Online Evaluation of Regular Tree Queries

Alexandru Berlea
Technische Universität München, Germany
berlea@in.tum.de

ABSTRACT

Regular tree queries (RTQs) are a class of queries considered especially relevant for the expressiveness and evaluation of XML query languages. The algorithms proposed so far for evaluating queries online, while scanning the input data rather than by explicitly building the tree representation of the input beforehand, only cover restricted subsets of RTQs. In contrast, we introduce here an efficient algorithm for the online evaluation of unrestricted RTQs. We prove our algorithm is optimal in the sense that it finds matches at the earliest possible time for the query and the input document at hand. The time complexity of the algorithm is quadratic in the input size in the worst case and linear in many practical cases. Preliminary experimental evaluation of our practical implementation are very encouraging.

1. INTRODUCTION

Most XML applications build the tree representation of their XML input data in the main memory before processing it. This approach is not suitable for handling very large XML documents or settings in which the XML data is received linearly via some communication channel, rather than being completely available in advance. For these applications, special algorithms have to be developed, which view the XML data as a stream of events, rather than as a tree. An event contains a small piece of information, e.g. a *start-tag* or an *end-tag*. An application receiving the stream performs its task online, by reacting to the events. The advantage of this event-driven approach is that it allows one to buffer only the relevant parts of the input, saving thus time and memory. In particular, it allows the construction of the XML tree in memory and its subsequent processing, being thus at least as expressive as the tree-based approach.

A fundamental task in XML applications is locating elements in XML input data which have a desired property. Here, we call these elements *matches* and the process of locating them *querying*.

In this paper we consider the online evaluation of *regular tree queries* (RTQs) [33]. Providing efficient algorithms for this class is on the one hand side particularly important from the efficiency point of view, as it includes an important fragment of XPath [48], the most widely used XML query language. This fragment, known as Core XPath [17], mainly featuring location paths and filters using location paths but without arithmetics and data value comparisons, is considered very relevant for the efficiency of the evaluation of full XPath queries. On the other hand side, RTQs are of particular importance for XML processing as they form a class of queries, which has established itself as a benchmark for comparing expressiveness of different XML query languages [37]. This class is equally expressive with the class of queries specifiable by using monadic second order logic (MSO) and many other formalisms [36, 16, 35, 34, 28].

The research interest in querying XML streams has been very vivid recently and there is a very rich literature on this topic. Related work is reviewed in Sec. 6. Most of the related work on the online evaluation of XML queries considers only restricted fragments of Core XPath and are typically covered already by first order logic, possibly extended with *regular path expressions*, i.e. regular expressions describing the string of labels on the path from the root to the match nodes.

In contrast, we introduce an algorithm for the online evaluation of unrestricted RTQs. RTQs not only can express all of Core XPath, but they also allow a more convenient and precise specification of complex horizontal and vertical contextual conditions which are not possible or are difficult to express in XPath.

The algorithm presented here is based on a pushdown tree automaton whose transitions are guided by the events in the input data stream. Despite the large expressiveness of the queries, the algorithm is highly efficient as proven by the complexity analysis. The algorithm has been completely implemented in the freely available XML querying tool Fxgrep [32] and preliminary experimental results presented here also suggest a very good efficiency in practice. Additional merits of the algorithm are as follows.

Unrestricted

Typically, algorithms for online evaluation for a specific query language restrict the language in order to guarantee the correct application or adaption of existing algorithms in an event-based setting. In particular, algorithms based on pushdown tree automata for evaluating RTQs have been presented for the online evaluation of so-called "right-ignoring" RTQs [33]. These are quite restricted of queries for which all information needed to decide whether an element is a match has been seen by the time the end-tag of the element is encountered. In contrast, our algorithm works for arbitrary RTQs.

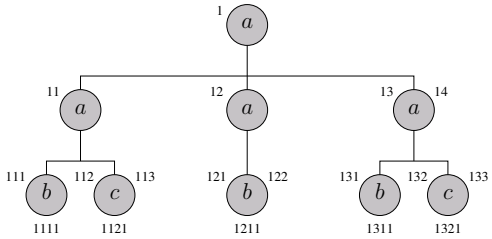

Figure 1: The tree representation of an XML document

Adaptive
Some approaches to online query processing, mostly based exclusively on the static analysis of queries, are conservative: a query which can be evaluated using the available memory resources for the input document at hand fails to be answered only because there might be other input documents in which the necessary memory can grow arbitrarily large. In contrast, our construction is *adaptive*: it adjusts its memory consumption to the requirements of the query and the input data stream at hand. The following example illustrates how the amount of memory necessary to answer a query depends on the query and on the input data.

Example 1. Consider an XML document whose tree representation is depicted in Fig. 1. Each location in the tree corresponds to an event in the stream of events, as depicted below. We identify locations by strings of numbers, s.t. the time ordering of the events corresponds to the lexicographic order of the locations:

```
 1   11  111 1111 112 1121  113  12  121 1211  122  ...
<a> <a> <b> </b> <c> </c> </a> <a> <b> </b> </a> ...
```

Consider for example the XPath query `//a/b` locating `b` nodes which have as father an `a` node. The node 111 is a match in our input. This can be detected as early as at location 111, as the events following 111 can not change the fact of 111 being a match.

The query `//a[c]/b` locates `b` nodes which have a node `a` as father and a `c` sibling. The node 111 is again a match but this becomes clear only after seeing that the `a` parent has also a child `c` at location 112. One has thus to remember 111 as a potential match between the events 111 and 112. As the events after 112 can not change the fact of 111 being a match, 111 can be reported and discarded at 112.

Finally, as an extreme case consider the query `/*[not(d)]//*` locating all descendant nodes of the root element if this has no child node `d`. Any node in the input is a potential match until seeing the last child of the root element. In our example all nodes have to be remembered as potential matches up to the last event 14. Note thus that any algorithm evaluating a query needs in the worst case linear space in the input size. However, many practical queries require a quite small amount of memory as compared to the size of the input (as argued in Sec. 5.3 and Sec. 5.4).

Optimal
The previous example also illustrates that for each element there is an earliest point in time at which one can tell whether it is a match or not. In this paper we prove the optimality of our algorithm: matches are detected as soon as possible for a given query on a given input data stream.

The paper is organized as follows. In Sec. 2 we introduce a set of useful notions and notations. In Sec. 3 we present the regular tree queries. Sec. 4 introduces *pre-order automata*. These are used by our algorithm for query evaluation on XML streams presented in Sec. 5 which further addresses its correctness, optimality, complexity and performance. Related work is discussed in Sec. 6. We conclude in Sec. 7.

2. PRELIMINARIES

Basically, XML elements are most naturally represented as ordered, labeled *unranked* trees: each element node may have an arbitrary number of successors. However, for the clarity of the presentation, it is more convenient to use an encoding of XML documents as binary trees. In this encoding every element node has two successors. The left successor contains the children of the XML element, while the right one contains its right siblings. For example, the binary representation of the XML tree in Fig. 1 is presented in Fig. 5 in Appendix B. Note that this binary encoding conserves the document order: a depth first search visits the XML elements in the same order in both encodings.

Formally, the set of *binary trees* over an alphabet Σ, denoted as $\mathcal{T}_\Sigma$ is defined by $t \in \mathcal{T}_\Sigma$ iff $t = \lambda$ or $t = a\langle t_1, t_2\rangle$ with $a \in \Sigma$ and $t_1, t_2 \in \mathcal{T}_\Sigma$, where λ denotes the empty tree. The *nodes* of a tree t, $N(t) \subseteq \{1,2\}^*$ are defined by $N(a\langle t_1, t_2\rangle) = \{\epsilon\} \cup \{1w \mid w \in N(t_1)\} \cup \{2w \mid w \in N(t_2)\}$ and $N(\lambda) = \{\epsilon\}$, where ϵ denotes the empty string. Note that the document order corresponds to the lexicographic order of the nodes. Also note that for the representation of XML documents Σ is the set of all Unicode strings, thus an infinite set. The *subtree of t located at node w*, denoted as $t[w]$ is defined by: $t[\epsilon] = t$ and $a\langle t_1, t_2\rangle[iw] = t_i[w]$. A tree t defines a *labeling function* $t : N(t) \mapsto \Sigma \cup \{\lambda\}$ as follows: $t(w) = a$ iff $t[w] = a\langle t_1, t_2\rangle$ and $t(w) = \lambda$ iff $t[w] = \lambda$. The node encoding the first XML element in the sequence of XML siblings containing the element encoded by a node n, denoted as $first(n)$, is defined by: $first(\epsilon) = \epsilon$, $first(w1) = w1$ and $first(w2) = first(w)$. The last XML sibling of a node w in a tree t, denoted as $last(w)$ (the tree will always be obvious from the context), is defined by $last(w) = w$ if $t[w] = \lambda$ and $last(w) = last(w2)$ otherwise. The set of ancestors of n is $ancestors(n) = \{w \mid ww_1 = n \text{ for some } w_1 \neq \epsilon\}$.

A (unary) *query* is a function $p : \mathcal{T}_\Sigma \mapsto 2^{\{1,2\}^*}$ s.t. $\forall t \in \mathcal{T}_\Sigma$ $p(t) \subseteq N(t)$. The elements of $p(t)$ are the *matches* of query p on input tree t.

Let $prec_t(w)$ be the set of nodes preceding a node w in document order in a tree t, formally defined as $prec_t(w) = \{w_1 \in N(t) | w_1 < w\}$, where "<" denotes lexicographic comparison. The set of *right-completions* of a tree $t \in \mathcal{T}_\Sigma$ at node $w \in N(t)$ is defined as $RightCompl_t(w) = \{t_1 \in \mathcal{T}_\Sigma \mid prec_t(w) = prec_{t_1}(w) \text{ and } \forall w_1 \in prec_t(w)\ t_1(w_1) = t(w_1)\}$. Given a query p on input t, $w \in N(t)$ is an *early detection location* for $w_1 \in N(t)$ iff $\forall t_1 \in RightCompl_t(w)\ w_1 \in p(t_1)$. An algorithm for answering a query p is *optimal* if $\forall t \in \mathcal{T}_\Sigma$ and $\forall w_1 \in p(t)$, w_1 is detected at its first early detection location in document order, which we call *earliest detection*

location. The definition of an earliest detection location w ensures thus that there is no location before w at which all relevant information for deciding whether w_1 is a match has been seen, and there is no location after w containing relevant information.

Example 2. Reconsider Example 1. Given the query `//a/b`, the earliest detection location of node 111 is 111. As for the query `//a[c]/b` the earliest detection location of node 111 is 112. Finally, for the query `/*[not(d)]//*` there is no early detection location for any match node. These matches can not be detected until the last location in the input has been reached.

3. REGULAR TREE QUERIES

A *regular tree grammar* over an alphabet Σ is a tuple $G = (X, P, x_0)$, with a finite set of variables X, a finite set of productions $P \subseteq (X \times \Sigma \times X \times X) \cup (X \times X \times X) \cup X$ and $x_0 \in X$ a *start variable*. For $(x, a, x_1, x_2) \in P$, $(x, x_1, x_2) \in P$ and $x \in P$ we write $x \to a\langle x_1, x_2\rangle$, $x \to *\langle x_1, x_2\rangle$ and $x \to \lambda$ respectively. We shorten "$x \to a\langle x_1, x_2\rangle$ or $x \to *\langle x_1, x_2\rangle$" to "$x \to (a \mid *)\langle x_1, x_2\rangle$". In the rest of the paper we use notations as above to denote elements of a grammar G which is always unambiguously identified in the context.

A variable x of a grammar G defines a relation $Deriv_x \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_X$ by: $(\lambda, x) \in Deriv_x$ iff $x \to \lambda$ and $(a\langle t_1, t_2\rangle, x\langle t_1', t_2'\rangle) \in Deriv_x$ iff $x \to (a \mid *)\langle x_1, x_2\rangle$ and $(t_i, t_i') \in Deriv_{x_i}$ for $i \in \{1, 2\}$. A variable x specifies a set of trees $[\![x]\!] = \{t \in \mathcal{T}_\Sigma \mid \exists (t, t') \in Deriv_x\}$.

Productions of the form $x \to *\langle x_1, x_2\rangle$ are necessary in order to be able to specify variables x s.t. $[\![x]\!] = \mathcal{T}_\Sigma$ since Σ is in general infinite or not a priori known (as in the case of XML input without schema information). Moreover, such universal variables are essential for the convenient specification of queries.

The set $Univ_G = \{x \mid [\![x]\!] = \mathcal{T}_\Sigma\}$ is the largest set s.t. $x \in Univ_G$ iff $x \to \lambda$, $x \to *\langle x_1, x_2\rangle$ and $x_1, x_2 \in Univ_G$, and can be computed in linear time. Testing whether $[\![x]\!] = \mathcal{T}_\Sigma$ can be thus performed very efficiently. Note the contrast with the universality problem of regular tree grammars over finite alphabets which has exponential computational complexity. Anyway, the algorithm that we are going to introduce is orthogonal w.r.t. to the universality test for variables[1].

A grammar G defines the language $\mathcal{L}_G = [\![x_0]\!]$. If $\exists (t, t') \in Deriv_{x_0}$ with $t'(w) = x$ we say that $t[w]$ may be derived from x w.r.t. G. A variable $x_\odot$ specifies a *(regular tree) query* $p_{x_\odot}$: the matches for $p_{x_\odot}$ on an input tree t are defined as $p_{x_\odot}(t) = \{w \in N(t) \mid \exists (t, t') \in Deriv_{x_0}$ with $t'(w) = x_\odot\}$. If $\exists (t, t') \in Deriv_{x_0}$ with $t'(w) = x_\odot$ we say that w is a match of $p_{x_\odot}$ w.r.t. t'.

Example 3. Consider the grammar G with start variable x_0 defined by the following productions:

$$
\begin{array}{ll}
x_\top \to *\langle x_\top, x_\top\rangle & (1) \\
x_\top \to \lambda & (2) \\
x_c \to c\langle x_\top, x_\top\rangle & (3) \\
x_b \to b\langle x_\top, x_c\rangle & (4) \\
x_0 \to a\langle x_b, x_\top\rangle & (5) \\
x_0 \to *\langle x_0, x_\top\rangle & (6) \\
x_0 \to *\langle x_\top, x_0\rangle & (7)
\end{array}
$$

Productions (1) and (2) ensure that $[\![x_\top]\!] = \mathcal{T}_\Sigma$. Production (3) makes x_c account for XML elements c with arbitrary content and arbitrary following XML siblings. Production (4) makes x_b account for XML elements b with arbitrary content and followed by an XML element c. Production (5) allows x_0 to generate XML elements a with a first XML child element b and followed by arbitrary elements. Productions (6) and (7) allows these a elements to be located arbitrarily deep under the root. The query p_{x_b} locates thus XML elements b having as parent an XML element a and as next sibling an XML element c.

Obviously, a regular tree grammar $G = (X, P, x_0)$ is equivalent with a non-deterministic top-down tree automaton with states X, transitions P and start state x_0. A tree t' is *a non-deterministic run* of G on a tree t iff $(t, t') \in Deriv_{x_0}$. Given an automaton G, the regular tree query p_x can be thus seen as a distinguished state of the automaton. These queries were proven to be equally expressive with queries definable in MSO on tree structures [16]. RTQs have thus the same expressive power as MSO queries, while being efficiently implementable, even in an event-based setting as we show in the Sec. 5.

Note that we decided for an *existential* semantics of our queries in order to cope with the non-determinism of the automaton, as needed in order to allow flexible specifications, rather then requiring the specification via deterministic tree automata, which would be an unnecessary complication of query specifications. We also decided for an *all-matches* semantics, i.e. all nodes w as in the definition are to be reported as matches, rather than, say, the first in document order. This is reasonable, because a user query typically is aimed at finding *all* locations with the specified properties, as for instance in XPath.

While very expressive, RTQs are not easily usable for users not-familiar with grammar formalisms. Alternatively, queries can be specified using the more intuitive Fxgrep [32] pattern language which is automatically translated to RTQs[2]. Fxgrep and XPath, are syntactically similar, but have different expressiveness. Fxgrep contains Core XPath and additionally allows the specification of very precise ordering constraints for nodes in patterns. In particular, any node in a pattern can be provided with two regular expressions over patterns to be fulfilled by the sequence of the node's left and right siblings, respectively. This allows for example the identification of nodes by their XML schema like types, even in the absence of schema information. For details on Fxgrep we refer the interested reader to [32].

4. PRE-ORDER AUTOMATA

[1]A more refined universality test is possible for example when schema information for the input is available, case in which the alphabet is finite. Furthermore, optimizations of our algorithm are conceivable in the presence of schema information; however they are outside the scope of this paper.

[2]The translation is straightforward but is outside of the scope here; details are given in [31, 7].

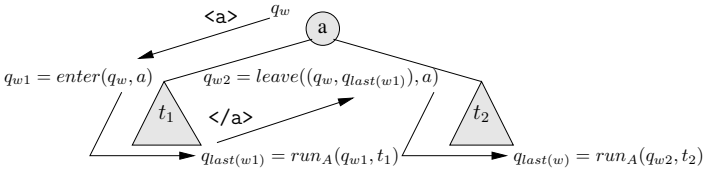

Figure 2: The processing model of a POA

A (deterministic) pre-order automaton[3] (POA) over an alphabet Σ is a tuple $A = (Q, enter, leave, F, q_0)$ consisting of a set of states Q, two total transition functions $enter : Q \times \Sigma \mapsto Q$ and $leave : Q^2 \times \Sigma \mapsto Q$, a set of final states $F \subseteq Q$ and a start state $q_0 \in Q$. A POA A defines a function $run_A : Q \times \mathcal{T}_\Sigma \mapsto Q$, which maps an input state q_i and an input tree t to an output state $q_o = run_A(q_i, t)$. In the following we denote the state in which an automaton reaches a node w by q_w. The processing model of A is illustrated in Fig. 2 on a subtree of the input t, $t[w] = a\langle t_1, t_2\rangle$, which encodes an XML a element.

Given our binary encoding, the output state may be seen as a refinement of the information q_w available when reaching the a element with information acquired by visiting its children (t_1) and its right siblings (t_2). The *enter* transition (triggered by the corresponding event `<a>`) is used to compute the start state for the left subtree t_1. The output state $q_{last(w1)}$ for t_1 is computed recursively as $run_A(q_{w1}, t_1)$. After finishing visiting t_1 (signalized by the `</a>` event after scanning the content of the a element), the input state q_{w2} for t_2 is computed via the *leave* transition using the current state $q_{last(w1)}$ and q_w. The start states q_w for the elements opened and not yet closed have thus to be remembered while scanning the content of an element. This can be easily done by using a stack to account for the fact that the XML elements occur in a last opened first closed manner. Finally, the output state for $t[w]$ is obtained by recursively processing t_2, starting with state q_{w2}. Summarizing, formally, $\forall q \in Q, \forall t \in \mathcal{T}_\Sigma$

$$run_A(q,t) = \begin{cases} q, \text{ if } t = \lambda \\ run_A(leave((q, run_A(enter(q,a), t_1)), a), t_2), \\ \quad \text{if } t = a\langle t_1, t_2\rangle \end{cases}$$

The language accepted by A is $\mathcal{L}_A = \{t \in \mathcal{T}_\Sigma \mid run_A(q_0, t) \in F\}$. The run of an automaton A can be easily implemented in an event-based manner, as presented in Appendix A.

It can be shown that POAs are equally expressive with regular tree grammars, and thus with deterministic bottom-up tree automata, but much more concise than these latter. We next give a construction which given a grammar G constructs a POA A_G, s.t. $\mathcal{L}_G = \mathcal{L}_{A_G}$. The idea of the construction is to let the state in which the automaton reaches a node w denote the variables from which w might be derived w.r.t. G when considering information from the already visited part of the tree. Consider that the automaton reaches node w in input t with $t(w) = a$ and it knows so far that $t[w]$ might be derived from some variable x_2 w.r.t. G as depicted in Fig. 3. By seeing the label a it finds the variables x_3 from which $t[w1]$ might be derived

[3]The name is due to the fact that a run of such an automaton performs a pre-order traversal over the input tree.

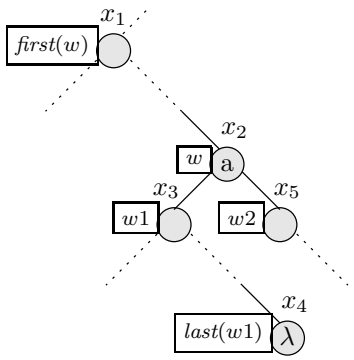

Figure 3: Variables involved at *enter* and at *leave* transitions

as $\{x_3 \mid x_2 \to (a \mid *)\langle x_3, x_5\rangle$ for some $x_5\}$. Suppose further that after recursively processing $t[w1]$, we know that its last node ($last(w1)$) might be derived from x_4 (implying that $x_4 \to \lambda$). To find out from which variables might be derived $t[w2]$, however, rather than x_4 we need to know from which variable at $w1$ this x_4 has originated. For this purpose, we have to associate variables x_4 for a node with the generating variable x_3 at $w1$. That is, rather than variables, our states will contain pairs of variables of the form (x_4, x_3). Now, when proceeding to $t[w2]$, and considering the variables x_2 and x_3 from which w and $w1$ respectively might be derived, the variables x_5 from which $t[w2]$ might be derived are computed as $\{x_5 \mid x_2 \to (a \mid *)\langle x_3, x_5\rangle\}$.

Formally, given a grammar $G = (X, P, x_0)$, the POA $A_G =$ $(Q, enter, leave, F, q_\epsilon)$ with $Q = X^2$ is defined by:

$q_\epsilon = \{(x_0, x_0)\}$
$enter(q_w, a) = \{(x$
$leave((q_w, q_{last(w1)}$
$\{(x_5, x_1) \mid (x$
x_2
$F = \{q \in Q \mid (x, x_0$

The proof that $\mathcal{L}_G = \mathcal{L}_{A_G}$ is straightforward by structural induction.

5. QUERY EVALUATION

Note that a naive direct algorithm for answering RTQs following the definition in Sec. 3 would have to individually consider all derivations w.r.t. to the underlying grammar. In the presence of recursive rules as needed for example to implement deep matching, as e.g. offered by the "//" operator, the number of derivations of a given input might be exponential in the size of the input data. In contrast, the algorithm automaton construction for the online evaluation that we introduce in this section has quadratic time complexity in the size of the input data in the very unlikely worst case and linear in many practical cases, and also a competitive space complexity.

5.1 Automata Construction

Let $G = (X, P, x_0)$ be a regular tree grammar and $p_{x_\odot}$ a regular tree query based on G. We construct a POA $A_{p_{x_\odot}} = (Q', enter', leave', F', q'_\epsilon)$ which we will use to answer the query $p_{x_\odot}$ over an input tree t.

The construction is an extension of the construction of the automaton A_G presented in the previous section. Rather than pairs from X^2 as before, the states of $A_{p_{x_\odot}}$ running over t are functions from X^2 to pairs from $2^{N(t)} \times \{true, false\}$. A mapping $q'_w(x_2, x_1) = (M, b)$ denotes:

(1) that x_2 and x_1 are as in the state q_w in which A_G reaches node w;

(2) that the nodes in M might be matches w.r.t. derivations in which w is labeled with x_2 and $first(w)$ with x_1; and,

(3) that if b is *true* then it depends only on the content of w whether there exists a derivation w.r.t. G in which w is labeled with x_2.

The construction which ensures these invariants is as follows. To ensure (1) we let the domain of the states of $A_{p_{x_\odot}}$ be constructed as the in the case of the states of A_G:

$$
\begin{aligned}
&dom(q'_\epsilon) \;=\; \{(x_0, x_0)\}\\
&dom(enter'(q'_w, a)) \;=\; enter(dom(q'_w), a)\\
&dom(leave'((q'_w, q'_{last(w1)}), a)) \;=\\
&\quad leave((dom(q'_w), dom(q'_{last(w1)})), a)\\
&F' \;=\; \{q' \in Q' \mid (x, x_0) \in dom(q'), x \to \lambda\}
\end{aligned}
$$

The invariants (2) and (3) are ensured by the following, in which whenever we write $q(x)$ for some function q we implicitly assume that $x \in dom(q)$. Initially $q'_0(x_0, x_0) = (\emptyset, true)$. The $enter'$ transition at node w is defined by $[enter'(q'_w, a)](x_3, x_3) = (M, b)$ iff M is the smallest set and b the smallest boolean value (assuming $false < true$) s.t.:

$\forall q'_w(x_2, x_1) = (M_1, b_1)$ with $x_2 \to (a \mid *)\langle x_3, x_5\rangle$ and $[\![x_5]\!] = \mathcal{T}_\Sigma$
it holds that $b_1 \leq b$
$M_1 \subseteq M$ if b_1
$w \in M$ if b_1 and $x_2 = x_\odot$

The $leave'$ transition at node w is defined by $[leave'((q'_w, q'_{last(w1)}), a)](x_5, x_1) = (M, b)$ where M and b are the smallest values s.t.:

$\forall q'_w(x_2, x_1) = (M_1, b_1)$ with $q'_{last(w1)}(x_4, x_3) = (M_2, b_2)$,
$x_4 \to \lambda$ and $x_2 \to (a \mid *)\langle x_3, x_5\rangle$
it holds that $b_1 \leq b$
$M_1 \cup M_2 \subseteq M$
$w \in M$ if $x_2 = x_\odot$

The correctness of the construction w.r.t. invariants (1)-(3) is presented immediately below.

Locating Matches

Correctness and completeness

The invariants (1)-(3) ensure that w is an early detection location for some match node w_1 iff $q'_w(x_2, x_1) = (M, true)$, $[\![x_2]\!] = \mathcal{T}_\Sigma$ and $w_1 \in M$ for some x_2 and x_1. The formal proof is given in the following theorem.

THEOREM 1. *Let t be an input tree, $p_{x_\odot}$ an input query and $A_{p_{x_\odot}}$ an automaton constructed as above. We have that $q'_w(x_2, x_1) = (M, true)$ and $[\![x_2]\!] = \mathcal{T}_\Sigma$ iff $\forall t_2 \in RightCompl_t(w)$ and $\forall w_1 \in M\ \exists (t_2, t') \in Deriv_{x_0}$ with $t'(w_1) = x_\odot$.*

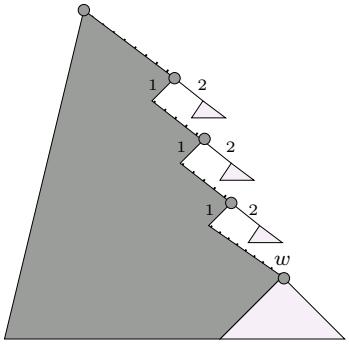

Figure 4: Locating matches

PROOF. We start by proving the correctness. Let $q'_w(x_2, x_1) = (M, true)$, $[\![x_2]\!] = \mathcal{T}_\Sigma$ and $t_2 \in RightCompl_t(w)$. As $A_{p_{x_\odot}}$ is deterministic it reaches node w in the same state q'_w for both t and t_2. Further, since $[\![x_2]\!] = \mathcal{T}_\Sigma$ we have that $t_2[w] \in [\![x_2]\!]$. We further use the following lemma which is proven in Appendix D:

LEMMA 1. *If $q'_w(x_2, x_1) = (M, true)$ and $t[w] \in [\![x_2]\!]$ then $\forall w_1 \in M\ \exists (t, t') \in Deriv_{x_0}$ with $t'(w) = x_2$ and $t'(w_1) = x_\odot$.*

It immediately follows that $\forall w_1 \in M\ \exists (t_2, t') \in Deriv_{x_0}$ with $t'(w_1) = x_\odot$.

Now we prove the completeness. Let w be an early detection location for a match node w_1 and consider that $A_{p_{x_\odot}}$ reaches a node w as depicted in Fig. 4. The part of the input already visited and the part yet to be visited are depicted in dark and light gray respectively. Let $t_2 \in RightCompl_t(w)$ be obtained by replacing each of the trees located at the nodes in $frontier(w) = \{w\} \cup \{w'2 \mid w' \in ancestors(w)\}$ (the subtrees depicted in light gray) with some arbitrary trees from $\mathcal{T}_\Sigma$. Further, let M be a set of nodes as in the hypothesis and let $w_1 \in M$. By the hypothesis $\exists t'$ s.t. $(t_2, t') \in Deriv_{x_0}$ and $t'(w_1) = x_\odot$. It follows that $\forall w' \in N(t_2)\ t_2[w'] \in [\![t'(w')]\!]$ and in particular $\forall w' \in frontier(w)\ t_2[w'] \in [\![t'(w')]\!]$. Since the latter trees were chosen arbitrarily and independently of each other it follows that $\forall w' \in frontier(w)\ [\![t'(w')]\!] = \mathcal{T}_\Sigma$. By instantiating the variables from x_1 to x_5 in the definitions of the transitions with the corresponding labels in t' it now straightforwardly follows q.e.d. □

Optimality

Every node w as in the theorem is an early detection location for any $w_1 \in M$. As the nodes w are visited in document order it immediately follows that the match nodes w_1 are detected at their earliest detection location. An algorithm implementing a run of $A_{p_{x_\odot}}$ can thus report matches as soon as the minimal necessary information from the input stream has been scanned. This is highly desirable for all querying applications and particularly for real-time applications such as monitoring or routing.

Besides allowing true matches to be reported as soon as possible, the construction also discards false matches as soon as possible. On performing the $enter'$ transition when reach-

ing an element, all potential matches M_1 for which no suitable production is found in the grammar are (implicitly) not propagated any further; the absence of any production indicates that the potential matches are incompatible with the current element. Accordingly, our construction deals very consciously with time and space resources.

In the following subsection we show that the construction can be implemented very efficiently.

5.2 Implementation

In the previous subsection we have defined a pre-order automaton and shown how this can be used to evaluate online queries. In this subsection we show how this can be efficiently implemented by using a demand-driven construction of the automaton: the states and the transitions are computed as they are needed while scanning the input stream, rather than eagerly, in advance. An eager construction of the automaton has no practical advantages over the lazy one and has on the other hand the disadvantage of an exponential blowup of the number of states to be considered.

The complete algorithm implementing the query evaluation in an event-based setting is presented in Listing 1. Basically, `enterNodeHandler` and `leaveNodeHandler` are implementations of the $enter'$ and $leave'$ transitions of $A_{p_{x_\odot}}$ and are callback functions for start and end tag events respectively. The functions `startDocHandler` and `endDocHandler` are called on beginning and ending the scan of the input respectively. We suppose here that `enterNodeHandler` and `leaveNodeHandler` receive as argument not only the label of the current element but also the corresponding node identifier. For the case in which the node identifier is not provided by the event-based parser, note that it can be easily propagated along by the event handlers.

We basically solve the equations given in the definition of the $enter'$ and $leave'$ transitions in an accumulative manner and compute the transitions lazily, as mentioned above. As an abbreviation we use the accumulative update operator $\oplus$ defined by:

$$(q \oplus [\alpha \mapsto (M,b)])(\beta) = \begin{cases} q(\beta)\ , \text{ if } \beta \neq \alpha \\ (M \cup M_1, b \text{ or } b_1)\ , \\ \quad \text{if } \beta = \alpha \text{ and } q(\beta) = (M_1, b_1) \end{cases}$$

Note the conditional expression in line 15 and line 28 by which matches are not unnecessarily propagated beyond their earliest detection location in the implementation, in contrast to the automaton construction where we ignored this only for the sake of presentation clarity. This has an additional positive influence on the memory complexity of the algorithm.

A run of $A_{p_{x_b}}$ for the query p_{x_b} presented in Example 3 is given in Appendix B.

5.3 Complexity

Let $|D|$ be the size of the input data, i.e. the number of nodes in the input tree and $|P|$ the number of productions in the grammar underlying the input query. Let pot_{max} be the maximum number of potential match nodes at any given

```
1  Stack s;
2  State q;
3
4  startDocHandler(){q := q'_0;}
5
6  enterNodeHandler(Node w, Label a){
7    s.push(q);
8
9    q_1 = ∅
10   for (x_2, x_1) ∈ dom(q) with q(x_2, x_1) = (M, b)
11     for x_2 → (a | *)⟨x_3, x_5⟩
12       if x_2 = x_⊙ then M := M ∪ {w};
13       b_1 := b & ([[x_5]] = T_Σ);
14       if b_1 & ([[x_3]] = T_Σ) then reportMatches(M);
15       q_1 =: q_1 ⊕ [(x_3, x_3) ↦ (b_1 & ([[x_3]] ≠ T_Σ) ? M : ∅, b_1)];
16
17   q := q_1;}
18
19 leaveNodeHandler(Node w, Label a){
20   q_father = s.pop();
21
22   q_1 = ∅
23   for (x_4, x_3) ∈ dom(q) with q(x_4, x_3) = (M_1, b_1) ∧ x_4 → λ
24     if b_1 then reportMatches(M_1);
25     for (x_2, x_1) ∈ dom(q_father) with q_father(x_2, x_1) = (M, b)∧
26           x_2 → (a | *)⟨x_3, x_5⟩
27       if (x_2 = x_⊙) & !(b & ([[x_5]] = T_Σ)) then M := M ∪ {w};
28       q_1 := q_1 ⊕ [(x_5, x_1) ↦ (b_1 ? M : (M_1 ∪ M), b)];
29
30   q := q_1;}
31
32 endDocHandler(){
33   for (x, x_0) ∈ dom(q) with q(x, x_0) = (M, true) and x → λ
34     reportMatches(M)}
```

Listing 1: Skeleton for the event-driven query evaluation

time during the scan of the input data and let dom_{max} be the maximal size of the domain of all states during the run of $A_{p_{x_\odot}}$.

In `enterNodeHandler` at node w, the outer loop is executed for every $(x_2, x_1) \in dom(q)$, that is $O(dom_{max})$ times, and the inner loop for every production $x_2 \to (a \mid *)\langle x_3, x_5\rangle$, that is $O(|P|)$ times. All operations in the loop body (lines 12 to 15) can be executed in time $O(pot_{max})$. A call to `enterNodeHandler` amounts thus to $O(dom_{max} \cdot |P| \cdot pot_{max})$ time. In `leaveNodeHandler`, the outer loop is executed $O(dom_{max})$ times and the inner loop is executed $O(dom_{max} \cdot |P|)$ times. Each operation within the loop takes $O(pot_{max})$ time. A call to `leaveNodeHandler` amounts thus to $O(dom_{max}^2 \cdot |P| \cdot pot_{max})$ time.

As `leaveNodeHandler` and `enterNodeHandler` are called once for every node, the overall time complexity of event driven evaluation of queries is thus in time $O(|D| \cdot dom_{max}^2 \cdot |P| \cdot pot_{max})$. The value of dom_{max} can be seen as a measure of the non-determinism of the underlying grammar and is limited by $|X|^2$ where $|X|$ is the number of variables in the grammar. The value of dom_{max} and the value of $|P|$ only depend on the query and not on the input document. The value of $|X|$ basically corresponds to the number of nodes referred to by the query and $|P|$ to the total number of qualifiers for nodes, hence they are usually small. Correspondingly, the algorithm scales well with the size of the query as presented in the next section.

Note that the pot_{max} parameter only depends on the query and the input document at hand, and not on the algorithm

used to answer the query. In this sense pot_{max} is an objective measure of the suitability of a query for online evaluation on a given input. The worst case is the unlikely case in which all nodes are potential matches until the end of the document in which case $pot_{max} = |D|$. This lead for our algorithm to quadratic time complexity in the size of the input. In general, however, in many practical cases the number of potential matches is much less than the total number of nodes ($pot_{max} \ll |D|$) and can be assimilated with a constant. In this case we obtain a time linear in the size of the document, as suggested by our experimental results.

As for the space complexity, let d be the maximal depth of the input document. During the scan of the document we store at each location the mappings q for all ancestor locations up to the root, which correspond to the opened and not yet closed elements at the current location. For every level, q has up to dom_{max} elements, each being mapped to an M which stores up to pot_{max} locations and a boolean value b. We obtain the worst case space complexity $O(d \cdot dom_{max}^2 \cdot pot_{max})$. Many of the practical queries need only a small amount of memory, as the information relevant to whether a node is a match is typically located in the relative proximity of the node (implying that pot_{max} is small). Also, while the depth of the document equals $|D|$ in the worst case (in which the input tree is degenerated to string), in practice the depth of large documents can be considered a small constant.

5.4 Experimental Results

To evaluate our algorithm in practice we implemented it in the SML programming language [27] as a part of the Fxgrep XML querying tool (version 4.6.4) which supports all fundamental features of the XML specification (e.g. attributes, text nodes, white spaces and processing instruction nodes) and provides a pattern-based front end, which makes the specification of queries more intuitive. All experiments were conducted on a 1.6 GHz Pentium M processor with 2 GB of RAM running Linux version 2.6.11. To run Fxgrep we used the SML of New Jersey [45] runtime environment version 110.0.7.

The first set of experiments was performed using the XMark benchmark set [43]. The XML input data was generated by the XMark data generator using factors 0.05, 0.1, 0.2 and 0.4. We restricted ourselves to the XMark queries expressible by Fxgrep; the remainder of the XMark queries basically either reorganize the input rather than just selecting nodes, or use value-based joins or arithmetic predicates. The results are presented in Table 1. A linear time increase in the size of the input can be observed for all tested queries. Remarkably, the query features tested by XMark: exact matching (Q_1), ordered access (Q_2 and Q_4), regular path expressions (Q_6 and Q_7), full text (Q_{14}), long path traversals (Q_{15} and Q_{16}) and missing elements (Q_{17}) do not significantly influence the evaluation times of Fxgrep. Instead, the evaluation time for Fxgrep is mainly dependent on how long potential matches are carried around until they are either confirmed or discarded. In the XMark queries, this time is relatively small, as the context information on whether a node is a match is contained in a relatively small fragment of the input situated around the node.

	Fxgrep			SPEX		
	P_1	P_2	P_3	P_1	P_2	P_3
3M	1.90	2.64	4.93	5.70	13.85	2580.39
16M	2.06	3.98	5.95	9.11	100.75	∞
32M	2.10	4.49	6.09	10.40	264.98	∞
62M	2.16	5.19	6.68	11.22	364.46	∞

Table 3: Relative evaluation times

Fxgrep is especially expressive at specifying precise contextual conditions. To account for this we performed a second set of experiments in which we used fragments of the Protein Sequence Database [41], an XML document of over 700 MB size, containing around 25 million nodes with a maximal depth of 7 and an average depth of approximately 5. We compared the performance with other XML querying tools which use both online and in-memory evaluation. Despite the large number of proposals for online evaluation of XML queries, there are surprisingly few tools publicly available. Furthermore, most of the proposals for which public implementations exist impose serious limitations on Core XPath (see Sec. 6 on related work). A more mature implementation we were able to experiment with was SPEX (version 1.0) [39] which basically covers Core XPath. As a reference for the in-memory evaluation we used Xalan-Java (version 2.7.0) [47] as a popular XSLT processor (which also provides a command line XPath processor) and Saxon (version 8.7.1) [21] as a popular XQuery processor. SPEX, Xalan and Saxon are all implemented in Java and were run in Sun's runtime environment version 1.5.0.

Even though the querying capabilities of Fxgrep go beyond those of Core XPath, for the comparison we had to limit ourselves to queries expressible in Core XPath. The queries used check for increasingly more context information as follows (their description in XPath is given in Appendix C) :

- P_1 finds authors of protein entries which contain a reference mentioning the year "2000".
- P_2 finds authors as for P_1 but additionally requires that the protein entries containing them be followed by a protein entry the description of which contains the word "iron".
- P_3 finds authors as for P_1 but asks the protein entries containing them to be followed by two entries as in P_2.

The results are presented in Table 2. The evaluation times for Fxgrep increased linearly with the input document size for all queries, as in the case the first set of experiments. In contrast, the evaluation times of the other XML processors showed a linear increase only for P_1, whereas their performance significantly degraded as the input size increased for P_2 and P_3. Their performance also apparently degraded as queries become more refined, from P_1 to P_3, whereas the ∞ signs denotes that the evaluation did not finish after 7 hours.

An absolute interpretation of the times listed in Table 2

	Q_1	Q_2	Q_4	Q_6	Q_7	Q_{14}	Q_{15}	Q_{16}	Q_{17}
5.5M	3.72	3.78	3.72	3.70	3.77	3.94	3.63	3.70	3.79
12M	7.33	7.41	7.47	7.69	7.60	7.97	7.35	7.47	7.69
23M	14.74	14.88	15.04	14.99	15.28	16.00	14.70	15.65	15.36
45M	29.24	29.75	29.79	29.79	30.38	31.83	29.23	29.74	30.55

Table 1: Evaluation times (in seconds) for XMark queries

	Fxgrep			SPEX			Saxon			Xalan		
	P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3
3M	3.55	4.94	9.23	2.06	5.00	8843.61	1.86	2.16	2.20	2.36	3.01	2.98
16M	17.68	34.16	51.01	9.44	104.38	∞	3.98	21.09	21.44	8.14	32.77	33.38
32M	36.03	76.98	104.55	18.91	481.47	∞	6.54	61.60	64.12	17.11	86.62	87.53
62M	66.55	160.21	206.27	38.47	1249.02	∞	11.35	189.15	262.34	33.05	276.70	366.45

Table 2: Evaluation times (in seconds) for increasingly refined queries for the Protein Database

is admittedly quite difficult, mainly because Fxgrep uses an SML runtime environment whereas the other XML tools uses a Java runtime environment. Java is generally known to perform better; for example, our XML parser is between 2 and 4 times slower as compared with state of the art Java parsers. A relevant measure for event-driven evaluators is obtained by dividing the evaluation time by the time needed by the underlying parser in order to generate the events; this accounts for the different paces at which events are made available in different runtime environments. The relative performance of Fxgrep was in all our test cases superior to that of SPEX as one can observe in Table 3.

As for memory usage, Fxgrep needed for all runs a space of under 30 MB including the SML runtime system and could be thus executed also on systems with little memory. SPEX needed for P_1 under 20 MB including the Java runtime environment but needed a heap larger than 64 MB to process Q_2 on input size above 32 MB. Saxon and Xalan needed a heap size 5 and 10 times larger than the XML input data, respectively, which limits their applicability for processing XML documents above 100 MB on systems with current average RAM size.

6. BIBLIOGRAPHICAL NOTES

RTQs can be evaluated using pushdown forest automata as presented in [33, 31]. The original construction generally requires to build the whole input tree in memory. The event-based query evaluation is addressed there only for right-ignoring queries. The restriction of right-ignoring queries is quite severe, as for instance they do not allow to capture XPath patterns with filters on the nodes in the path such as e.g. `//a[c]/b`. In this paper we have lifted this restriction. Rather than a priori handling only a restricted subset of queries, we have shown here how arbitrary RTQs can be evaluated in an event-driven manner.

A basic task in XML processing is XML validation. The problem of validating XML streams is addressed in [44, 11, 26]. XML schema languages are basically regular tree languages[4], hence conformance to such a schema can be checked by a pre-order automaton. As presented in Sec. 4 this can be performed efficiently on XML streams in the event-based manner.

Conventional attribute grammars (AGs) and compositions thereof are proposed by Nakano and Nishimura in [30, 38] as a means of specifying tree transformations. A deforestation method is first used to derive one AG from a composition of AGs. An algorithm is presented which allows an event-driven evaluation of the attribute values of the resulting AG. Specifying transformations, or in particular queries, using AGs is however quite elaborate even for simple context-dependent queries. The deforestation method used restricts the AGs to use attributes of non-terminal symbols at most once in a rule and the maximum nesting depth of the input trees to a statically fixed value.

More suited for XML are attribute grammars based on (extended) regular tree grammars as considered in XML Stream Attribute Grammars (XSAGs) [22] and TransformX [42]. The restricted class of L-attributed AGs is considered: the attribute values are required to be "right-ignoring", which ensures their straightforward evaluation in a single pass in document order. Another restriction is that of *unambiguousity* which basically means that there is at most one derivation with respect to the grammar and which ensures that attributes can be unambiguously specified and evaluated. While XSAGs are targeted at ensuring scalability and have the expressiveness of deterministic pushdown transducers, the TransformX AGs allow the specification of the attribution functions in a Turing-complete programming language (Java). In both cases, for the evaluation of the attribute grammars pushdown transducers are used. The pushdown transducers used in TransformX validate the input according to the grammar in a similar manner to the pre-order automata. Additionally, a sequence of attribution functions is generated as specified by the attribute grammar. A second transducer uses this sequence and performs the specified computation. For the identification of the non-terminals from which nodes are derived in the (unique) parse tree, as needed for the evaluation of the AGs in [22, 42], pre-order automata can be used. The unambiguousness restriction of the attribute grammars allows one to proceed as in the case of right-ignoring queries. That is, the non-terminal corresponding to the current node can be directly determined as

[4]The correlation between the most popular available schema languages and regular tree languages has been studied by Murata *et al.* [29].

the (single) non-terminal in the current automaton state, as it does not depend on the events after the current one.

Most research work dealing with querying of XML streams consider subsets of XPath as a query language. Some of them deal with XQuery, which is in fact more than a language for node selection as it allows the transformation of the input. In the following we are mainly interested in the node selection capabilities of the considered languages.

A number of approaches handle the problem of querying XML streams in the context of selective dissemination of information (SDI), also known as XML message brokering [9, 1, 13, 14, 2, 18, 19, 8]. In this scenario a large number of users subscribe to a dissemination system by specifying a query which acts like a filter for the documents of interest. Given an input document, the system simultaneously evaluates all user queries and distributes it to the users whose queries lead to at least one match. Strictly speaking, the queries are not answered. The documents which contain matches are dispatched but the location of the matches is not reported. XFilter [1] handle XPath patterns without nested XPath patterns as filters. These can be expressed with regular expressions, hence they are evaluated using finite string automata. YFilter [13] improves on XFilter by eliminating redundant processing by sharing common paths in patterns. In [14] the querying capabilities of YFilter are extended to handle filters comparing attributes or text data of elements with constants and nested path expressions are allowed to occur basically only for the last location step. Green *et al.* [18] consider regular path expressions without filters. A lazy construction of the deterministic finite automaton resulting from multiple XPath expressions is used in order to avoid the exponential blow-up in the number of states for a large number of queries. XPush [19] also handles nested path expressions and addresses the problem of sharing both path navigation and filter evaluation among multiple patterns. XTrie [8] considers a query language which allows the specification of nested path expressions and, besides, an order in which they are to be satisfied. Even though Fxgrep is not targeted at SDI, note that it basically exceeds the essential capabilities of all previously mentioned query languages.

The query language of XSM [25] handles only XPath patterns without deep matching (//). XSQ [40] deals with XPath patterns in which at most one filter can be specified for a node and filters cannot occur inside another filter. The filters only allow the comparison of the text content of a child element or an attribute with a constant. STX [6] is basically a restriction of the XSLT transformation language to what can be handled locally by considering only the visited part of the tree and selecting nodes from the remaining part of the tree. Sequential XPath [12] presents a subset of XPath, handling only right-ignoring XPath patterns, which can be implemented without the need of any buffering. FluXQuery [23] considers only XPath without filters and deep matching. In [24] techniques for rewriting XQuery expressions are presented in order to support their online evaluation.

Some of the approaches to online XML querying are, in contrast to ours, non-progressive: matches are reported only when reaching the end of the input stream. One of them is the already mentioned XPush. Other such approaches are ViteX [10], considering Core XPath and Chaos [5] which considers both forward and backward XPath axes.

SPEX [39] basically covers Core XPath and is based on a network of transducers. Each transducer in the network processes the input stream and transmits it augmented with computed information to its successors. The number of transducers is linear in the query size. The complexity of answering queries depends on whether filters are allowed and is polynomial in both the size of the query and of the input. XStreamQuery [15] is an XQuery engine based on a pipeline of SAX-like event handlers augmented with the possibility of returning feedback to the producer. TurboXPath [20] introduces an algorithm for answering XPath queries containing both arithmetic and structural filters which is neither directly based on automata nor on transducer networks. The time complexity of the algorithm is not addressed. The space complexity for different fragments of TurboXPath is addressed in [3] and [4]. To the best of our knowledge, in contrast to our work, none of the approaches above address the time-optimality of their match reporting.

Optimizations based on the availability of XML schema information for the input are considered in [25], YFilter [14], XPush [19], FluXQuery [23] and Raindrop [46]. Schema-based optimizations of our approach as mentioned in Sec. 3 are considered for future work.

7. CONCLUSION

We have presented an automata construction which can be used for the online evaluation of RTQs. We have introduced a formalism which allows to define the optimality of query evaluation w.r.t. to the time points at which matches are detected and formally proven that our construction is optimal in this respect. Further, we have introduced an implementation of our construction and analyzed its performance based on its theoretical time and space complexity. The algorithm has been used in the freely available Fxgrep XML query language. The efficiency of our approach has been tested in practice by experimental results comparing Fxgrep with other online and in-memory XML querying tools.

8. REFERENCES

[1] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Procedings of the 28th International Conference on Very Large Data Bases (VLDB 2000)*, Sept. 2000.

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

[3] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the Memory Requirements of XPath Evaluation over XML Streams. In *Proceedings of the 20th Symposium on Principles of Database Systems (PODS 2004)*, 2004.

[4] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Buffering in Query Evaluation over XML Streams. In *Proceedings of the 21st Symposium on Principles of Database Systems (PODS 2005)*, 2005.

[5] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josikovski. Streaming XPath

Processing with Forward and Backward Axes. In *Proceedings of the International Conference on Data Engineering (ICDE 2003)*, 2003.

[6] O. Becker. Transforming XML on the Fly. In *XML Europe 2003*, 2003.

[7] A. Berlea. *Efficient XML Processing with Tree Automata*. PhD thesis, Technical University of Munich, Munich, 2005.

[8] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the International Conference on Data Engineering (ICDE 2002)*, 2002.

[9] J. Chean, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a Scalable Continuous Query System for Internet Databases. In *Proceedings of the International Conference on Management of Data (SIGMOD 2000)*, 2000.

[10] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. In *Proceedings of the International Conference on Data Engineering (ICDE 2006)*, 2006.

[11] C. Chitic and D. Rosu. On Validation of XML Streams using Finite State Machines. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 85–90, New York, NY, USA, 2004. ACM Press.

[12] A. Desai. Introduction to Sequential XPath. In *XML Conference 2001*, Dec. 2001.

[13] Y. Diao, P. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proceedings of the International Conference on Data Engineering (ICDE 2002)*, Feb. 2002.

[14] Y. Diao and M. Franklin. YFilter: Query Processing for High-Volume XML Message Brokering. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)*, 2003.

[15] L. Fegaras. The Joy of SAX. In *Proceedings of the First International Workshop on XQuery Implementation, Experience and Perspectives*, June 2004.

[16] M. Frick, M. Grohe, and C. Koch. Query Evaluation on Compressed Trees. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*, pages 188–197, 2003.

[17] G. Gottlob, C. Koch, and R. Pichler. The Complexity of XPath Query Evaluation. In *Proceedings of the Eighteenth Symposium on Principles of Database Systems (PODS 2003)*, June 2003.

[18] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *Proceedings of International Conference on Database Theory (ICDT 2003)*, pages 173–189, 2003.

[19] A. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *Proceedings of the International Conference on Management of Data(SIGMOD 2003)*, 2003.

[20] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. *The VLDB Journal*, 14(2):197–210, 2005.

[21] M. Kay. Saxon. Software Documentation, 2005.

[22] C. Koch and S. Scherzinger. Attribute Grammars for Scalable Query Processing on XML Streams. In *Database Programming Languages (DBPL)*, pages 233–256, 2003.

[23] 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 the 30th International Conference on Very Large Data Bases (VLDB 2004)*, 2004.

[24] X. Li and G. Agrawal. Efficient evaluation of xquery over streaming data. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 265–276. VLDB Endowment, 2005.

[25] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, 2002.

[26] W. Martens, F. Neven, and T. Schwentick. Which xml schemas admit 1-pass preorder typing? In *Proceedings of International Conference on Database Theory (ICDT 2005)*, 2005.

[27] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[28] M. Murata. Extended Path Expressions for XML. In *Proceedings of the 17th Symposium on Principles of Database Systems (PODS 2001)*, 2001.

[29] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages Using Formal Language Theory. In *Extreme Markup Languages 2001, Montreal, Canada*, Aug. 2001.

[30] K. Nakano and S. Nishimura. Deriving Event-Based Document Transformers from Tree-Based Specifications. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[31] A. Neumann. *Parsing and Querying XML Documents in SML*. PhD thesis, University of Trier, Trier, 2000.

[32] A. Neumann and A. Berlea. fxgrep 4.6.1. http://www2.informatik.tu-muenchen.de/~berlea/Fxgrep/, 2005.

[33] A. Neumann and H. Seidl. Locating Matches of Tree Patterns in Forests. In V. Arvind and R. Ramamujan, editors, *Foundations of Software Technology and Theoretical Computer Science, (18th FST&TCS)*, volume 1530 of *Lecture Notes in Computer Science*, pages 134–145, Heidelberg, 1998. Springer.

[34] F. Neven. Extensions of Attribute Grammars for Structured Document Queries. *Journal of Computer and System Sciences*, 70(2):221–257, 2005.

[35] F. Neven and J. V. D. Bussche. Expressiveness of Structured Document Query Languages Based on Attribute Grammars. *Journal of the ACM*, 49(1):56–100, 2002.

[36] F. Neven and T. Schwentick. Query Automata. In *Proceedings of the Eighteenth Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 205–214. ACM Press, 1999.

[37] F. Neven and T. Schwentick. Automata- and logic-based pattern languages for tree-structured data. In K.-D. S. L. Bertossi, G. Katona and B. Thalheim, editors, *Semantics in Databases*, volume 2582 of *Lecture Notes in Computer Science*, pages 160–178, Heidelberg, 2003. Springer.

[38] S. Nishimura and K. Nakano. XML stream transformer generation through program composition and dependency analysis. *Science of Computer Programming*, 54(2–3):257–290, 2005.

[39] D. Olteanu, T. Furche, and F. Bry. Evaluating Complex Queries against XML Streams with Polynomial Combined Complexity. In *Proc. of 21st Annual British National Conference on Databases (BNCOD21)*, July 2004.

[40] F. Peng and S. S. Chawathe. XPath Queries on Streaming Data. In *Proceedings of the International Conference on Management of Data(SIGMOD 2003)*, 2003.

[41] Protein Information Ressource: The Protein Sequence Database. http://pir.georgetown.edu/home.shtml.

[42] S. Scherzinger and A. Kemper. Syntax-directed Transformations of XML Streams. In *Workshop on Programming Language Technologies for XML (PLAN-X) 2005*, 2005.

[43] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, China, August 2002.

[44] L. Segoufin and V. Vianu. Validating Streaming XML Documents. In *Symposium on Principles of Database Systems*, pages 53–64, 2002.

[45] Standard ML of New Jersey. http://www.smlnj.org/, 2006.

[46] H. Su, E. A. Rundensteiner, and M. Mani. Semantic query optimization for xquery over xml streams. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 277–288. VLDB Endowment, 2005.

[47] The Apache XML Project: Xalan-Java 2.7.0. Software Documentation, 2006.

[48] XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, Nov. 1999.

APPENDIX

A. EVENT-DRIVEN RUN OF A POA

```
Stack s, q;

startDocHandler(){
  /* called before starting reading the stream */
  q := q0;}

enterNodeHandler(Label a){
  /* called when a start-tag <a> is read */
  s.push(q);
  q := enter(q, a);}

leaveNodeHandler(Label a){
  /* called when an end-tag </a> is read */
  q_father = s.pop();
  q := leave((q_father, q), a);}

endDocHandler(){ /*on ending reading the stream*/
  if q ∈ F then output("Input accepted.")
  else output("Input rejected.");}
```

Listing 2: Event-driven run of a POA

B. SAMPLE RUN OF A POA

Example 4. Consider the query p_{x_b} presented in Example 3. The run of $A_{p_{x_b}}$ as implemented by Listing 1 is presented in Fig. 5 where *true* and *false* are abbreviated as t and f respectively. The match node 11 is recognized at its earliest detection location 1121 in the mapping $q_{1121}(x_\top, x_\top) = (t, \{11\})$. Similarly, 1221 is recognized at its earliest detection location 122121.

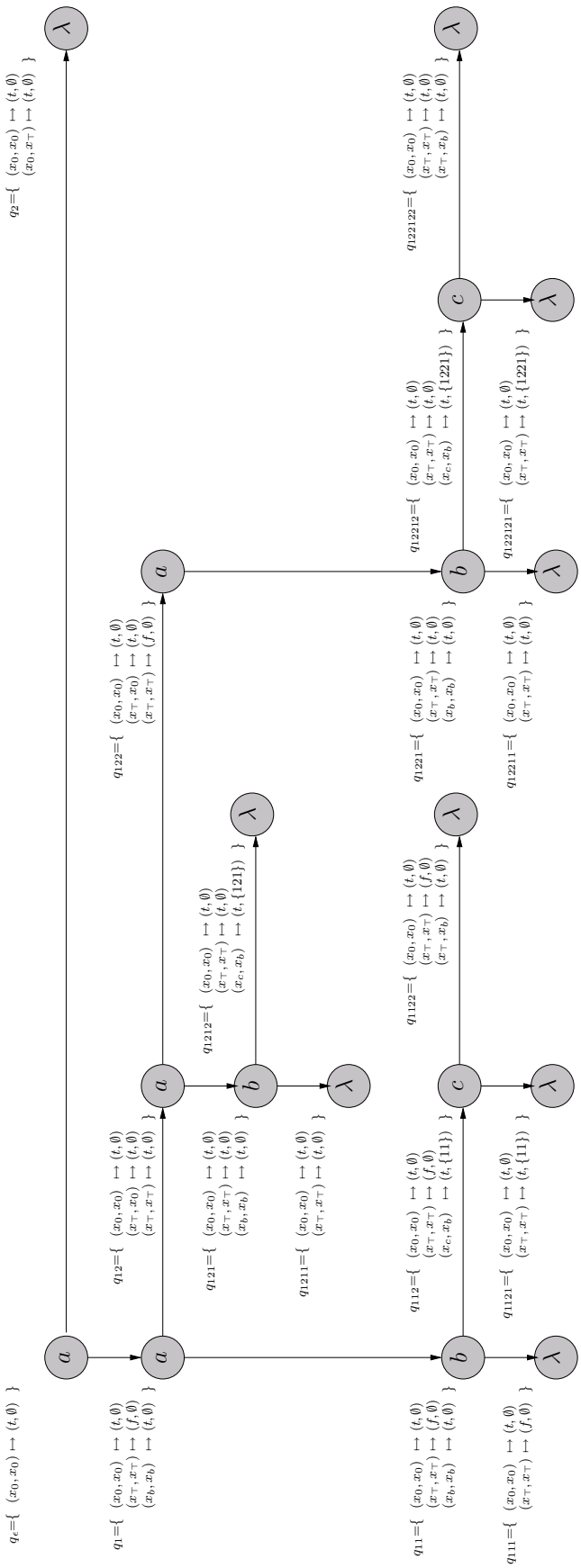

Figure 5: Sample run

C. XPATH PATTERNS USED FOR THE PROTEIN DATABASE

The XPath patterns are in Table 4.

D. PROOF OF Lemma 1

We use the notations from Theorem 1. Lemma 1 is proven immediately using the following more general lemma:

LEMMA 2. *If* $q_n(x, x_{first}) = (M, true)$ *and* $t[n] \in [\![x]\!]$ *then:*

(a) $\exists (t, t') \in Deriv_{x_0}$ *with* $t'(first(n)) = x_{first}$ *and* $t'(n) = x$

(b) $\forall n_1 \in M\ \exists (t, t') \in Deriv_{x_0}$ *with* $t'(first(n)) = x_{first}$, $t'(n) = x$ *and* $t'(n_1) = x_\odot$.

The proof is by induction on the document order of the nodes. The base case is for the root node, and $q_\epsilon = q'_0$. Let $q_\epsilon(x, x_{first}) = (M, true)$ and $t[\epsilon] \in [\![x]\!]$. Since $q_\epsilon = q'_0 = \{(x_0, x_0) \mapsto (\emptyset, true)\}$ and $t[\epsilon] = t$, it follows that $x = x_{first} = x_0$, $t \in [\![x_0]\!]$ and immediately (a) and trivially (b).

As for the induction step we distinguish the cases in which n is (A) a first or (B) a second child in the binary encoding . As far as possible, we use variable names as in Fig. 3.

In case (A) let $q_{w1}(x_3, x_{first}) = (M, true)$ and $t[w1] \in [\![x_3]\!]$. By the definition of the $enter'$ transition it follows that $x_3 = x_{first}$ and $q_w(x_2, x_1) = (M^{father}, true)$, $x_2 \to (t(w) \mid *)\langle x_3, x_5\rangle$ and $[\![x_5]\!] = \mathcal{T}_\Sigma$ for some x_2, x_1 and x_5. From $t[w1] \in [\![x_3]\!]$, $[\![x_5]\!] = \mathcal{T}_\Sigma$ and $x_2 \to (t(w) \mid *)\langle x_3, x_5\rangle$ it follows that $t[w] \in [\![x_2]\!]$. From $q_w(x_2, x_1) = (M^{father}, true)$ and $t[w] \in [\![x_2]\!]$ it follows by the induction hypothesis that $\exists (t, t') \in Deriv_{x_0}$ with $t'(first(w)) = x_1$, $t'(w) = x_2$ and $t'(n) = x_\odot\ \forall n \in M^{father}$. From $t[w1] \in [\![x_3]\!]$ it follows that $\exists (t[w1], t^1) \in Deriv_{x_3}$. From $[\![x_5]\!] = \mathcal{T}_\Sigma$ it follows that $t[w2] \in [\![x_5]\!]$ and thus $\exists (t[w2], t^2) \in Deriv_{x_5}$. From $\exists (t, t') \in Deriv_{x_0}$ with $t'(first(w)) = x_1$ and $t'(w) = x_2$, $\exists (t[w1], t^1) \in Deriv_{x_3}, t[w2] \in [\![x_5]\!]$ and $x_2 \to (t(w) \mid *)\langle x_3, x_5\rangle$ it follows that $\exists (t, t'') \in Deriv_{x_0}$, with t'' obtained by replacing in t' the subtrees $t'[w1]$ and $t'[w2]$ with t^1 and t^2 respectively, s.t. $t''(first(w1)) = t''(w1) = x_{first} = x_3$ and $t''(n_1) = x_\odot\ \forall n_1 \in M^{father}$. Now, considering the $enter'$ transitions, either $M = M^{father}$ or $M = M^{father} \cup \{w\}$ and $x_2 = x_\odot$ and thus $t''(n_1) = x_\odot\ \forall n_1 \in M$, which concludes the proof of (A).

In case (B) let $q_{w2}(x_5, x_1) = (M, true)$ and $t[w2] \in [\![x_5]\!]$. By the definition of the $leave'$ transition it follows that $q_w(x_2, x_1) = (M_1, true)$, $q_{last(w1)}(x_4, x_3) = (M_2, b)$, $x_2 \to (t(w) \mid *)\langle x_3, x_5\rangle$, $x_4 \to \lambda$ and $M = M_1 \cup M_2 \cup (x_2 = x_\odot\ ?\ \{w\}\ :\ \emptyset)$ for some x_2, x_4 and x_3. From $(x_4, x_3) \in dom(q_{last(w1)})$ and $x_4 \to \lambda$ it easily follows by induction on the number of XML right siblings of $w1$ that $t[w1] \in [\![x_3]\!]$. From $t[w1] \in [\![x_3]\!]$, $t[w2] \in [\![x_5]\!]$ and $x_2 \to (t(w) \mid *)\langle x_3, x_5\rangle$ it follows that $t[w] \in [\![x_2]\!]$. From $q_w(x_2, x_1) = (M_1, true)$ and $t[w] \in [\![x_2]\!]$ it follows by the induction hypothesis that $\exists (t, t') \in Deriv_{x_0}$ with $t'(first(w)) = x_1$ and $t'(w) = x_2$. From $t[w1] \in [\![x_3]\!]$ it follows that $\exists (t[w1], t^1) \in Deriv_{x_3}$. From $t[w2] \in [\![x_5]\!]$ it follows that $\exists (t[w2], t^2) \in Deriv_{x_2}$. Similarly as in case (A) it follows by replacing the subtrees $t'[w1]$ and $t'[w2]$ in t' with t^1 and t^2 that $\exists (t, t'') \in Deriv_{x_0}$ with $t''(first(w2)) = t''(first(w)) = x_1$ and $t''(w2) = x_5$. This concludes the proof of(a). For the proof of (b) let $n_1 \in M$. From $M = M_1 \cup M_2 \cup (x_2 = x_\odot\ ?\ \{w\}\ :\ \emptyset)$ it follows that either (i) $n_1 \in M$, or (ii) $n_1 \in M_2 \setminus M_1$, or (iii) $n_1 = w$ and $x_2 = x_\odot$. In case (i) $n_1 \in M$ it follows from $q_w(x_2, x_1) = (M_1, true)$ by the induction hypothesis that $\exists (t, t') \in Deriv_{x_0}$ with $t'(first(w)) = x_1$, $t'(w) = x_2$ and $t'(n_1) = x_\odot$. With t^1 and t^2, as in case (a) it follows q.e.d. In case (ii) $n_1 \in M_2 \setminus M_1$ it follows from $q_{last(w1)}(x_4, x_3) = (M_2, b)$ and $x_4 \to \lambda$ by the induction hypothesis that $\exists (t, t^1) \in Deriv_{x_0}$ with $t^1(w1) = x_3$, $t^1(last(w1)) = x_4$ and $t^1(n_1) = x_\odot$. From $n_1 \in M_2 \setminus M_1$ it easily follows by induction that $w1 \le n_1 < last(w1)$. With t' and t^2 as in case (a) it follows q.e.d. In case (iii) $n_1 = w$ and $x_2 = x_\odot$ it follows with t', t^1 and t^2 as in case (a) q.e.d.

P_1	*//ProteinEntry//refinfo[.//year[contains(.,'2000')]]//author*
P_2	*//ProteinEntry[following-sibling::ProteinEntry[.//description[contains(.,'iron')]]]//refinfo[.//year[contains(.,'2000')]]//author*
P_3	*//ProteinEntry[following-sibling::ProteinEntry[.//description[contains(.,'iron')]]* *[following-sibling::ProteinEntry[.//description[contains(.,'iron')]]]//refinfo[.//year[contains(.,'2000')]]//author*

Table 4: XPath patterns used in the Protein Database experiments