState $\in \delta$(state, endElement(tag)) do
15:         next=(nextState, context);
            // nextState is a state in the 1st layer NFA
            // next is a state in the 2nd layer NFA
16:         nextStateSet += next;
17:     end for
18: end for
19: currentStateSet = stateStack.pop() + nextStateSet;
20: flush(hitNodes);
21: removeRelatedStatesFrom(hitNodes, currentStateSet);
```

We define below the data structure of state in the second layer NFA (*NFA2*).

Definition 4.1 *A state in the second layer NFA is a pair (s, n) where s is the corresponding state in the first layer NFA. If s is labeled with target (T) or non-leaf predicate (NP), the current node in the XML stream is set to n. Otherwise, the node of the previous state that has a transition to this state is set to n.*

In the following, we first describe details of state construction and transition, and then describe the behaviors of target step and predicate processing.

State construction and transition
Let the first layer NFA be (Σ, S, s_0, δ) for input alphabet Σ, set of states S, initial state s_0, and state transition function δ. Let n_0 and N be the root and a set of nodes in the XML stream, respectively. The initial state of *NFA2* is defined as (s_0, n_0). The following states are dynamically constructed as demanded by SAX events. Let (s_i, n_i) be a current state of *NFA2*. For a startElement(tag) event, state (s_j, n_j) is constructed and is set as a next state where s_j is in $\delta(s_i,$ startElement(tag)) and n_j is set as defined in Def.4.1: If s_j is labeled with target (T) or non-leaf predicate (NP), the current node of the startElement(tag) event is set to n_j. Otherwise, n_i is set to n_j. For preparing the next SAX event, the current states are pushed on the state stack and the next states are set to the current states. For an endElement(tag) event, state (s_j, n_j) is constructed and is set as a next state where s_j is in $\delta(s_i,$ endElement(tag)) and n_j is set as defined in Def.4.1. For preparing the next SAX event, the states are popped from the state stack and added to the next states.

The above procedures for startElement(tag) and endElement(tag) events are implemented in Alg.1 and Alg.2, respectively. A state is obtained by δ function (line 8) and then added as a next state (lines 16-17). The current states are pushed on the state stack (line 20). In addition, sink state operations, sinkDown and sinkUp, are present (line 5 Alg.1 and line 5 Alg.2). A state is a sink state if and only if there are no out-going transitions [18]. For efficient sink state implementation, we mark a state as a sink state with sink depth to avoid constructing sink states and transitions to those states. Current state (s, n) becomes a sink state if s has no out-going transitions or it has no successful transition (lines 4-5 Alg.1). The states labeled with P in (b) Fig.4 are examples of s without out-going transitions. When a current state is state \$5 in (c) Fig.4 and it receives SAX events except S(s), \$5 becomes a sink state because it has no successful transition. A state continues as a sink state by incrementing the sink depth for startElement events.

With regard to endElement(tag) event (Alg.2), there are only wildcard E(*) transitions in the NFA encoding rule Fig.5 so there is no need for a transition check to obtain a next state by δ function (line 14). The states popped from the state stack are also added as the next states (line 19). The sink state operations occupy lines 4-6. The sinkUp function (line 5) decrements the sink depth for endElement events.

Context node tree construction The context node tree of a query is a tree of nodes that are matched with steps with predicates or with the target step in the query. The tree is maintained while the second layer NFA is being processed. Assume we have arrived at state (s, n) by a startElement event where s is a state in the first layer NFA labeled with T (target) or NP (non-leaf predicate). We then construct a new node for n, put it as a child of the current node in the tree, and set it as the current node for subsequent processing. A context node tree plays two important roles. 1) a node matched with a step with predicates (NP) records the results indicating which predicates have been satisfied, whereas a node matched with target (T) is buffered as a candidate node, and 2) a node keeps the status of the path scope of predicate/trunk parts. It is maintained as described in Def.2.3-2.4 and is initially assigned to a node as follows: if some step in the path is the following axis, the status is set to `following`, otherwise the status is set to the first step's axis of the path.

Fig.4 (d) shows a context node tree. `t1` is a matched node of `//inproceedings` path and is used as a candidate node as well as a context node of `section` path. `np1,np2` are matched nodes of `section` path and used as context nodes for `title='Overview'` and `following::section` paths. In addtion, `np1`, `np2` record predicate results.

A new node is constructed by the contextNode constructor (line 10 in Alg.1) and it is put into the context node tree by the addChildInContextTree function (line 11 in Alg.1).

Target step processing Recall that a predicate result may be determined after receiving the SAX events of the target nodes. To handle this case, we need to buffer the candidate nodes, and then to propagate those candidate nodes and the predicate results upward in the context node tree, so that we can check if context nodes of predicates become effective. Assume we have arrived at state (s, n) by a startElement event where s is a state in the first layer NFA labeled with T (target). This state indicates that the target step in the query matches the SAX event, so current node n is buffered. When n becomes effective, we propagate the predicate results stored at n and the buffered candidate node n up to the parent node of n in the context node tree, and repeat this process. If the propagation reaches the first branching node, that is the effectiveness of n is determined not to be terminated, the buffered candidate node n is flushed to the user. On the other hand, n is removed from the context node tree, when the effectiveness of n is terminated.

For a startElement(tag) event (Alg.1), when a current state is constructed from a T-labeled state ($state$) in the first layer NFA (line 9), we start buffering the current node in the XML stream as a candidate node by putting it into the context node tree (lines 10-11). When some predicate is satisfied by a startElement event, several nodes may become effective, so the propagate&getHitNode function propagates predicate results and the candidate nodes up in the context node tree (lines 12-14). This propagation also occurs in endElement events (lines 8-9 Alg.2). In both cases, if the propagation reaches the first branching node in the context node tree, the candidate nodes are flushed out to the user. (line 22 Alg.1 and line 20 Alg.2).

Predicate processing The predicate processing is similar to the target step processing. The difference is that, instead of buffering candidate nodes in the context node tree, the context nodes of predicates are put into the context node tree and predicate results are recorded in those nodes. Assume we have arrived at state (s, n) by a startElement event where s is a P-labeled state. This state indicates that a predicate is satisfied by the SAX event, so the predicate result is put into n. When n becomes effective, we perform the same propagation procedure as described in the target step processing.

For a startElement(tag) event (Alg.1), when a current state is constructed from an NP-labeled state ($state$) (line 9), we construct a context node and put it into the context node tree for recording predicate results (lines 10-11). Else, if a current state is constructed from a P-labeled state, it indicates that a predicate is satisfied by the SAX event, so the result is put into the context node and we then check and propagate the predicate results up in the context node tree (lines 12-14).

Dynamic scope control During the above propagation, we also propagate scope status up in the context node tree for the purpose of implementing the node effectiveness and step/path scope as defined in Section 2. When a node effectiveness is terminated, other effective nodes that are located by Def.2.2 are removed from the context node tree and related states are also removed.

The scope status of a node is maintained in the propagate function (line 14 in Alg.1, line 9 in Alg.2). The isEffectivenessTerminate function checks node effectiveness. When a node effectiveness is terminated, related effective nodes and states are removed (lines 11-12 Alg.2).

4.4 Correctness (sketch)

It is obvious that queries in $XP^{\{\downarrow,\rightarrow,*\}}$ are precisely evaluated according to the NFA encoding rules in Fig.5. For predicate parts, the context node tree stores matched nodes of target and non-leaf predicate steps in the query tree. A branch node in the query tree is precisely handled, since both predicate results and buffered candidate nodes are propagated upward in the context node tree according to the node effectiveness and step/path scope. Therefore, Layered NFA precisely evaluates XPath queries in $XP^{\{\downarrow,\rightarrow,*,[]\}}$.

4.5 Example

We explain how Layered NFA evaluates the query in Fig.1 over the XML stream in Fig.2. First, the initial state ($0) is constructed and the XML stream is parsed. State $1 is also constructed and is transited to by the epsilon transition in the first layer NFA. For startElement(`dblp`) in line 1 Fig.2, state $2 is constructed. For startElement(`inproceedings`) in line 2, state $3 is constructed from the T-labeled state in the first layer NFA and the current node is buffered in the context node tree as *t1*. State $5 is also constructed and is transited to by the epsilon transition.

1st section element When the 1st startElement `section` in line 6 is emitted, state $6 is constructed from the NP-labeled state in the first layer NFA and the current node is put under *t1* as *np0* in the context node tree. Also the scopes of both leaf predicates (`title='Overview'` and `following::section`) start. $7 is constructed and is transited to because of the epsilon transition. When we receive endElement(`section`) in line 9, we transit from $7 to $8 by the E(*) transition and also go back to $5 which is popped from the state stack. Because the scope status of *np0* is child, the scope ends and the inner predicate evaluation is deemed to have failed. The corresponding context node *np0* is removed from the context node tree and the related states ($6, $7, $8) are also removed. Fig.6 shows the snapshot.

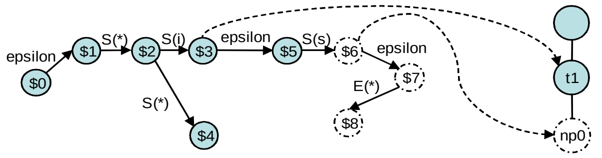

Figure 6: After processing endElement of the 1st `section`: dotted state and node indicate removed state and node, respectively

2nd section element The startElement of the 2nd `section` in line 10 is emitted, the same procedure is performed as done in the 1st `section` element; state $6 is constructed and the current node *np1* is put under *t1*. When characters(`Overview`) in line 11 is emitted, the inner predicate `[title='Overview']` becomes satisfied and the result is put in the context node. At this point, the context node of the predicate becomes effective, so the end point of the scope of the outer predicate `[section[title='Overview'] /following::section]` reaches the end of the stream. When the endElement of the 2nd `section` in line 13 is emitted, we transit from $7 to $8 by the E(*) transition and also go back to $5. The difference with respect to the 1st endElement(`section`) is that the end point of the scope of the outer predicate is at the end of the stream, so the context node *np1* and the related states ($6, $7, $8) are kept. Fig.7 shows the snapshot.

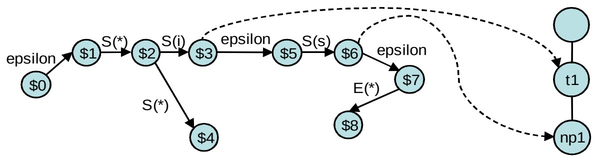

Figure 7: After processing endElement of the 2nd `section`

3rd section element When the 3rd startElement `section` (line 14) is emitted, not only is the same procedure performed as done in the 1st `section` element ($11 is constructed and the current node *np2* is put under *t1*), but also $9 is reached from $8. The second layer NFA in Fig.4 (c) shows the snapshot; the current states are $9, $10, $11, $12. The first layer NFA state of $9 is labeled with P, so the predicate result is put to *np1* and then propagated to *t1*. At this point, *t1*, a context node of the outer predicate, also becomes effective. Since all predicates are satisfied at *t1*, the buffered `inproceedings` node is flushed to the user.

4.6 Optimization

State sharing A problem of the original Layered NFA is that the number of states exponentially increases against query size, when a query contains the descendant/following axes. Our solution is to introduce the *state sharing* technique; it enables Layered NFA to eliminate redundant transitions by grouping the states that are constructed from the same state in the first layer NFA. Those states can be grouped together, because they have the same transitions to the following SAX events. Notice that, since Layered NFA is based on NFAs, once the descendant/following axes are encoded into NFAs, the same *state sharing* technique eliminates redundant transitions required for both axes.

The *state sharing* works as follows. During the construction of the next states ($nextStateSet$) at SAX events (line 17 in Alg.1, line 16 in Alg.2), we put them in a hashmap grouped by their source state in the first layer NFA. For each group of states, we choose one state as active and treat the remaining states as inactive. When the active state has updates on its candidate nodes, predicate results, or scope status, those updates are propagated to the inactive states.

State pruning for positive result of predicate For more efficient predicate processing, Layered NFA prunes states in the second layer NFA. When a predicate becomes satisfied, the related states can be removed because of the existential semantics of XPath predicates; once a predicate is satisfied, the predicate does not need further processing for the same context node. Again, since Layered NFA is based on NFAs, the same state pruning technique removes unneeded states regardless of their source axis types. In the running example, after receiving the 3rd startElement `section`, the unneeded states ($6, $7, $8, $9, $10) are removed in Fig.4, because the predicate is satisfied.

The removeRelatedStatesFrom function implements the state pruning (line 23 Alg.1, line 21 Alg.2). This function detects states that refer to a node in *hitNodes* or their descendant context nodes in the context node tree, and removes those states from *currentStateSet*. In addition, we don't transit to the states that are encoded from predicates that are already satisfied.

Global queue The descendant/following axes may produce duplication of the candidate nodes. Layered NFA uses a global queue [26] that contains a single copy of buffered candidate nodes to avoid the duplication. A candidate node stores a range label to allow its XML fragment to be extracted from the global queue. The preorder label is assigned to a candidate node at node construction time and the post-order label is assigned upon receiving the corresponding endElement event.

4.7 Complexity

Let $|D|$ be the number of nodes in XML stream D, $|Q|$ be the number of steps in query Q, d be the maximum depth of D, and B be the maximum number of candidate nodes buffered during the running time. According to [15], B might reach $|D|$ in the worst case, which cannot be avoided by any stream processing algorithm. The global queue enables us to store $|D|$ nodes at most as candidate nodes.

Table 1: XPath queries and sizes (numbers of states) in NFAs

Protein: $P = ProteinEntry, $R = reference, $Y = {1970,1980,1990,1995}		hit rate (%)	1st NFA	2nd NFA
Q1	/dummy	0	3	2
Q2	//*[.//*]	24.7	5	22
Q3	/ProteinDatabase//protein/name	1.2	5	9
Q4	/ProteinDatabase/$P/*/*/*/author	26.6	8	8
Q5	//$P/$R/refinfo/xrefs/xref/db	1.3	8	15
Q6	//$P//$R//refinfo//xrefs//xref//db	1.3	8	30
Q7	//organism[source]	1.2	5	9
Q8	//$P[$R]/sequence	1.2	7	12
Q9	//$P//refinfo[volume]//author	23.4	8	20
Q10	//$P/$R/refinfo[year=1988]/title	0.03	10	13
Q11	//$P[.//refinfo[title][citation]]/sequence	1.0	11	18
Q12	//$P/*[created_date ='10-Sep-1999']/uid	0.02	9	12
Q13	/ProteinDatabase/$P[$R/accinfo/mol-type='DNA'][$R/refinfo/year>1990]	1.0	14	9
Q14	/ProteinDatabase/$P[$R[accinfo[mol-type='DNA']]][$R[refinfo[year>1990]]]	1.0	18	9
Q15	//$P[.//mol-type='DNA'][.//year>1990]	1.0	9	21
Q16	//$P[$R[accinfo/mol-type='DNA']/following-sibling::$R/refinfo/year>$Y]	{0.1,0.1,0.1,0.1}	15	{14,14,14,14}
Q17	//$P[$R[accinfo/mol-type='DNA']/following::$R/refinfo/year>$Y]	{1.0,1.0,1.0,0.8}	15	{20,20,20,20}
TreeBank:		hit rate (%)	1st NFA	2nd NFA
Q1	/dummy	0	3	2
Q2	//*[.//*]	40.7	5	84
Q3	//EMPTY[.//S/NP/NNP='U.S.']	0.021	9	81
Q4	//EMPTY[.//S/NP[NNP='U.S.']/following-sibling::MD[text()='will']]	0.001	15	81
Q5	//EMPTY[.//S[NP/NNP='U.S.'][VP/NP/NNP='Japan']]	0.000	15	100
Q6	//EMPTY[.//PP[IN[text()='in']/following-sibling::NP/NNP='U.S.']]	0.011	15	75
Q7	//EMPTY[.//S/NP/NP[NNP='U.S.']/following-sibling::JJ='economic']	0	15	81

Space complexity The space complexity of Layered NFA is determined by the sum of the sizes (i.e. number of states) of two NFAs and the size of the context node tree.

First, we discuss the sizes of the NFAs. The size of the first layer NFA for Q is always $O(|Q|)$ according to the NFA encoding rules in Fig.5. For the second layer NFA, the state sharing technique eliminates redundant transitions to the states that are constructed from the same state in the first layer NFA, therefore there can be $|Q|$ states constructed at most for every startElement event, and those states are kept in the state stack. When Q is in $XP^{\{\downarrow,*,[]\}}$, the space complexity of the second layer NFA is $O(d|Q|)$, because there can be d startElement events that are continuously matched with each downward axis in Q. When Q is in $XP^{\{\downarrow,\rightarrow,*,[]\}}$, the space complexity of the second layer NFA is $O(|D||Q|)$, because there can be $|D|$ startElement events that are continuously matched with each forward axis in Q.

Second, we discuss the size of the context node tree, which consists of the context nodes of non-leaf predicates and the buffered candidate nodes (B). When Q doesn't contain predicates, $XP^{\{\downarrow,\rightarrow,*\}}$, there is only a root node in the context node tree, since there is no need for constructing context nodes of non-leaf predicates or buffering candidate nodes. If we permit to include predicates, the maximum size of the context nodes of non-leaf predicates is $d|Q|$ when Q is in $XP^{\{\downarrow,*,[]\}}$, and $|D||Q|$ when Q is in $XP^{\{\downarrow,\rightarrow,*,[]\}}$, which is obtained by the same discussion as the second layer NFA. Therefore, we obtain the following theorem.

Theorem 4.2 *The space complexity of Layered NFA is $O(|Q|)$ for the first layer NFA, $O(d|Q|)$ for the second layer NFA of $XP^{\{\downarrow,*,[]\}}$, $O(|D||Q|)$ for the second layer NFA of $XP^{\{\downarrow,\rightarrow,*,[]\}}$, $O(d|Q| + B)$ for the context node tree of $XP^{\{\downarrow,*,[]\}}$, and $O(|D||Q| + B)$ for the context node tree of $XP^{\{\downarrow,\rightarrow,*,[]\}}$.*

Time complexity The time complexity of Layered NFA consists of the cost of second layer NFA operation and the cost of propagation that occurs within the *state sharing* technique. The former is $O(|D||Q|)$, the product of the XML stream size $|D|$ and the maximum number of the current states $|Q|$ in the second layer NFA, since each current state handles every SAX event in D in the worst case. The worst case of *state sharing* propagation is that there can be $|Q|$ inactive states that receive the propagation for every SAX event in D, resulting in $O(|D||Q|)$ time complexity. Since both time complexities are $O(|D||Q|)$, the total time complexity of Layered NFA is also $O(|D||Q|)$.

5. EXPERIMENTS

We validated the efficiency (response time and space) of Layered NFA in a comparison to related algorithms, SPEX [24][6], XSQ [26], and xmltk [16].

Our execution environment consisted of a dual Intel(R) Core (TM) PC (CPU 2.40GHz) with 1.99GB of main memory, running Windows XP. We used Java 1.5.0_06 with default settings and a lightweight XML parser Crimson for all algorithms except xmltk, which was implemented in C++, we used g++ compiler version 3.44 (cygwin) with the -O2 optimization option. xmltk uses its own non-validating XML parser.

In these experiments, we customized the implementations to prevent output of the matched XML fragments to avoid the overhead of system I/O. Some parts of the results were not gathered due to the limitations of the algorithms or implementation issues as fol-

[6]We used PHASE2 optimization option.

Table 2: XML streams statistics

	File size (MB)	Depth		Num. of elements	
		avg	max	schema	data(K)
Protein	706	5.15	7	66	21306
TreeBank	60	7.87	36	250	2438

lows. The SPEX implementation failed to evaluate the following axis even though its algorithm should be capable to doing so. The XSQ algorithm evaluates queries in $XP^{\{\downarrow,[]\}}$ with unnested predicates whose paths have at most one step. The xmltk algorithm evaluates queries limited to $XP^{\{\downarrow,*\}}$.

XML streams: We used two real XML streams available at XML Data repository[7]: the PIR-International Protein Sequence Database (Protein) and the XML-ized TreeBank linguistic database (TreeBank). They have been widely used in experiments on XML stream algorithms [15, 8, 26, 16]. Protein is an integrated collection of functionally annotated protein sequences; it is one of the largest XML streams. TreeBank consists of English sentences tagged with parts of speech and has deep recursive structures. Their statistics are summarized in Table 2.

XPath queries: We used various types of XPath queries to characterize the performance features of the XPath evaluation algorithms. For both XML streams, we used Q1 `/dummy` (no element match) to validate the base performance of the algorithms, and used Q2 `//*[.//*]` to validate in an extreme case with predicates. For Protein, in addition to the queries used in [8], we added queries with multiple predicates and/or the following/following-sibling axes. For TreeBank, we used interesting queries for linguistic analysis with predicates and/or the following-sibling axis: Q3 selects descriptions whose subject is a specified country `U.S.`. Q4-Q7 are obtained by adding various predicates to Q3 as follows. Q4 selects descriptions on future actions of the country. Q5 selects descriptions whose subject is two countries, U.S. and Japan. Q6 selects descriptions of something that occurred in the country. Q7 selects descriptions about the economy of the country. Table 1 shows the queries, their hit rate, and the sizes of Layered NFA in the first and second layers.

5.1 Query processing time

Fig.8 and Fig.9 show the performance comparisons of XPath evaluation algorithms over Protein and TreeBank XML streams, respectively. According to the results of Q1 `/dummy` which show that all algorithms have comparable performance, we conjecture that we can basically ignore the differences in programming languages (Java, C++) and XML parsers (Crimson, xmltk parser). Recall, only xmltk uses a different programming language and XML parser.

We found that Layered NFA is up to four times faster, and twice as fast on average, than SPEX. Our observations on the difference between Layered NFA and SPEX are as follows. First of all, as pointed out in Section 1, the performance of SPEX indeed worsens as the number of predicates increases. For the Protein dataset, except for the queries with large hit rate (Q2, Q4, Q9), Q1, Q3, Q5, Q6 (no predicates), Q7 (single predicate, no branches in query trees) are faster than Q8, Q10, Q12 (single predicate, single branch in query trees), and the latter are faster than Q11, Q13-16 (multiple predicates). For the TreeBank dataset, except for the queries with large hit rate (Q2), Q1 (no predicates) is faster than Q3 (single predicate, no branches in query trees), and the latter is faster than Q4-Q7 (multiple predicates). This is due to SPEX's design: it independently evaluates each predicate and the trunk part of the query, then merges the intermediate results, yielding predicate processing overhead. In contrast, the performance of Layered NFA is stable for all queries regardless of the number of predicates, because it processes the whole query at SAX events while generating smaller intermediate results than SPEX.

Second point is forward axes processing. The performance of Layered NFA for the queries with the following axis (Protein Q17) is slightly slower than that with the following-sibling axis (Q16). This result proves the effectiveness of the *state sharing* technique for following axis processing; it dramatically reduces NFA size. We will describe the effect on the space reduction in the next section.

Third point is descendant axis processing. SPEX processes the descendant axis as efficiently as the child axis, as we can observe that the SPEX performance for Protein Q5 is the same as that for Protein Q6, which is obtained by replacing every child axis in Q5 with the descendant axis. In addition, although Protein Q13 and Q15 have different expressions, their query results are the same. Since Q15 has shorter length than Q13, SPEX processes Q15 more efficiently than Q13. In contrast, Layered NFA processes the descendant axis slower than the child axis, because the descendant axis requires two transitions whereas the child axis requires one transition. Actually, the performances of Layered NFA for Q6 and Q15 with the descendant axis are slower than Q5 and Q13 without the descendant axis, respectively.

For the case of XSQ in $XP^{\{\downarrow,[]\}}$ (Q1, Q3, Q5-10), Layered NFA is comparable to XSQ. This is mainly because they are both NFA-based algorithms. xmltk is always the fastest in $XP^{\{\downarrow,*\}}$ (Q1, Q3-Q6) especially for queries that contain many descendant axes (Q6). This is explained by the fact that xmltk is a DFA-based algorithm, thus it only needs to keep track of a single current state. Layered NFA and XSQ are both NFA-based algorithms, so they need to keep track of multiple current states resulting in lower performance.

5.2 Space consumption

The last two columns in Table 1 show the sizes of Layered NFA in the first and second layers. Since the second layer NFA is dynamically constructed and removed, we report the maximum summations of current and stacked states at any moment during stream processing. All the results support Theorem 4.2: the size of the first layer NFA is linear to query size $|Q|$ and the upper bound of the size of the second layer NFA is $O(d|Q|)$ for $XP^{\{\downarrow,*,[]\}}$ and $O(|D||Q|)$ for $XP^{\{\downarrow,\rightarrow,*,[]\}}$. We describe details of the second layer NFA below. For Q1 `/dummy`, there is only the initial state since there is no successful transition. For Protein Q4 in $XP^{\{\downarrow,*,\}}$, the state size is the same as or smaller than the size of the first layer NFA. For other queries in $XP^{\{\downarrow,\rightarrow,*,[]\}}$, there are more states in TreeBank than in Protein, mainly because the depth of TreeBank XML stream is 36 while that of Protein XML is 7. The value parameters ({1970,1980,1990,1995}) in the queries do not impact NFA size. The size of the second layer NFA is significantly improved by the state sharing technique, for example, from {25,100,700,3937} to {20,20,20,20} against Protein Q17 for value parameters ({1970,1980,1990,1995}), respectively.

[7]http://www.cs.washington.edu/research/xmldatasets/

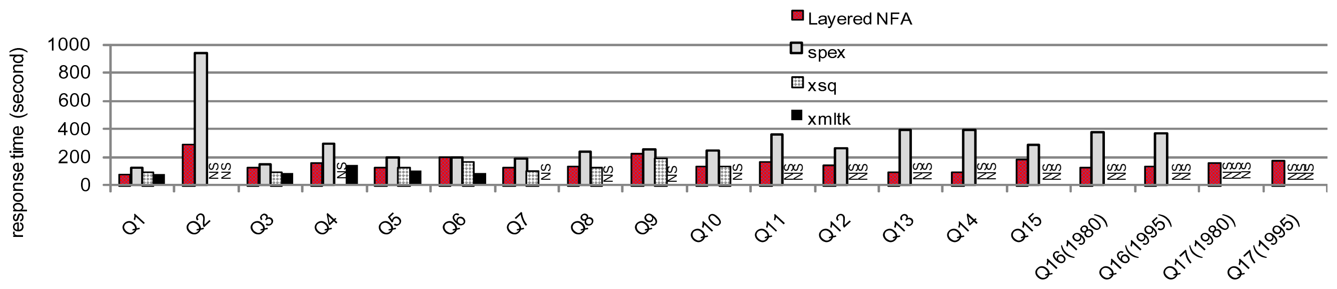

Figure 8: Response times (Protein XML): NS indicates that the implementation doesn't support the query.

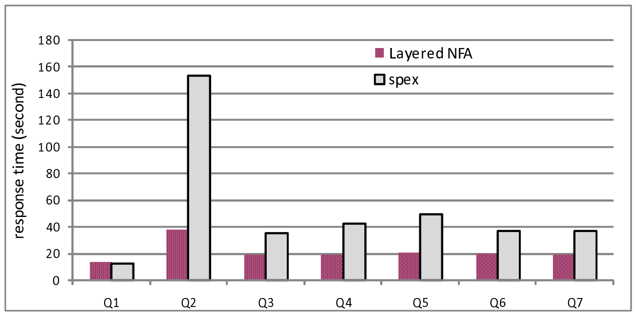

Figure 9: Response times (TreeBank XML)

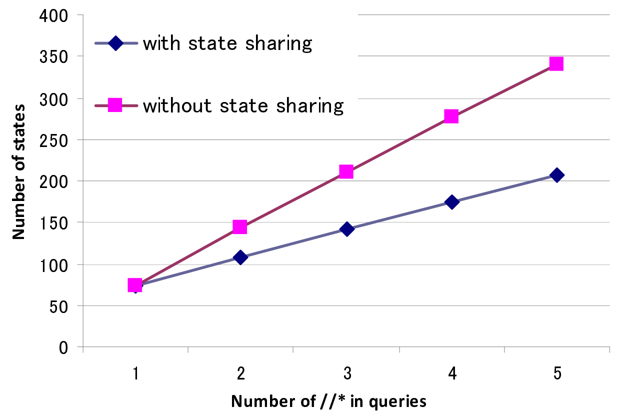

Figure 10: Effect of state sharing (TreeBank XML)

Fig.10 also shows the effect of the state sharing technique. We used the treebank XML stream and queries consisting only of //* with lengths from 1 to 5. The results show that the size of second layer NFA is linear to the query size in both cases with/without state sharing. Consider the size of second layer NFA without state sharing. Theoretically, it is $O(d|Q|)$ for $XP^{\{\downarrow,*\}}$, $O(d^{|Q|})$ for $XP^{\{\downarrow,*,[]\}}$, and $O(|D|^{|Q|})$ for $XP^{\{\downarrow,\rightarrow,*,[]\}}$. $O(d^{|Q|})$ for $XP^{\{\downarrow,*,[]\}}$ is obtained, because a step with the descendant axis matches d nodes at most, those nodes are used as context nodes for its predicates, and this is repeated $|Q|$ times at most. For $XP^{\{\downarrow,\rightarrow,*,[]\}}$, a step with the following axis can match $|D|$ nodes, so $O(|D|^{|Q|})$ becomes the upper bound size. In contrast, as Theorem 4.2 shows, the state sharing technique avoids the exponential growth in the state size of Layered NFA.

Overall, Layered NFA scales well on very large or complicated XML streams for XPath queries in $XP^{\{\downarrow,\rightarrow,*,[]\}}$.

6. RELATED WORK

Query rewrite [14, 23, 27] define XPath semantics based on query rewrite schemes which evaluate one step at a time in a query and change context nodes. This approach is suitable for XML databases that randomly access any node in an XML data. However, it is not suitable for XML stream processing, in which the access to the XML stream is limited to just sequential order. Query rewrite techniques [25, 13] rewrite queries with reverse axes (parent, ancestor, preceding, preceding-sibling) into equivalent queries without reverse axes. They allow our techniques to be applied to a larger class of queries.

XML Stream algorithms We can classify the algorithms of XPath query processing over XML streams into XML filtering and full-fledged XPath evaluation.

The problem of XML filtering is to detect matched queries over XML streams given a large number of queries. Various approaches [1, 7, 9, 10, 16, 17, 6] have been proposed for this problem. They focus on how (NFA-based, DFA-based, Bloom filter based) and what (step, simple path, branch) to share among a large number of queries in $XP^{\{\downarrow,*,[]\}}$. None of them supports $XP^{\{\downarrow,\rightarrow,*,[]\}}$. The algorithms in [24, 3, 15, 8, 26, 16] support full-fledged XPath evaluations, while the algorithms in [22, 19, 21, 20, 12] are for XQuery evaluations. The buffering space lower bound was studied by [2] for $XP^{\{\downarrow,*,[]\}}$.

SPEX [24, 5] has already been described in Section 1. Algorithms [15, 8, 12] employ a stack-based algorithm [4] for efficient descendant axis processing. One study [15] reported O($|D||Q|$) time complexity for XML stream processing using a multi-stack framework for $XP^{\{\downarrow,*,[]\}}$ evaluation. The eager querying algorithm, which doesn't delay the actions when a predicate is evaluated as true, also achieves optimal buffering-space complexity. TwigM [8] efficiently evaluates $XP^{\{\downarrow,*,[]\}}$ queries by employing twigStack [4] for candidate node representation. For the queries with the descendant axis and predicates, twigStack encodes n^2 matched patterns using a $2n$-size data structure. The BEA streaming XQuery engine [12] gives an overview of the XQuery processor implementation but doesn't discuss the details of streaming processing. It combines streaming processing with materialization of common subexpressions in XQuery expressions. It also exploits schema information to rewrite queries [11] and a stack-based algorithm [4] for efficient descendant axis processing.

XSQ [26] is based on hierarchical NFA augmented with buffers for $XP^{\{\downarrow,[]\}}$ evaluations. XSQ compiles a single query into a hierarchical NFA consisting of $2^{|Q|-1}$ NFAs where the states are generated from all possible combinations of predicate results. At run time, a query is subjected to $d^{|Q|}$ different context nodes. Thus the space complexity is O($2^{|Q|} \times d^{|Q|}$) and time complexity is O($|D| \times 2^{|Q|} \times d^{|Q|}$). A state in XSQ remembers which predicates have been satisfied and has transitions to the states that contain unsatisfied predicates that remain to be evaluated. Instead of constructing $2^{|Q|-1}$ NFAs, Layered NFA dynamically maintains the second layer NFA, which corresponds to the hierarchical NFA of XSQ: 1) a context node of the NFA state remembers which predicates have been satisfied, and 2) the state pruning for positive predicate result suppresses transitions to the following states encoded from predicates that are already satisfied. There are two limitations to XSQ. First, XSQ uses static scope control since it doesn't process the following/following-sibling axes. Second, the XSQ algorithm limits queries in $XP^{\{\downarrow,[]\}}$ to unnested predicates whose paths have at most one step.

xmltk [16] is a collection of fundamental tools for XML stream operations (selection, sort, nest, unnest); it is based on Lazy DFA, an algorithm for evaluating queries without predicates ($XP^{\{\downarrow,*\}}$). Its time complexity is O($|D|$) because of the deterministic transitions: there is always a single current state for every SAX event.

7. CONCLUSION

We proposed Layered NFA, an efficient XML stream processing algorithm for queries in $XP^{\{\downarrow,\rightarrow,*,[]\}}$. Layered NFA evaluates queries within one XML parsing pass and outputs the matched fragments in the XML stream. It uses double-layered NFA: first layer NFA is compiled from XPath queries and is able to evaluate subqueries in $XP^{\{\downarrow,\rightarrow,*\}}$ while second layer NFA handles predicate parts. Layered NFA achieves $O(|D||Q|)$ time complexity by introducing the *state sharing* technique, which avoids the exponential growth in the state size of Layered NFA by eliminating redundant transitions. Experiments showed that Layered NFA is up to four times faster, and twice as fast on average, than SPEX, and is comparable to XSQ for $XP^{\{\downarrow,[]\}}$ queries.

Acknowledgment

The author would like to express his gratitude to Dario Colazzo for his invaluable help in improving the paper presentation, and anonymous reviewers for their constructive comments on this paper.

8. REFERENCES

[1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of VLDB*, 2000.

[2] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Buffering in query evaluation over XML streams. In *Proceedings of PODS*, New York, NY, USA, 2005. ACM.

[3] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath processing with forward and backward axes. In *Proceedings of ICDE*, 2003.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of SIGMOD*, 2002.

[5] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. The XML stream query processor SPEX. In *Proceedings of ICDE*, 2005.

[6] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. AFilter: adaptable XML filtering with prefix-caching suffix-clustering. In *Proceedings of VLDB*, 2006.

[7] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of ICDE*, 2002.

[8] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient XPath query processor for XML streams. In *Proceedings of ICDE*, 2006.

[9] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database System*, 28(4), December 2003.

[10] Y. Diao and M. J. Franklin. Query processing for high-volume XML message brokering. In *Proceedings of VLDB*, 2003.

[11] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of ICDE*, 1998.

[12] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, J. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *The VLDB Journal*, 13(3), 2004.

[13] P. Genevès and K. Rose. Compiling XPath for streaming access policy. In *Proceeedings of DocEng*, 2005.

[14] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database System*, 30(2), 2005.

[15] G. Gou and R. Chirkova. Efficient algorithms for evaluating XPath over streams. In *Proceedings of SIGMOD*, 2007.

[16] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems*, 29(4), December 2004.

[17] A. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proceedings of SIGMOD*, 2003.

[18] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation (Second Edition)*. Addison-Wesley, 2001.

[19] Z. Ives, A. Halevy, and D. Weld. Efficient evaluation of regular path expressions on streaming XML data. Technical Report UW-CSE-2000-05-02, 2000.

[20] C. Koch, S. Scherzinger, and M. Schmidt. The GCX system: Dynamic buffer minimization in streaming XQuery evaluation. In *Proceedings of VLDB*, 2007.

[21] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An optimizing XQuery processor for streaming XML data. In *Proceedings of VLDB*, 2004.

[22] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *Proceedings of VLDB*, 2002.

[23] M. Marx and M. de Rijke. Semantic characterizations of navigational XPath. *SIGMOD Record*, 34(2), 2005.

[24] D. Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Transactions on Knowledge and Data Engineering*, 19(7), 2007.

[25] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proceedings of Workshop on XML Data Management (XMLDM)*, LNCS. Springer, 2002.

[26] F. Peng and S. S. Chawathe. XSQ: A streaming XPath engine. *ACM Transactions on Database Systems*, 30(2), 2005.

[27] P. Wadler. Two semantics for XPath, 1999.