An Efficient XPath Query Processor for XML Streams

Yi Chen
Arizona State University
yi@asu.edu

Susan B. Davidson
University of Pennsylvania
susan@cis.upenn.edu

Yifeng Zheng
University of Pennsylvania
yifeng@cis.upenn.edu

Abstract

Streaming XPath evaluation algorithms must record a potentially exponential number of pattern matches when both predicates and descendant axes are present in queries, and the XML data is recursive. In this paper, we use a compact data structure to encode these pattern matches rather than storing them explicitly. We then propose a polynomial time streaming algorithm to evaluate XPath queries by probing the data structure in a lazy fashion. Extensive experiments show that our approach not only has a good theoretical complexity bound but is also efficient in practice.

1 Introduction

XML has become the de facto standard for data exchange. The problem of efficiently evaluating XML queries, e.g. XPath, in both *main memory* and *streaming* environments has therefore attracted a lot of attention from the research community [7, 16, 28].

In this paper, we focus on a streaming environment, as found with stock market data, network monitoring or sensor networks. In such an environment, data streams, which can be potentially infinite, arrive continuously and must be processed using a single sequential scan because of the limited storage space available. Query results should be distributed incrementally and as soon as they are found, potentially before we read all the data. Furthermore, the query processing algorithm should scale well in time and space. An algorithm that meets these requirements for XPath processing over XML data is called a *streaming XPath evaluation algorithm*.

Several streaming XPath evaluation algorithms based on finite state automata (FSA) have been proposed to process XPath queries containing the child axis ('/'), descendant axis ('//') and wildcard ('*') [3, 19]. Automaton-based methods are attractive due to their efficiency and clean design. However, they cannot evaluate XPath queries which contain predicates ('[...]') since an FSA is memory-less, as

observed in [25]. Since predicates are common in XPath queries, we must be able to handle not only wildcards, child and descendant axes, but also predicates.

When evaluating predicates on XML streams, we may encounter data that potentially can be a query solution – a *candidate* node – before we encounter the data required to evaluate the predicates to decide its membership; therefore, we must remember candidates as well as their query pattern matches until the relevant data is encountered. For example, consider the XPath query $//a[d]/b[e]//c$ and the sample XML document shown in figure 1(a)[1]. When we process the XML element c_1 in the document order (or equivalently, a pre-order traversal of the XML tree), we cannot determine whether or not it is in the query result at the point that it is encountered. We therefore need to record information about the pattern match to subquery $//a/b//c$: (a_n, b_1, c_1) until we can determine the predicate satisfaction of a_n and b_1, thus deciding whether or not c_1 is a solution.

Based on this intuition, several algorithms [23, 25, 26, 21, 20] have been proposed to process XML queries containing predicates. These algorithms are efficient and scale well for *nonrecursive* XML streams, i.e. data in which tags do not repeat along a root-to-leaf path. However, when predicates are combined with descendant axis traversal and the XML data is recursive, evaluating XPath queries in a streaming fashion raises new challenges:

- Due to the combination of descendant axis traversal in a query and the recursive structure of XML data, the number of pattern matches of a single XML node to a subquery can be potentially exponential to the query size. Consider the query Q_1 : $//a[d]//b[e]//c$ and the XML data in figure 1(a). Note that Q_1 is different from the earlier query due to the descendant (rather than child) axis traversal between tags a and b. For the XML node c_1 there are n^2 ways for c_1 to match subquery $//a//b//c$: (a_i, b_j, c_1), where $1 \le i \le n, 1 \le j \le n$. Representing n^2 in terms of the size of the data and query, this becomes $O((|D|//|Q|)^{|Q|})$, where $|D|$ is the XML data size, $|Q|$ is the XPath query size.
- At least one pattern match must satisfy the query predi-

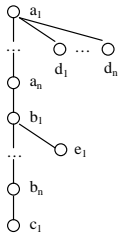[1]Throughout the paper, we use subscripts to distinguish nodes with the same tag.
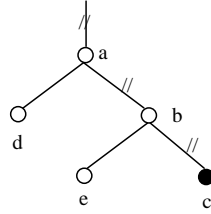
(a) Data D_1 (b) Query Q_1

Figure 1. Sample XML Data and an XPath Query

cates to make a candidate become part of the result. However, until a pattern match which satisfies the query predicates is found, we must record all the pattern matches for a candidate and test their predicate satisfaction. In the worst case, we will not know whether or not the candidate is indeed a solution until all pattern matches are considered.

For example, when node c_1 is met, since we are not able to determine the predicate satisfaction of its n^2 subquery pattern matches we must record all of them. Processing the data in document order, we then verify that the matches (a_i, b_j, c_1), $2 \leq i \leq n$, $2 \leq j \leq n$ fail the predicates in Q_1. At the very end, we find that the match (a_1, b_1, c_1) satisfies the predicates and therefore c_1 is a query solution.

These challenges do not exist in a non-streaming environment since XML nodes can be randomly accessed during query processing. For example, in the best known polynomial time main memory algorithms for evaluating XPath queries [16], the whole document is loaded into main memory before query processing. Since XML nodes can be randomly accessed, predicates can be checked first so that we do not need to remember the pattern matches. These techniques are not suitable for processing XML streams, where only a single sequential scan is allowed. Furthermore, as will be shown in section 5, the algorithms have trouble processing large XML files.

Previous XPath streaming algorithms that handle predicates either do not support descendant axis traversal [23, 21], or explicitly store all pattern matches [25, 26]. As analyzed in [26], the worst case complexity of the algorithms in [25, 26] is $O(|D| \times 2^{|Q|} \times k)$, where k is the number of different query pattern matches in which an XML node participates.

As discussed, recording pattern matches by enumerating and storing them explicitly can be expensive. Motivated by [7], we therefore design a stack-based data structure to concisely encode pattern matches. We then propose a novel XPath streaming algorithm, TwigM, which searches for satisfying matches in the compact data structure by pruning the

search space without enumerating all the pattern matches. In this way, TwigM achieves a complexity which is polynomial in the size of the data and query. As tested in extensive experiments, TwigM is a practically efficient and worst case polynomial algorithm.

The TwigM algorithm was implemented in an XPath query processor for XML streams and demonstrated in [11].

The contributions of this paper are:

1. We design a data structure to encode the matches in a compact form. For example, to process Q_1 on the sample data in figure 1(a), TwigM stores $2n$ nodes to encode n^2 pattern matches.

2. We propose a streaming XPath evaluation algorithm called TwigM. Recall that to determine whether a candidate is indeed a solution entails finding at least one pattern match that satisfies all the query predicates. Rather than computing all the pattern matches in the search space explicitly from the compact data structure, and then testing predicate satisfaction, TwigM prunes the search space as we process the XML stream by checking predicate satisfaction on a small number of elements in the data structure. For example, we only need to check predicate satisfaction on $2n$ elements instead of checking n^2 pattern matches to evaluate Q_1.

3. We analyze the time complexity of TwigM, which is polynomial in terms of the size of the data and query.

4. We present a detailed performance evaluation of an implementation of TwigM compared with several other related systems. The results show that our approach not only has a good theoretical complexity but a good performance on various practical queries and data sets.

The remainder of the paper is organized as follows. Section 2 presents the data model and query language. Section 3 gives an overview our XPath streaming evaluation strategy. TwigM is introduced and analyzed in section 4. Section 5 presents performance results. Section 6 discusses related work, and section 7 concludes the paper.

2 Data Model and Query Language

In this paper, XML data is modeled as a stream of modified SAX events: startElement(tag, $level$, id) and endElement(tag, $level$), where tag is the tag of the node being processed, $level$ is level of the node in the corresponding XML tree, and id is unique identifier of the node. [2] These events are the input to our algorithms.

Definition 2.1: The *current node* is the XML node whose tag is currently being parsed by the SAX parser. An *active*

---

[2]We omit the discussion of attributes for now, however our implementation supports attributes as well as elements.

node is an XML node whose start tag has been processed but end tag has not yet been processed by the SAX parser. ∎

Proposition 2.1: At any point in time, the number of active nodes is bounded by the depth of the XML tree. ∎

We focus the discussion on a commonly used subset of XPath 1.0: $XP^{\{/,//,*,[]\}}$, following [25, 24, 17, 32]. $XP^{\{/,//,*,[]\}}$ consists of child axis traversal (/), descendant axis traversal (//), wildcards (*), branches (or predicates, denoted as [...]), and name tests.

Following previous work [1, 7], we represent an XPath query in $XP^{\{/,//,*,[]\}}$ as a *query tree*. For example, the query Q_1:$//a[d]//b[e]//c$ searches for all c nodes that are descendants of b nodes, which in turn have a child e and an ancestor a with at least one child d. The tree corresponding to Q_1 is shown in figure 1(b). The c node in Q_1 is called the *return* node and is indicated by darkening the node. An unannotated line between two nodes represents a child axis, and a line annotated with // represents a descendant axis. If a node has more than one child or is the return node, then it is called a *branching node*. For example, nodes a, b and c are branching nodes in Q_1. We use "XPath query" and "query tree" interchangeably.

As discussed in section 1, there are two challenges in efficiently processing $XP^{\{/,//,*,[]\}}$ on XML streams: first, descendant axes in the query combined with the recursive structure of XML data; and second, predicates in the query combined with the single-scan requirement of stream processing. Therefore we will start by considering two simple subsets of $XP^{\{/,//,*,[]\}}$: $XP^{\{/,//,*\}}$, which denotes XPath queries without branching; and $XP^{\{/,[]\}}$, which denotes XPath queries without descendent axis traversal and wildcards. The techniques used in processing queries in these sub-languages will then be combined in a query processor for $XP^{\{/,//,*,[]\}}$.

3 Overview of Query Processing

In this section, we give the intuition of how $XP^{\{/,//,*,[]\}}$ queries are evaluated over XML data streams; details of the algorithms will be given in the next section.

For an XPath query Q, we build a machine M which takes as input a sequence of SAX events of an XML stream D and computes a set of node ids as solutions of Q [3]. The structure of M resembles that of Q, with data structures attached to machine nodes to record information about matches(see figure 2(b) and (c)). We start by describing machines for the simple cases, PathM for queries in $XP^{\{/,//,*\}}$ and BranchM for queries in $XP^{\{/,[]\}}$, before extending them to one for $XP^{\{/,//,*,[]\}}$, TwigM.

---

[3]Our implementation returns XML fragments instead of node ids.

(a) Data D_2 (b) Query Q_2 (c) PathM M_2 for Q_2
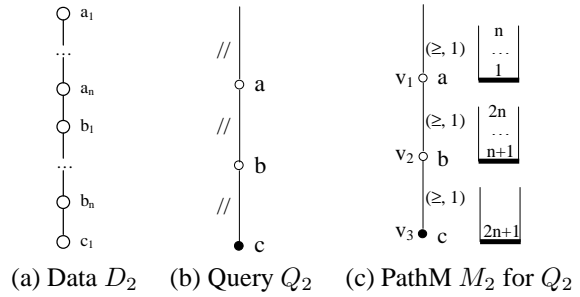
Figure 2. Example for PathM

3.1 PathM: $XP^{\{/,//,*\}}$ Query Processing

As discussed earlier, the combination of descendant axis traversal in queries and the recursive structure of XML data can result in an exponential number of pattern matches to the query size for a single query solution. To evaluate queries using polynomial time, we encode matches compactly and avoid enumerating all the matches for a solution.

To achieve the first requirement, we attach a stack to each machine node v to record *active XML nodes* which are solutions to the subquery from the machine root to v. The XML nodes on a stack are retained only as long as they are active, and therefore by Proposition 2.1 the memory requirement for a stack is bounded by the document depth. Since a tag may occur multiple times in a query, the total memory requirement is bounded by the size of the query times the document depth.

To achieve the second requirement, we push an XML node onto the stack of machine node v if its relationship with the nodes in the stack of v's parent node u satisfies the axis between u and v. Since u's stack stores all active solutions for the subquery from the root to u, the XML node pushed onto v's stack is a solution for the subquery from the root to v. The time to check the push condition for each XML node is therefore bounded by the number of active nodes (i.e the document depth, Proposition 2.1) times the query size.

Example 3.1: Figure 2(c) shows the machine M_2 for the query Q_2 in figure 2(b). A machine node is created for each query node and given the label of its corresponding query node's tag. For example, machine node v_1 is labeled a, v_2 is labeled b, and v_3 is labeled c. The machine node built for the root (return) node in the query is also called the root (return) node in the machine. Thus v_1 is the root of M_2 and v_3 is the return node.

A stack is built for each machine node to record information about active XML nodes that are solutions to its subquery. For example, the stack for v_1 records active XML nodes reachable by $//a$, and the stack for v_2 records active XML nodes reachable by $//a//b$. Since active XML nodes

can be distinguished by their levels, the stacks record only the level of matching active nodes.

The edge between machine nodes is annotated with a node push condition according to the axis between the corresponding query nodes. For example, the parent edge of v_2 is labeled by "$(\geq, 1)$", since the corresponding query node b has a parent edge of '//'. The edge label indicates that an XML node will be pushed to v_2's stack if and only if there exists a node in v_1's stack such that their level difference is ≥ 1.

PathM accepts the SAX events of an XML stream and computes the solutions for the query. Each SAX event (tag, $level$, id) will be sent to machine nodes whose label is the same as tag or '*'. In M_2's execution on the XML tree D_2 of figure 2(a), the SAX event startElement($a, 1, a_1$) will be sent to v_1 since its label is a.

A machine node v qualifies for a startElement SAX event (tag, $level$, id), if (1) v is the root and $level$ satisfies its parent edge label; or (2) v is not the root and there exists an l' on the stack of v's parent such that $level - l'$ satisfies v's parent edge label. This XML node is pushed onto the qualified node v's stack as a solution to the subquery from the root to v. Continuing with our example, a_1 is pushed on v_1's stack since v_1 is the root, v_1's parent edge is labeled $(\geq,1)$, and a_1's $level = 1 \geq 1$. Similarly, we push data nodes $a_2, \ldots, a_n$ on v_1's stack, data nodes $b_1, \ldots, b_n$ on v_2's stack, and data nodes c_1 on v_3's stack. The snapshot of M_2's state at this point of execution is shown in figure 2(c). Since v_3 is the return node, node id c_1 is output.

A machine node v qualifies for an endElement event(tag,$level$) if $level$ is equal to the top node in v's stack (meaning that this is the matching end tag); v then pops its stack to guarantee that only active nodes remain. For example, endElement($c, 2n + 1$) is sent to machine node v_3 since its label is c. Furthermore, the top element in v_3's stack is level $2n + 1$ and is equal to the $level$ of the endElement event; we therefore pop v_3's stack. ∎

As an optimization, we do not need to create machine nodes for '*' interior query nodes. Instead, we record the level difference of non-'*' query nodes through the edge labels of their corresponding machine nodes.

Observe that in this example, although there are n^2 pattern matches which qualify c_1 as a query solution ((a_i, b_j, c_1) where $1 \leq i \leq n$, $1 \leq j \leq n$), we only store $2n$ nodes. Furthermore, to verify each sub-query solution it is sufficient to check nodes on the parent stack of a qualified machine node. For example, to determine that c_1 is a query solution we only need to check nodes in v_2's stack, rather than enumerating and testing all n^2 query pattern matches that c_1 participates in.

Although the stacks used in PathM are similar to those proposed in [7], the algorithms to compute query results are quite different. First, PathM pushes nodes onto a stack
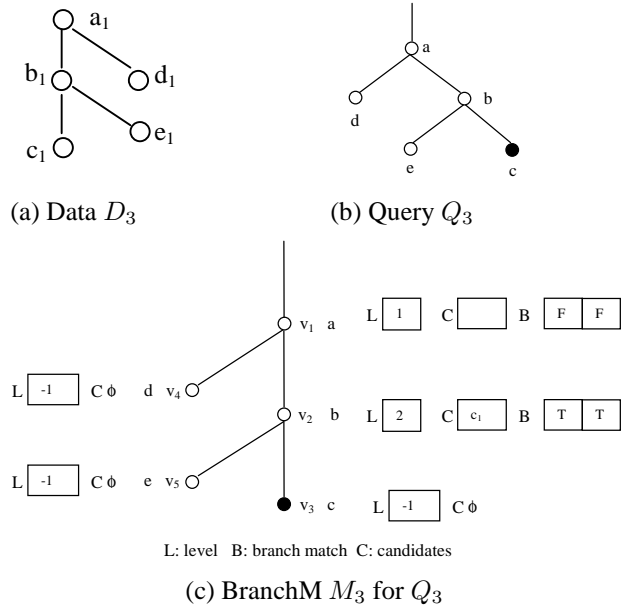

(a) Data D_3 (b) Query Q_3

L: level   B: branch match   C: candidates

(c) BranchM M_3 for Q_3

Figure 3. Example for BranchM

if and only if they are solutions to the corresponding subquery, rather than pushing all nodes whose tag matches the label of the stack. Second, to determine if an XML node is a query solution, we only need to check its relationship with nodes in the parent stack of the query return node (polynomial time), rather than enumerating and verifying all the pattern matches the XML node participates in.

3.2 BranchM: $XP^{\{/,[]\}}$ Query Processing

Another challenge for evaluating XPath queries in a streaming environment lies in checking predicate satisfaction. We focus on this problem in BranchM, which processes queries in $XP^{\{/,[]\}}$. The issue is that when a data node matches a machine node, we may not be able to determine if it is a solution to the query since the nodes matching the predicate conditions may not have been seen yet.

We therefore associate with each machine node a set of possible solutions called *candidates*. To verify a candidate, we need to record pattern matches to subqueries and test predicate satisfaction on them. Since a predicate is an existential quantifier which can be satisfied with a single match to each condition, we attach a boolean array called *branch match* to each machine node and record whether or not each of the child conjunctions has found a match in the XML data rather than recording all the matches.

Example 3.2: Figure 3(c) shows the machine M_3 for query Q_3 in figure 3(b). The BranchM machine is different from a PathM machine in three key ways: First, an edge label in BranchM is always $(=,1)$ since there are only child axes

4

and no '*'-nodes; we therefore omit edge labels from the figure. Second, at any moment there is at most one active XML node matching a query node; machine nodes therefore record a single match rather than a stack of matches. Third, for each BranchM machine node, we associate the level L of the match (initialized to 0), a set of candidates C (initialized to $\emptyset$), and the branch match boolean array B (initialized to $false$ (F)).

Now, consider the action of M_3 on the data in figure 3(a). In this example, we are not able to determine if c_1 is a query solution until we test predicate satisfaction of the pattern match (a_1, b_1, c_1) for subquery $/a/b/c$.

As in PathM, when BranchM receives a SAX event (tag, $level$, id) it will send it to the machine nodes whose label is the same as tag. For example, startElement(a, 1, a_1) will be sent to the machine node v_1 since its label is a.

A machine node determines if it qualifies for a startElement(tag, $level$, id) event by comparing $level$ with the element in its parent node according to the parent edge condition, as in PathM. The $level$ of a matched active data node is then recorded in L. If the machine node is the return node, we add id to its candidate set C. Continuing the example above, machine node v_1 qualifies for this event since the label of v_1 is a and v_1 is $root$. The level information 1 is therefore recorded in v_1's L. Similarly, we record the level of b_1 (2) in v_2's L and the level of c_1 (3) in v_3's L, and put the id of c_1 into v_3's candidate set C.

When a machine node receives an endElement(tag, $level$) event, it checks if the event corresponds to the recorded data node. For any qualified machine node v, it then checks if all the components in its branch match B are $true$ (T). If so, this branch has found a match to all its predicates. If v is the root, we output its candidate set C; otherwise, we set its parent's branch match component for v to $true$, and add v's candidate set to its parent's candidate set. We then reset the qualified machine node's state to $(L = -1, C = \emptyset, B = <F, \ldots, F>)$.

Continuing the example above, when endElement(c, 3) is sent to M_3, it again matches v_3. Since v_3's branch match is trivially true (there are no qualifications on the match, and therefore B is missing for v_3 in the figure), we set its parent v_2's branch match for v_3 to T, add v_3's candidate set $\{c_1\}$ to v_2's candidate set, and reset v_3's state. We proceed in a similar fashion for the other startElement and endElement events. The snapshot after the endElement event for e_1 is shown in figure 3(c). Finally, on the endElement event for a_1, machine node v_1 qualifies; since its branch match B is all T, its candidate set $\{c_1\}$ is output as the query solution. ∎
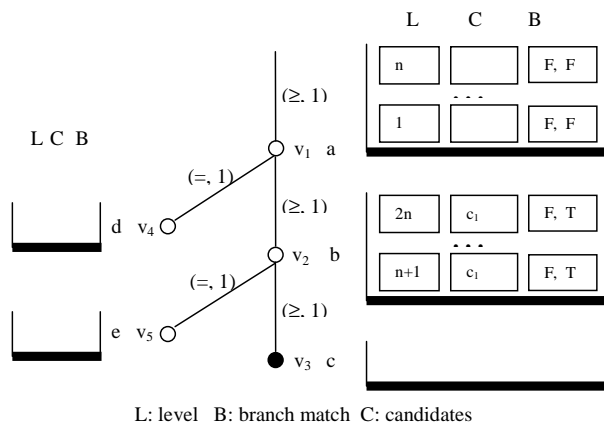

L: level   B: branch match   C: candidates

Figure 4. TwigM M_1 for Q_1 in Figure 1

3.3   Putting Them Together in TwigM

Having discussed the technical approaches for dealing with descendant axis traversal and predicate testing, we now put the two together in TwigM for evaluating queries in the language $XP^{\{/,//,*,[]\}}$ over recursive XML streams.

There are three important features of TwigM. First, as in BranchM, TwigM tests predicate satisfaction on a given pattern match using a branch match boolean array in a recursive fashion to verify a candidate query solution. Second, because of the presence of '//' and '*', TwigM builds a stack to record active XML node matches for each query node as in PathM. However, each stack element now contains the level, candidate set, and branch match array for the matched data node. The stacks compactly encode all the pattern matches to subqueries that a candidate query solution participates in. Third, since TwigM needs to test predicates on multiple pattern matches to verify each candidate, it groups pattern matches and eliminates the ones with failed predicates effectively without enumeration. Therefore TwigM only probes a polynomial number of matches in a potentially exponential search space and achieves a polynomial time complexity.

Example 3.3: Figure 4 shows the machine M_1 for query Q_1 in figure 1(b). There are five nodes in M_1, v_1, v_2, v_3, v_4 and v_5 labeled a, b, c, d and e, respectively. Each node has the level difference requirement on its parent edge as in PathM. Each node also has a stack, which is initially empty. An element of a stack has the same data structure as the state of a node in BranchM; it is a triple, representing the matching XML node's *level*, *candidate set* and *branch match*.

Now, consider the action of M_1 on the data D_1 in figure 1(a). The startElement processing is similar to PathM except a triple with level information, candidates and branch match is pushed onto the stack instead of only level

5

information. For example, when processing the data node a_1 with level 1 and id a_1, since machine node v_1 qualifies for this event, the triple $(L = 1, C = \emptyset, B = <F, F>)$ is pushed on v_1's stack. Similarly, we push corresponding information for nodes $a_2, \ldots, a_n$ on v_1's stack, nodes $b_1, \ldots, b_n$ on v_2's stack, and c_1 along with its node id as a candidate on v_3's stack.

When the endElement event of c_1 is received by M_1, the event is sent to v_3. The top element n in v_3's stack has the same level as c_1. Since n's branch match B is trivially all T, for each element in the stack of v_3's parent (v_2) whose level satisfies the parent edge condition, we set its branch match for v_3 to T, denoting that it has found a match to query child c, and upload the candidate set: $\{c_1\}$ [4]. In this example, every element in v_2's stack satisfies the parent edge condition. Then we pop v_3's stack. The snapshot at this moment is shown in figure 4, where the second component of branch matches for v_2 has been set to T to denote that the match for v_3 has been found.

When the endElement event for b_n is received, it matches the top element n in v_2's stack. However, since one component of the branch match of n is F (predicate e has not found a match), we directly pop v_3's stack. Similarly, we process the endElement events for nodes $b_{n-1}, \ldots, b_2$ and startElement events for node e_1. On the endElement event of e_1, a match is now found for the top element n of v_5's stack. Since n satisfies the parent edge condition with the top element in v_2's stack, b_1, we set b_1's branch match for v_5 to T and pop n. On the endElement event for b_1, since its branch match is all T, we set the branch matches of the elements in v_1's stack (which all satisfy the parent edge label condition) for v_2 to T, upload their candidate sets, and pop the stack. Finally, on the endElement event for a_1, since v_1's branch match is all T and it is the root, its candidate set $\{c_1\}$ is output as the query result. ∎

As we can see, TwigM uses stacks to compactly encode query pattern matches. In the above example, n^2 pattern matches ($3n^2$ elements) to subquery $//a//b//c$ in which the XML node c_1 participates are recorded using $2n + 1$ elements in stacks. To verify a candidate query solution, TwigM removes a set of unsatisfied pattern matches by popping one element in the stack. For example, since node b_n does not satisfy the predicate we pop it, eliminating all n matches in which b_n was participating $((a_i, b_n, c_1), 1 \leq i \leq n)$. When multiple satisfying pattern matches exist for a query solution, TwigM eliminates duplicate candidates by taking the union. Therefore TwigM only needs to process a polynomial number of elements in the stacks to verify a candidate query solution instead of computing all the pattern matches. In the above example, TwigM processes $2n$ elements in the stacks rather than n^2 pattern matches.

---

[4] The stacks we use allow examining all elements.

4 Algorithms

4.1 Abstract Machine TwigM

We now formally present TwigM discussed in the previous section. First we define an $XP^{\{/,//,*,[]\}}$ query Q.

Definition 4.1: An $XP^{\{/,//,*,[]\}}$ query is a tree $Q(V, \Sigma, \eta, \rho, root, \zeta, sol)$, where
- V is a finite set of nodes
- Σ is a finite alphabet of node tags
- $\eta{:}V \rightarrow \{`*'\} \cup \Sigma$ is the *name function*; $\eta(n)$ returns the name of n, which can be either a tag or '*'
- $\rho{:}V \rightarrow \{\epsilon\} \cup V$ is the *parent function*; $\rho(n)$ returns the parent node of n
- $\zeta{:}V \rightarrow \{/,//\}$ is the *parent edge function*; $\zeta(v)$ returns the label of the incoming edge of v
- $root \in V$ is the root of Q
- $sol \in V$ is the return node. ∎

TwigM machine is a tuple $(V, \eta, \rho, \zeta, I, S, \delta_s, \delta_e, root, sol)$ built from a query Q $(V', \Sigma, \eta', \rho', root', \zeta', sol')$ where
- V is a set of machine nodes corresponding to nodes in Q. $root$ and sol are machine nodes corresponding to $root'$ and sol' of Q, respectively. Several functions are defined on machine nodes.

A name function η returns the label of a machine node.

Each machine node except the $root$ has a parent which can be retrieved by a parent function ρ.

Parent edge function $\zeta{:}V \rightarrow \{=, \geq\} \times \mathbb{N}$ on a node v returns v's parent edge label, which records the level difference and axis information between v and its parent $\rho(v)$ as the condition on which an XML node should be pushed into the state of v. The first component is a function, either "$\geq$" or "$=$", depending on the axis between query node v and $\rho(v)$, and the second component is a positive integer representing level difference of between v and $\rho(v)$. We say that an integer l satisfies the *parent edge condition* of v if the function $\zeta(v)[1](l, \zeta(v)[2])$ returns $T(true)$.

A *child identity function* $\beta : V \rightarrow \mathbb{N}$ identifies a child within its parent by its order, so that the match information of the child can be recorded in its parent's branch match.
- I is the input SAX event startElement $(tag, level, id)$ or endElement $(tag, level)$ which are described by the domain of node tags Σ [5], node level Γ, and XML node ids Υ.
- S denotes the state associated with machine nodes, and is described by a stack function $\xi{:} V \rightarrow (\Gamma \times \{T, F\}^* \times \Upsilon^*)^*$. In the following we use S to denote the states $(\Gamma \times \{T, F\}^* \times \Upsilon^*)^*$. $\xi(v)$ returns a stack of active XML nodes that are solutions to the subquery from the query root to

---

[5] We assume that the alphabet of node tags in the query is the same as that of the XML data, and add ϵ for empty id in the last component of endElement events.

Algorithm 1 Functions δ_s, δ_e for TwigM

Start Element Function δ_s

1: **for all** v such that $((\eta(v) = a) \vee (\eta(v) = `*')) \wedge ((v = root) \wedge \zeta(v)[1](l, \zeta(v)[2]) \vee (v! = root) \wedge \exists e \in \xi(\rho(v))(\zeta(v)[1](l - e[1], \zeta(v)[2]))$ **do**
2:    $push(\xi(v), < l, < F, \ldots F >, \emptyset >)$;
3:    **if** $(v = sol)$ **then**
4:      $top(\xi(v))[3] = top(\xi(v))[3] \cup \{id\}$;
5:    **end if**
6: **end for**

End Element Function δ_e

1: **for all** v such that $((\eta(v) = a) \vee (\eta(v) = `*')) \wedge (top(\xi(v))[1] = l)$ **do**
2:    **if** $(\forall i((top(\xi(v))[2][i] = T))$ **then**
3:      **if** $(v = root)$ **then**
4:        $ouput(top(\xi(v))[3])$;
5:      **else**
6:        **for all** e such that $e \in \xi(\rho(v)) \wedge \zeta(v)[1](l - e[1], \zeta(v)[2])$ **do**
7:          $e[2][\beta(v)] = T$;
8:          $e[3] = e[3] \cup top(\xi(v))[3]$;
9:        **end for**
10:     **end if**
11:    **end if**
12:    $pop(\xi(v))$
13: **end for**

the corresponding query node. For a node in a stack, we record its level, branch match and candidate set information. Branch match records for each child of v whether or not a match on data has been found. The candidate set records the set of possible solutions to be verified with respect to v.

• δ_s and δ_e are transition functions corresponding to startElement and endElement events, respectively. The functions compute the next state of a machine node according to its current state and the input SAX event.

4.2 Machine Construction

Next, let us describe how to construct a TwigM for a given query Q:

• Nodes V, sol, name function η :

For each query node v' in V' whose name is an XML tag ($\eta'(v') \in \Sigma$), we build a machine node v and set $\eta(v) = \eta'(v')$; for each branching or leaf query node whose label is `*', we build a machine node $v$ and set $\eta(v) = `*'$. All v thus constructed comprise V. Let f and f' represent the mapping functions between query nodes and machine nodes, $f'(v') = v$ and $f(v) = v'$. Set $sol = f'(sol')$.

Note that we could create a machine node for *each* query node. However, we do not need to build machine nodes for interior `*' nodes since we capture them in the level difference between nodes, as described next.

• Parent function ρ, parent edge function ζ, child identity function β, $root$:

To construct ζ and ρ, for two query nodes v'_1 and v'_2, such that v'_1 is an ancestor of v'_2 with no intervening non *-nodes (that is, the path between v'_1 and v'_2 is comprised of all *-nodes), let c be the number of *-nodes between v'_1 and v'_2. Set $f'(v'_1)$ to be the parent of $f'(v'_2)$, and set the second component of the edge label between them in TwigM to be $c + 1$. If one of the edges between v'_1 and v'_2 in Q is labeled `//', then set the first component of the edge to $\geq$; otherwise set it to $=$.

We set the $root$ as the machine node without parent. For each child c of a machine node v, we set $\beta(c)$ to be the order of c within v.

• Stack function ξ

For each node $v \in V$, we build a stack and initialize it to be empty.

• Start element function δ_s

$\delta_s : I \times S \times \{=, \geq\} \times \mathbb{N} \to (V \to S)$ computes the next state of node v according to input startElement SAX events, current state of v's parent and v's parent edge label.

A startElement(a, l, id) event invokes the δ_s function on each qualified machine node v. A node v is *qualified* if: (1) $\eta(v) = a$ or $\eta(v) = `*'$; (2) v is the root and l satisfies the parent edge condition; or (3) v is not the root and there exists an element e in the stack of v's parent whose level is l' and $l - l'$ satisfies v's parent edge condition. If v is qualified, then we push $< l, < F, \ldots F >, \emptyset >$ onto v's stack. Furthermore, if v is sol then we add id to v's candidate set. δ_s is formally defined in algorithm 1.

• End element function δ_e

$\delta_e : I \times S \times \{=, \geq\} \times \mathbb{N} \to (V \to S)$ computes the next state of node v and that of its parent according to the input endElement SAX events, v's current state, and v's parent edge label.

An endElement event (a, l) invokes the δ_e function on every qualified machine node v. A node v is *qualified* if $\eta(v) = a$ or $\eta(v) = `*'$, and the level of the top element of v's stack is l.

Let n be the top element of v's stack. If v is $root$ and n's branch match is all T, we can determine that there is a pattern match for the query, therefore n's candidates are output as query solutions. If v is not $root$ and n's branch match is all T, then for every element n' in v's parent stack, such that the level difference of n and n' satisfies v's parent edge label condition, we know that n is a match to a query child of n''s, and therefore set the component of v in n''s branch match to T. Furthermore, we load the candidates of n to n' to be verified with respect to v''s subtree query. Finally, we pop v's stack. Note that if node n's branch match contains F, we not only discard n, but all the pattern matches n participates in; therefore we can remove failed pattern matches without having to enumerate them. δ_e is formally defined in

Algorithm 1.

4.3 Correctness and Complexity Analysis

Theorem 4.1: TwigM correctly evaluates queries. ∎

To prove the theorem, we first give several definitions.

Definition 4.2: Given a node v in a query tree Q, we name the subquery from $root$ to v without branches the *prefix subquery* of v. If we cut off all the branches of the nodes between root and v, excluding v, the remainder of Q is called the *suffix subquery* of v. ∎

For example, $//a//b$ is the prefix subquery of node b with respect to Q in figure 1(b), $//a//b[e]//c$ is the suffix subquery of b, however, $//a//b//c$ is not a suffix subquery of b.

Since each query node whose label is not '*' has a corresponding machine node, we blur machine nodes and query nodes in the following.

Proposition 4.2: On the startElement event for a node a, a is pushed onto a machine node v's stack if and only if a is an active node and a solution to the prefix subquery of v.
Proof Sketch: Proof by induction on the levels of the machine nodes in TwigM starting from the root. According to the construction of the parent edge function ζ, it holds immediately for the root. Assume the proposition holds for a machine node with level l. An XML node a is pushed onto the stack of node v with level $l + 1$, if and only if there is a node in v's parent's stack, and their level difference satisfies v's parent edge condition. Therefore the proposition holds. ∎

Proposition 4.3: On an endElement event for a data node a, which matches the top node n in the stack of a machine node v, we set the branch match of v to the nodes in v's parent's stack if and only if each candidate of n is a solution to the suffix subquery of v.
Proof Sketch: Proof by induction on the levels of machine nodes in TwigM starting from leaves. The base case holds according to proposition 4.2. ∎

Theorem 4.1 is a special case of proposition 4.3.

Theorem 4.4: The time complexity of TwigM is $O((|Q| + RB)|Q|\,|D|)$, where R is the depth of the XML tree.
Proof Sketch: The polynomial time complexity results from three key features of TwigM. First we use stacks to store an exponential number of pattern matches compactly. Second, before we push a node onto a stack, we check its relationship with the nodes in its parent stack, and therefore guarantee that the nodes in a stack are solutions to the prefix subquery. Third, for predicates, we only use a boolean to record its satisfaction. ∎

All the detailed proofs can be found in [10].

Name	Size	Node Number	Tag Number	Depth
Book	9MB	149K	12	20
Benchmark	34MB	616K	77	12
Protein	75MB	2277K	66	7

Figure 5. Dataset Description

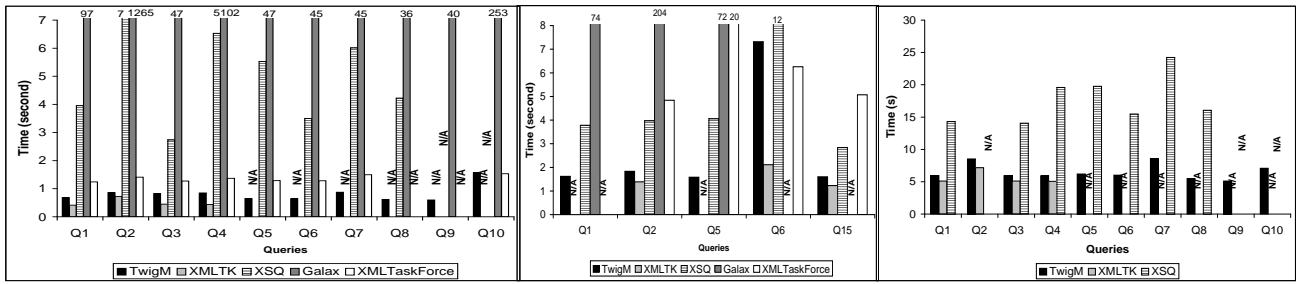| *Book* Dataset	
Q_1	//section/title
Q_2	//section//figure
Q_3	//title
Q_4	//book//section//title
Q_5	//section[./figure]/title
Q_6	//section[./section]/title
Q_7	/book//section[./title]/figure
Q_8	//section/figure/image[@source="defaultCDATA24553"]
Q_9	//section[./figure/image/@source="defaultCDATA3"]/title
Q_{10}	//section[.//section]/figure/*
Protein Dataset	
Q_1	/ProteinDatabase//protein/name
Q_2	/ProteinDatabase/ProteinEntry/*/*/*/author
Q_3	//ProteinEntry/reference/refinfo/xrefs/xref/db
Q_4	//ProteinEntry/reference//refinfo//xrefs//xref//db
Q_5	//organism[./source]
Q_6	//ProteinEntry[./reference]/@id
Q_7	//ProteinEntry//refinfo[./volume]//author
Q_8	//ProteinEntry/reference/refinfo[./year="1988"]/title
Q_9	//ProteinEntry[./refinfo[./title, ./citation/@type]]/@id
Q_{10}	//ProteinEntry/*[./created_date = "10-Sep-1999"]/uid

Figure 6. Query Sets

5 Experimental Evaluation

We have implemented TwigM in C++ and demonstrated it in [11]. The SAX parser used is Expat [12]. In this section, we present a detailed performance study of this implementation.

5.1 Experiment Setup

Environment. All experiments were conducted on a Pentium III 1.5GHz machine with 512MB memory, running the Redhat 9 distribution of GNU/ Linux(kernel 2.4.20-8). All experiments were repeated 10 times and the average processing time was calculated disregarding the maximum and minimum values.
Datasets. We conducted experiments on three datasets. The first is a synthetic dataset generated by IBM's XML Generator [18], which takes a DTD and a set of parameters as input. We use the *Book* DTD from the XQuery use cases [30] as the input DTD. We apply the default settings of XML Generator for all the parameters except for *NumberLevels* and *MaxRepeats*. *NumberLevels* bounds the maximum depth of the XML document generated and is set to 20. *MaxRepeats*

8

(a) Book Dataset   (b) Benchmark Dataset   (c) Protein Dataset

N/A means that the system doesn't support the query, take long time or report error for query evaluation.

Figure 7. Query Execution Time
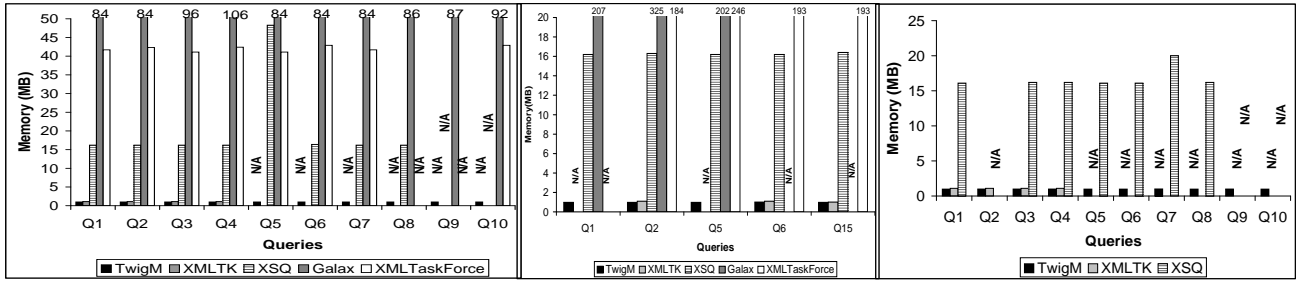
(a) Book Dataset   (b) Benchmark Dataset   (c) Protein Dataset

N/A means that the system doesn't support the query, take long time or report error for query evaluation.

Figure 8. Memory Usage

determines the maximum number of times an element can repeat in its parent and is set to 9. The second is a benchmark data set generated by XMark [31] conforming to the default auction DTD. The third one is a real dataset from the International Protein Sequence Database [15]. Features of the datasets are shown in figure 5.

Queries. We tested 10 queries on the *book* and *protein* datasets, as listed in figure 6. Q_1 to Q_4 belong to $XP^{\{/,//,*\}}$, Q_5 to Q_8 belong to $XP^{\{/,//,[]\}}$, but restrict the path expressions in predicates to be either an attribute or a single child axis. Q_8 has a value test as predicate and produces results of small sizes. Q_9 and Q_{10} belong to $XP^{\{/,//,*,[]\}}$, and allow multiple predicates to apply to a single node, path expressions in $XP^{\{/,//,*,[]\}}$ to be present in predicates, predicates to be nested, and '*'s to appear anywhere. For the benchmark dataset, we tested the benchmark queries provided by XMark [31] which only contain "/", "//", "*" and predicates. We use the original query names for the benchmark queries.

Systems. We compare TwigM with several XML query processing systems. *XMLTK* (version 1.01) [3] is a streaming XPath $XP^{\{/,//,*\}}$ processor using a DFA (Deterministic Finite Automaton) constructed lazily. *XSQ* (version 1.0) [25] is a streaming XPath $XP^{\{/,//,[]\}}$ processor using transducers, in which a predicate is restricted to be a single

child axis or an attribute, with an optional value test. The release versions of other XML query streaming processors such as SPEX [24], XSM [23], XAOS [6], TurboXPath [20] and BEA/XQRL [14] are not yet publically available. We also compare with two non-streaming XML query processors: *Galax* (release 0.3.5) [28] is a comprehensive implementation of XQuery 1.0. *XMLTaskForce* (release 2003-01-30) [16] is a main-memory, nearly complete implementation of the XPath 1 recommendation, and the only such system with polynomial time complexity in the literature.

5.2 Query Processing Time

First we compare the processing time of TwigM with XMLTK, XSQ, Galax and XMLTaskForce [6]. Figures 7(a),(b) and (c) report the query execution time for the *Book*, *Benchmark* and *Protein* datasets respectively. As we can see, for $XP^{\{/,//,*\}}$ XMLTK has the best performance; for other queries, TwigM is the fastest.

The performance of TwigM and XMLTK is stable, and does not degrade on complex queries. The performance of the other systems degrades due to enumerating multiple pattern matches to a subquery for a query result. For example,

---

[6]The processing time reported here is normalized according to benchmark [8, 2]

9
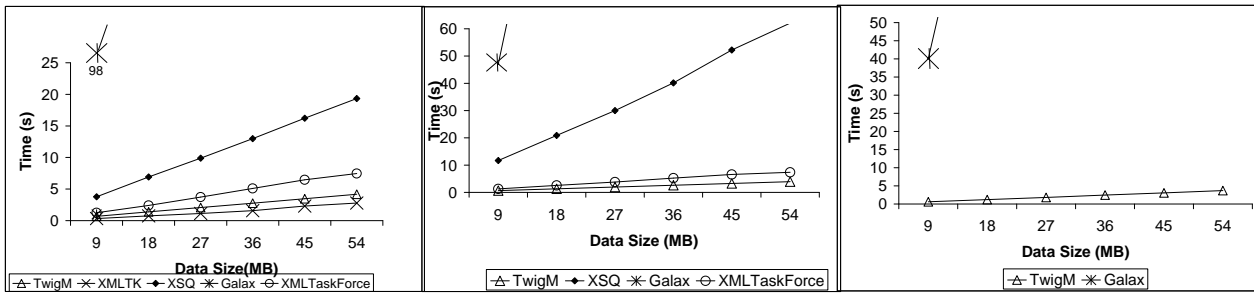
(a) Q_1 (b) Q_5 (c) Q_9

The system reports errors for missing points. Systems that are not shown in the legend do not support this query.

Figure 9. Query Execution Time as Book Data Size Increases

Q_3 and Q_4 on *Book* have the same set of nodes as query result, while each result of Q_4 has multiple pattern matches to $//book//section$ since *section* nodes are nested. TwigM and XMLTK use the same amount of processing time for both queries, the processing time of XMLTaskForce on Q_4 increases a little compared to Q_3, while the performance of XSQ and Galax degrades significantly.

As we can see, only TwigM is able to evaluate all the test queries on the three datasets. Its performance is good and stable: it performs well on recursive and non-recursive data, simple as well as complex queries. When multiple pattern matches are present (e.g. the *book* dataset), the performance advantage of TwigM is substantial. Although XMLTK outperforms TwigM on Q_1 to Q_4, the difference between their performance is small compared to the difference between TwigM's performance and that of XSQ and Galax. For queries containing multiple '*', XMLTK needs to build a DFA with an exponential number of states in the worst case and its performance degrades significantly.

5.3 Memory Usage

Next we compare the memory usage of TwigM with XMLTK, XSQ, Galax and XMLTaskForce. Process memory usage is measured using Redhat's system monitor. The total memory usage of XSQ includes memory consumed by the Java virtual machine. Figures 8(a), (b) and (c) report the memory usage for the *Book*, *Benchmark* and *Protein* datasets, respectively. There are several observations. First, the streaming processors, TwigM, XMLTK and XSQ, use substantially less memory than the non-streaming processors, Galax and XMLTaskForce, which require memory much larger than the data size. Second, as the sizes of the datasets change from 9MB(*Book*) to 34MB(*Benchmark*) to 75MB(*Protein*), the memory consumption of TwigM, XMLTK and XSQ remains roughly the same; XMLTaskForce runs out of memory for *Protein.* Third, TwigM and XMLTK use as little as 1MB memory for all queries in all datasets.
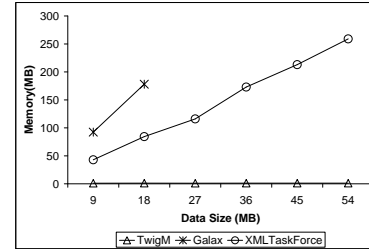
The system reports errors for missing points. Systems that are not shown in the legend do not support this query.

Figure 10. Memory Usage for Q_{10} as Book Data Size Increases

5.4 Scalability of Query Processing Time

We also measure the scalability of the systems as data size increases.

To test the scalability as the data size increases, we duplicated the *Book* dataset between 2 and 6 times. Figure 9 reports the processing time on increasing sizes of XML data for queries of different types: Q_1, Q_5 and Q_9, respectively. The performance of other queries are similar and are omitted. The results show that as the file size increases the execution time of TwigM increases very slowly for both simple and complex queries.

5.5 Scalability of Memory Usage

Figure 10 shows the memory usage of different systems as the *book* data size increases. As we can see, when the data size increases from $9MB$ to $54MB$, the memory usage of the streaming processors (TwigM, XMLTK and XSQ) is constant, while the memory consumption of Galax and XMLTaskForce increases much faster than the data size.

10

5.6 Overall Results

From the experiments, we can see that TwigM has several benefits:

- TwigM algorithm is practically efficient with guaranteed polynomial time complexity.

- TwigM has polynomial time complexity in the size of the data and query, which is verified in the experiment.

- When multiple pattern matches are present, the performance of TwigM is substantially better than other systems, as shown in the *book* dataset.

- For all the datasets in the experiments, TwigM is the most efficient query engine that handles XPath with child, descendant axes and predicates.

- TwigM is suitable for processing data streams with small memory usage. The memory consumption of TwigM remains almost constant (1MB) as the data and query sizes change in the experiments. We have also tested benchmark queries over data that is over 1GB in size, and found that the memory usage remains at 1MB.

6 Related Work

Several XPath streaming engines have been proposed. XSQ [25], SPEX [24], and XSM [23] use a hierarchical arrangement of transducers augmented with a buffer. XSQ processes XPath queries with child and descendant axes, and predicates with the restriction that predicates do not contain axes. SPEX processes regular expressions, which are similar to the XPath queries of XSQ. XSM and FluX [21] do not support descendant axis traversal. [5] analyzes the buffer requirement for evaluating XPath queries without wildcards on XML streams. In contrast, TwigM is a polynomial algorithm for XPath queries containing child axis, descendant axis, wildcards and (unrestricted) predicates.

XAOS [6] is an XPath processor that supports reverse axes (parent and ancestor) using a matching structure to store XML nodes. XAOS produces query results by traversing the matching structure at the end of the stream. In contrast, TwigM can produce results incrementally.

[20] discusses how to handle child, descendant axes, predicates and wildcards in XQuery using TurboXPath. When a recursive data node matching a query node is met, an independent thread of control is generated. Therefore the pattern matches to the query are independently recorded and manipulated explicitly.

BEA/XQRL [14] is a full implementation of XQuery. The design goal of BEA/XQRL is different from our work:

BEA/XQRL targets processing general XQuery queries on small XML messages (a few 100KB in size), while our goal is to process a commonly used subset of XPath queries efficiently on large XML streams (megabytes and gigabytes). We believe that these two systems demonstrate the trade-offs between query language expressiveness and system simplicity and efficiency.

[29] discusses how to exploit schema information to optimize XQuery evaluation on XML streams. [22] proposes XQuery rewriting techniques according to the dependency relationship between clauses in an XQuery for streaming process.

A number of XPath query filtering systems have been proposed. Filtering systems focus on determining whether or not an incoming data stream matches a large number of queries. YFilter [13] uses a single automaton to evaluate common expressions of queries to improve the performance. XTrie [9] uses a trie structure instead of a flat table to index XPath queries based on common substrings. XPush [17] lazily constructs a single deterministic pushdown automaton to filter XPath queries with predicates. [4] gives the lower bounds of XPath filter algorithms.

There are many papers on non-streaming XPath query processing. [16, 27] propose polynomial main memory algorithms for answering full XPath queries by randomly accessing an XML document. Galax [28] is a full-fledged XQuery query engine based on random accesses on a DOM model. [32, 1, 7] process tree pattern queries over a database system for XML data. Although [1, 7] also use a stack-based data structure, they focus on a different processing environment than ours, and therefore have different algorithms. First, our algorithm takes a streaming XML document as input, while the input of [1, 7] is relations of XML node labels {DocID, StartPos, EndPos, Level} with optional indices. Second, our algorithm produces results incrementally, while [1, 7] use sort and merge-join, which are blocking operations; they therefore cannot produce results incrementally. Furthermore, we focus on XPath query processing which returns the nodes matching the return node in the query; therefore it is possible to achieve a polynomial time complexity. On the other hand, the queries handled by [1, 7] return all the pattern matches.

7 Conclusions

We have discussed reasons for the potentially exponential time complexity of existing XPath streaming query processors: computing all the pattern matches for each solution of queries containing both predicates and descendant axis traversal on recursive data. Using a compact data structure to encode pattern matches, we gave a polynomial time algorithm to evaluate a large class of XPath queries, $XP^{\{/,//,*,[]\}}$, over streaming XML data. The al-

gorithm, TwigM, achieves its efficiency by searching for results lazily without enumerating all the pattern matches. A detailed experimental study shows that our approach not only has a good theoretical complexity bounds but also works well in practice on a wide variety of queries and datasets.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE*, 2002.

[2] J. Assange. Ocaml Draws with C in Trivial Benchmark. http://caml.inria.fr/archives/200008/msg00018.html/.

[3] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. XMLTK: An XML Toolkit for Scalable XML Stream Processing. In *Programming Language Technologies for XML(PLAN-X)*, 2002.

[4] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the Memory Requirements of XPath Evaluation over XML Streams. In *Proceedings of PODS*, 2004.

[5] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Buffering in Query Evaluation over XML Streams. In *Proceedings of PODS*, 2005.

[6] C. M. Barton, P. G. Charles, D. Goyal, M. Raghavachari, V. Josifovski, and M. F. Fontoura. Streaming XPath Processing with Forward and Backward Axes. In *Proceeding of ICDE*, 2003.

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

[8] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *Java Grande*, pages 97–105, 2001.

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

[10] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Streaming Processor. Technical Report MS-CIS-04-02, University of Pennsylvania, 2004.

[11] Y. Chen, S. B. Davidson, and Y. Zheng. ViteX: A Streaming XPath Processing System. In *Proceedings of ICDE*, 2005.

[12] J. Clark. The Expat XML Parser, 2003. http://expat.sourceforge.net/.

[13] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *TODS*, 28(4):467–516, 2003.

[14] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In *Proceedings of VLDB*, 2003.

[15] Georgetown Protein Information Resource. Protein Sequence Database, 2001. http://www.cs.washington.edu/research/xmldatasets/.

[16] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of VLDB*, 2002.

[17] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *Proceedings of SIGMOD*, 2003.

[18] IBM. XML Generator, 1999. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[19] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *VLDB Journal*, 11(4), 2002.

[20] V. Josifovski, M. F. Fontoura, and A. Barta. Querying XML Steams. *VLDB Journal(to appear)*, 2004.

[21] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *Proceedings of VLDB*, 2004.

[22] X. Li and G. Agrawal. Efficient Evaluation of XQuery over Streaming Data. In *Proceedings of VLDB*, 2005.

[23] B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB*, 2002.

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

[25] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proceedings of SIGMOD*, 2003.

[26] F. Peng and S. S. Chawathe. XSQ: A Streaming XPath Engine. Technical Report CS-TR-4493, University of Maryland, 2003.

[27] L. Segoufin. Typing and querying XML documents: some complexity bounds. In *Proceedings of PODS*, 2003.

[28] J. Simeon and M. Fernandez. Galax. http://db.bell-labs.com/galax.

[29] H. Su, E. A. Rundensteiner, and M. Mani. Semantic Query Optimization for XQuery over XML Streams . In *Proceedings of VLDB*, 2005.

[30] W3C. XML Query Use Cases, 2003. http://www.w3.org/TR/xquery-use-cases.

[31] XMARK the XML-benchmark project, April 2001. http://monetdb.cwi.nl/xml/index.html.

[32] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, 2001.