Processing XML Streams with Deterministic Automata

Todd J. Green[1], Gerome Miklau[2], Makoto Onizuka[3], and Dan Suciu[2]

[1] Xyleme SA, Saint-Cloud, France `todd.green@xyleme.com`
[2] University of Washington Department of Computer Science
`{gerome,suciu}@cs.washington.edu`
[3] NTT Cyber Space Laboratories, NTT Corporation, `oni@acm.org`

Abstract. We consider the problem of evaluating a large number of XPath expressions on an XML stream. Our main contribution consists in showing that Deterministic Finite Automata (DFA) can be used effectively for this problem: in our experiments we achieve a throughput of about 5.4MB/s, independent of the number of XPath expressions (up to 1,000,000 in our tests). The major problem we face is that of the size of the DFA. Since the number of states grows exponentially with the number of XPath expressions, it was previously believed that DFAs cannot be used to process large sets of expressions. We make a theoretical analysis of the number of states in the DFA resulting from XPath expressions, and consider both the case when it is constructed eagerly, and when it is constructed lazily. Our analysis indicates that, when the automaton is constructed lazily, and under certain assumptions about the structure of the input XML data, the number of states in the lazy DFA is manageable. We also validate experimentally our findings, on both synthetic and real XML data sets.

1 Introduction

Several applications of XML stream processing have emerged recently: content-based XML routing [24], selective dissemination of information (SDI) [3, 6, 9], continuous queries [7], and processing of scientific data stored in large XML files [13, 25, 19]. They commonly need to process large numbers of XPath expressions (say 10,000 to 1,000,000), on continuous XML streams, at network speed.

For illustration, consider XML Routing [24]. Here a network of *XML routers* forwards a continuous stream of XML packets from data producers to consumers. A router forwards each XML packet it receives to a subset of its output links (other routers or clients). Forwarding decisions are made by evaluating a large number of XPath filters, corresponding to clients' subscription queries, on the stream of XML packets. Data processing is minimal: there is no need for the router to have an internal representation of the packet, or to buffer the packet after it has forwarded it. Performance, however, is critical, and [24] reports very poor performance with publicly-available tools.

Our contribution here is to show that the lazy Deterministic Finite Automata (DFA) can be used effectively to process large numbers of XPath expressions, at guaranteed throughput. The idea is to convert all XPath expressions into a single DFA, then evaluate it on the input XML stream. DFAs are the most efficient means to process XPath expressions: in our experiments we measured a sustained throughput of about 5.4MB/s for arbitrary numbers of XPath expressions (up to 1,000,000 in our tests), outperforming previous techniques [3] by factors up to 10,000. But DFAs were thought impossible to use when the number of XPath expressions is large, because the size of the DFA grows exponentially with that number. We analyze here theoretically the number of states in the DFA for XPath expressions, and consider both the case when the DFA is constructed eagerly, and when it is constructed lazily. For the eager DFA, we show that the number of label wild cards (denoted $*$ in XPath) is the only source of exponential growth in the case of a single, linear XPath expression. This number, however, is in general small in practice, and hence is of little concern. For multiple XPath expressions, we show that the number of expression containing descendant axis (denoted // in XPath) is another, much more significant source of exponential growth. This makes eager DFAs prohibitive in practice. For the lazy DFA, however, we prove an upper bound on their size that is independent of the number and shape of XPath expressions, and only depends on certain characteristics of the XML stream, such as the data guide [11] or the graph schema [1, 5]. These are small in many applications. Our theoretical results thus validate the use of a lazy DFA for XML stream processing. We verify these results experimentally, measuring the number of states in the lazy DFA for several synthetic and real data sets. We also confirm experimentally the performance of the lazy DFA, and find that a lazy DFA obtains constant throughput, independent of the number of XPath expressions.

The techniques described here are part of an open-source software package[4].

Paper Organization We begin with an overview in Sec. 2 of the architecture in which the XPath expressions are used. We describe in detail processing with a DFA in Sec. 3, then discuss its construction in Sec. 4 and analyze its size, both theoretically and experimentally. Throughput experiments are discussed in Sec. 5. We discuss implementation issues in Sec. 6, and related work in Sec 7. Finally, we conclude in Sec. 8.

2 Overview

2.1 The Event-Based Processing Model

We start by describing the architecture of an XML stream processing system [4], to illustrate the context in which XPath expressions are used. The user specifies several correlated XPath expressions arranged in a tree, called the *query tree*. An input XML stream is first parsed by a SAX parser that generates a stream of *SAX events* (Fig. 1); this is input to the query processor that evaluates the

---

[4] Described in [4] and available at `xmltk.sourceforge.net`.

XPath expressions and generates a stream of *application events*. The application is notified of these events, and usually takes some action such as forwarding the packet, notifying a client, or computing some values. An optional Stream Index (called SIX) may accompany the XML stream to speed up processing [4]: we do not discuss the index here.

The query tree, Q, has nodes labeled with variables and the edges with linear XPath expressions, P, given by the following grammar:

$$P ::= /N \mid //N \mid PP \qquad\qquad N ::= E \mid A \mid \mathit{text}(S) \mid * \tag{1}$$

Here E, A, and S are an element label, an attribute label, and a string constant respectively, and $*$ is the wild card. The function `text(S)` matches a text node whose value is the string `S`. While filters, also called predicates, are not explicitly allowed, we show below that they can be expressed. There is a distinguished variable, $\$R$, which is always bound to the root. We leave out from our presentation some system level details, for example the fact that the application may specify under which application events it wants to receive the SAX events. We refer the reader to [4] for system level details.

Example 1. The following is a query tree (tags taken from [19]):

```
$D IN $R/datasets/dataset   $H  IN $D/history
$T IN $D/title              $TH IN $D/tableHead
$N IN $D//tableHead//*      $F  IN $TH/field
$V IN $N/text("Galaxy")
```

Fig. 2 shows this query tree graphically. Fig. 3 shows the result of evaluating this query tree on an XML input stream: the first column shows the XML stream, the second shows the SAX events generated by the parser, and the last column shows the application events.

Filters Currently our query trees do not support XPath expressions with filters (a.k.a. predicates). One can easily implement filters over query trees in a naive way, as we illustrate here on the following XPath expression:

```
$X IN $R/catalog/product[@category="tools"][sales/@price > 200]/quantity
```

First decompose it into several XPath expression, and construct the query tree Q in Fig. 4. Next we use our query tree processor, and add the following actions. We declare two boolean variables, `b1, b2`. On a `$Z` event, set `b1` to `true`; on a `$U` event test the following text value and, if it is > 200, then set `b2` to `true`. At the end of a `$Y` event check whether `b1=b2=true`. This clearly implements the two filters in our example. Such a method can be applied to arbitrary filters and predicates, with appropriate bookkeeping, but clearly throughput will decrease with the number of filters in the query tree. Approaches along these lines are discussed in [3, 6, 9]. More advanced methods for handling filters include event detection techniques [20] or pushdown automata [21].

The Event-based Processing Problem The problem that we address is: given a query tree Q, preprocesses it, then evaluate it on an incoming XML

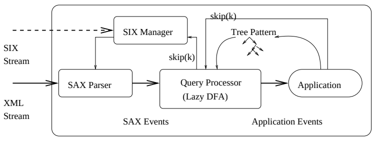


Fig. 1. System's Architecture

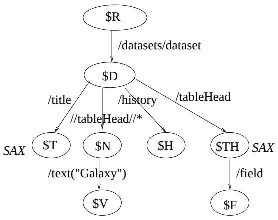


Fig. 2. A Query Tree

XML Stream	Parser SAX Events	Variable Events
`<datasets>`	`start(datasets)`	`start($R)`
`<dataset>`	`start(dataset)`	`start($D)`
`<history>`	`start(history)`	`start($H)`
`<date>`	`start(date)`	
10/10/59	`text("10/10/59")`	
`</date>`	`end(date)`	
`</history>`	`end(history)`	`end($H)`
`<title>`	`start(title)`	`start($T)`
`<subtitle>`	`start(subtitle)`	
Study	`text(Study)`	
`</subtitle>`	`end(subtitle)`	
`</title>`	`end(title)`	
		`end($T)`
...		
`</dataset>`	`end(dataset)`	`end($D)`
...	...	
`</datasets>`	`end(datasets)`	`end($R)`

Fig. 3. Events generated by a Query Tree

stream. The goal is to maximize the throughput at which we can process the XML stream. A special case of a query tree, Q, is one in which every node is either the root or a leaf node, i.e. has the form: $\mathtt{\$X_1}$ in $\mathtt{\$R/e_1}$, $\mathtt{\$X_2}$ in $\mathtt{\$R/e_2}, \ldots,$ $\mathtt{\$X_p}$ in $\mathtt{\$R/e_p}$ (each $\mathtt{e_i}$ may start with // instead of /): we call Q a *query set*, or simply a *set*. Each query tree Q can be rewritten into an equivalent query set Q', as illustrated in Fig. 4.

```
Q:                                 Q':
$Y IN $R/catalog/product           $Y IN $R/catalog/product
$Z IN $Y/@category/text("tools")   $Z IN $R/catalog/product/@category/text("tools")
$U IN $Y/sales/@price              $U IN $R/catalog/product/sales/@price
$X IN $Y/quantity                  $X IN $R/catalog/product/quantity
```

Fig. 4. A query tree Q and an equivalent query set Q'.

3 Processing with DFAs

3.1 Background on DFAs

Our approach is to convert a query tree into a Deterministic Finite Automaton (DFA). Recall that the query tree may be a very large collection of XPath expressions: we convert *all* of them into a *single* DFA. This is done in two steps: convert the query tree into a Nondeterministic Finite Automaton (NFA), then convert the NFA to a DFA. We review here briefly the basic techniques for both steps and refer the reader to a textbook for more details, e.g. [14]. Our running example will be the query tree P shown in Fig. 5(a). The NFA, denoted A_n, is illustrated in Fig. 5(b). Transitions labeled $*$ correspond to $*$ or // in P; there is one initial state; there is one terminal state for each variable (`$X`, `$Y`, ...); and there are ε-transitions [5]. It is straightforward to generalize this to any query tree. The number of states in A_n is proportional to the size of P.

Let Σ denote the set of all tags, attributes, and text constants occurring in the query tree P, plus a special symbol ω representing any other symbol that could be matched by $*$ or //. For $w \in \Sigma^*$ let $A_n(w)$ denote the set of states in A_n reachable on input w. In our example we have $\Sigma = \{a, b, d, \omega\}$, and $A_n(\varepsilon) = \{1\}$, $A_n(ab) = \{3, 4, 7\}$, $A_n(a\omega) = \{3, 4\}$, $A_n(b) = \emptyset$.

The DFA for P, A_d, has the following set of states:

$$\mathit{states}(A_d) = \{A_n(w) \mid w \in \Sigma^*\} \tag{2}$$

For our running example A_d is illustrated[6] in Fig. 5 (c). Each state has unique transitions, and one optional `[other]` transition, denoting any symbol in Σ *except* the explicit transitions at that state: this is different from $*$ in A_n which denotes *any* symbol. For example `[other]` at state $\{3, 4, 8, 9\}$ denotes either a or ω, while `[other]` at state $\{2, 3, 6\}$ denotes a, d, or ω. Terminal states may be labeled now with more than one variable, e.g. $\{3, 4, 5, 8, 9\}$ is labeled `$Y` and `$Z`.

3.2 The DFA at Run time

Processing an XML stream with a DFA is very efficient. We maintain a pointer to the current DFA state, and a stack of DFA states. SAX events are processed as follows. On a `start(element)` event we push the current state on the stack, and replace the state with the state reached by following the `element` transition[7]; on an `end(element)` we pop a state from the stack and set it as the current state. Attributes and `text(string)` are handled similarly. No memory management is needed at run time[8]. Thus, each SAX event is processed in $O(1)$ time, and we can guarantee the throughput, independent of the number of XPath expressions. The main issue is the size of the DFA, which we discuss next.

---

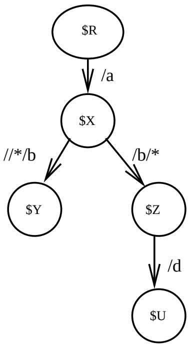
[5] These are needed to separate the loops from the previous state. For example if we merge states 2, 3, and 6 into a single state then the $*$ loop (corresponding to //) would incorrectly apply to the right branch.

[6] Technically, the state $\emptyset$ is also part of the DFA, and behaves like a "failure" state, collecting all missing transitions. We do not illustrate it in our examples.

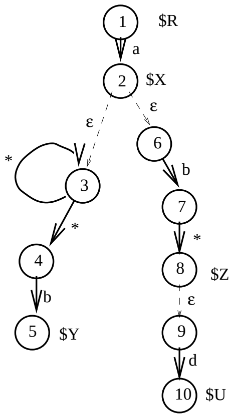
[7] The state's transitions are stored in a hash table.

[8] The stack is a static array, currently set to 1024: this represents the maximum XML depth that we can handle.

```
$X IN $R/a
$Y IN $X//*/b
$Z IN $X/b/*
$U IN $Z/d
```



(a)



(b)