February 2003

Processing XML Streams with Deterministic Automata

Todd J. Green
Xyleme SA, tjgreen@cis.upenn.edu

Gerome Miklau
University of Washington

Makoto Onizuka
NTT Cyber Space Laboratories

Dan Suciu
University of Washington

Postprint version. Published in *Lecture Notes in Computer Science*, Volume 2572, Database Theory: - ICDT 2003: 9th International Conference Proceedings, 2003, pages 173-189.
Publisher URL: http://www.springerlink.com/content/crj2xv1em5vnyyf6/

At the time of publication, author Todd Green, was affiliated with Xyleme SA. Currently February 2007 he is a student in the Department of Computer and Information Sciences at the University of Pennsylvania.

Processing XML Streams with Deterministic Automata

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.

Comments
Postprint version. Published in *Lecture Notes in Computer Science*, Volume 2572, Database Theory: - ICDT 2003: 9th International Conference Proceedings, 2003, pages 173-189.
Publisher URL: http://www.springerlink.com/content/crj2xv1em5vnyyf6/

At the time of publication, author Todd Green, was affiliated with Xyleme SA. Currently February 2007 he is a student in the Department of Computer and Information Sciences at the University of Pennsylvania.

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:

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

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

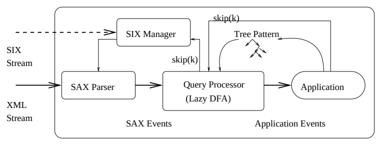


Fig. 1. System's Architecture

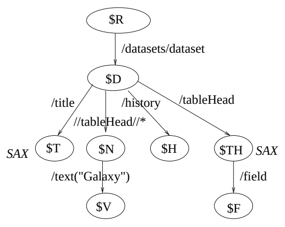


Fig. 2. A Query Tree

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: $\texttt{\$X}_1$ `in` `$R/`$\texttt{e}_1$, $\texttt{\$X}_2$ `in` `$R/`$\texttt{e}_2, \ldots,$ $\texttt{\$X}_\texttt{p}$ `in` `$R/`$\texttt{e}_\texttt{p}$ (each $\texttt{e}_\texttt{i}$ 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`, $\ldots$); 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:

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

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.

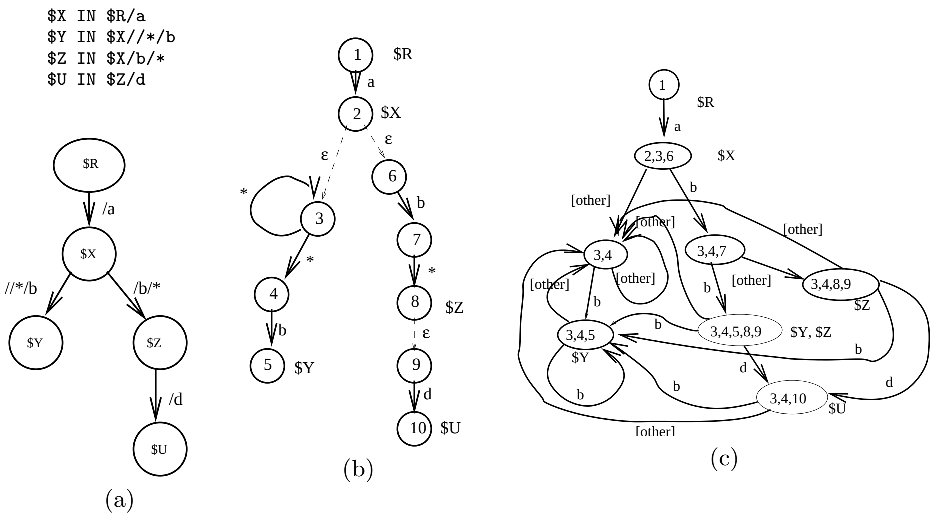


Fig. 5. (a) A query tree; (b) its NFA, A_n, and (c) its DFA, A_d.

4 Analyzing the Size of the DFA

For a general regular expression the size of the DFA may be exponential [14]. In our setting, however, the expressions are restricted to XPath expressions defined in Sec. 2.1, and general lower bounds do not apply automatically. We analyze and discuss here the size of the eager and lazy DFAs for such XPath expressions. We shall assume first that the XPath expressions have no text constants (`text(S)`) and, as a consequence, the alphabet Σ is small, then discuss in Sec. 4.4 the impact of the constants on the number of states. As discussed at the end of Sec.2 we will restrict our analysis to query trees that are sets.

4.1 The Eager DFA

Single XPath Expression A linear XPath expression has the form $P = p_0//p_1//\ldots//p_k$ where each p_i is $N_1/N_2/\ldots/N_{n_i}$, $i = 0,\ldots,k$, and each N_j is given by (1). We consider the following parameters:

$$\begin{array}{ll} k = \text{ number of } //\text{'s} & n_i = \text{length of } p_i,\ i = 0,\ldots,k \\ m = \text{max \# of } *\text{'s in each } p_i & n = \text{length of } P,\ \sum_{i=0,k} n_i \\ s = \text{ alphabet size } = \mid \Sigma \mid & \end{array}$$

For example if $P = //a/*//a/*/b/a/*/a/b$, then $k = 2$ ($p_0 = \varepsilon$, $p_1 = a/*$, $p_2 = a/*/b/a/*/a/b$), $s = 3$ ($\Sigma = \{a,b,\omega\}$), $n = 9$ (node tests: $a,*,a,*,b,a,*,a,b$), and $m = 2$ (we have 2 $*$'s in p_2). The following theorem gives an upper bound on the number of states in the DFA, and is, technically, the hardest result in the paper. The proof is provided in the appendix.

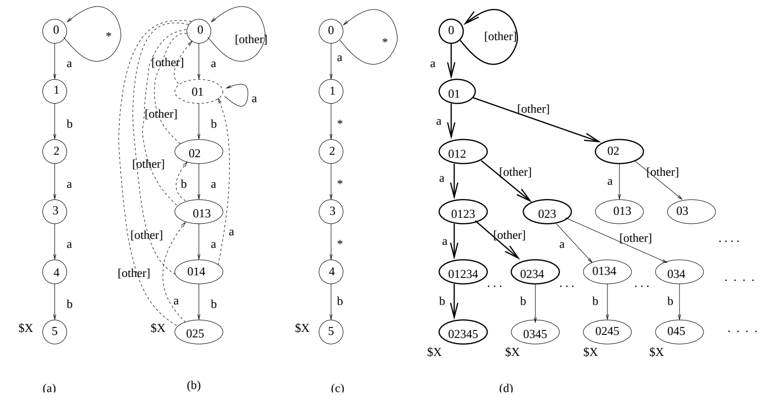


Fig. 6. The NFA (a) and the DFA (b) for `//a/b/a/a/b`. The NFA (c) and the DFA (with back edges removed) (d) for `//a/*/*/*/b`: here the eager DFA has $2^5 = 32$ states, while the lazy DFA, assuming the DTD `<!ELEMENT a (a*|b)>`, has at most 9 states.

Theorem 1. *Given a linear XPath expression P, define prefix*$(P) = n_0$ *and suffix*$(P) = k + k(n - n_0)s^m$*. Then the eager DFA for P has at most prefix*$(P)+$ *suffix*(P) *states. In particular, when* $k = 0$ *the DFA has at most n states, and when* $k > 0$ *the DFA has at most* $k + kns^m$ *states.*

We first illustrate the theorem in the case where there are no wild-cards ($m = 0$); then there are at most $k+kn$ states in the DFA. For example, if $p = //a/b/a/a/b$, then $k = 1, n = 5$: the NFA and DFA shown in Fig. 6 (a) and (b), and indeed the latter has 6 states. This generalizes to $//N_1/N_2/\ldots/N_n$: the DFA has only $n + 1$ states, and is an isomorphic copy of the NFA plus some back transitions: this corresponds to Knuth-Morris-Pratt's string matching algorithm [8].

When there are wild cards ($m > 0$), the theorem gives an exponential upper bound. There is a corresponding exponential lower bound, illustrated in Fig. 6 (c), (d), showing that the DFA for $p = //a/*/*/*/b$, has 2^5 states. It is easy to generalize this example and see that the DFA for $//a/*/\ldots/*/b$ has 2^{m+2} states[9], where m is the number of $*$'s.

Thus, the theorem shows that the only thing that can lead to an exponential growth of the DFA is the maximum number of $*$'s between any two consecutive $//$'s. One expects this number to be small in most practical applications; arguably users write expressions like `/catalog//product//color` rather than `/catalog//product/*/*/*/*/*/*/*/*/*/color`. Some implementations of XQuery already translate a *single* linear XPath expression into DFAs [15].

Multiple XPath Expressions For sets of XPath expressions, the DFA also grows exponentially with the number expressions containing $//$. We illustrate first, then state the lower and upper bounds.

Example 2. Consider four XPath expressions:

[9] The theorem gives the upper bound: $1 + (m + 2)3^m$.

```
$X1 IN $R//book//figure      $X2 IN $R//table//figure
$X3 IN $R//chapter//figure   $X4 IN $R//note//figure
```

The eager DFA needs to remember what subset of tags of {`book`, `table`, `chapter`, `note`} it has seen, resulting in at least 2^4 states. We generalize this below.

Proposition 1. *Consider p XPath expressions:* `$X`$_1$ `IN $R//a`$_1$`//b` ... `$X`$_p$ `IN $R//a`$_p$`//b` *where $a_1, \ldots, a_p, b$ are distinct tags. Then the DFA has at least 2^p states.*[10]

Theorem 2. *Let Q be a set of XPath expressions. Then the number of states in the eager DFA for Q is at most: $\sum_{P \in Q}(\mathit{prefix}(P)) + \prod_{P \in Q}(1 + \mathit{suffix}(P))$ In particular, if A, B are constants s.t. $\forall P \in Q$, $\mathit{prefix}(P) \leq A$ and $\mathit{suffix}(P) \leq B$, then the number of states in the eager DFA is $\leq p \times A + B^{p'}$, where p' is the number of XPath expressions $P \in Q$ that contain //.*

Recall that *suffix*(P) already contains an exponent, which we argued is small in practice. The theorem shows that the extra exponent added by having multiple XPath expressions is precisely the number of expressions with //'s. This result should be contrasted with Aho and Corasick's dictionary matching problem [2, 22]. There we are given a dictionary consisting of p words, $\{w_1, \ldots, w_p\}$, and have to compute the DFA for the set $Q = \{//w_1, \ldots, //w_p\}$. Hence, this is a special case where each XPath expression has a single, leading //, and has no $*$. The main result in the dictionary matching problem is that the number of DFA states is linear in the total size of Q. Theorem 2 is weaker in this special case, since it counts each expression with a // toward the exponent. The theorem could be strengthened to include in the exponent only XPath expressions with at least two //'s, thus technically generalizing Aho and Corasick's result. However, XPath expressions with two or more occurrences of // *must* be added to the exponent, as Proposition 1 shows. We chose not to strengthen Theorem 2 since it would complicate both the statement and proof, with little practical significance.

Sets of XPath expressions like the ones we saw in Example 2 are common in practice, and rule out the eager DFA, except in trivial cases. The solution is to construct the DFA lazily, which we discuss next.

4.2 The Lazy DFA

The *lazy DFA* is constructed at run-time, on demand. Initially it has a single state (the initial state), and whenever we attempt to make a transition into a missing state we compute it, and update the transition. The hope is that only a small set of the DFA states needs to be computed.

This idea has been used before in text processing, but it has never been applied to such large number of expressions as required in our applications (e.g. 100,000): a careful analysis of the size of the lazy DFA is needed to justify its feasibility. We prove two results, giving upper bounds on the number of states

[10] Although this requires p distinct tags, the result can be shown with only 2 distinct tags, and XPath expressions of depths $n = O(\log p)$, using standard techniques.

in the lazy DFA, that are specific to XML data, and that exploit either the schema, or the data guide. We stress, however, that neither the schema nor the data guide need to be known in order to use the lazy DFA, and only serve for the theoretical results.

Formally, let A_l be the lazy DFA. Its states are described by the following equation which should be compared to Eq.(2):

$$\mathit{states}(A_l) = \{A_n(w) \mid w \in \mathcal{L}_{data}\} \tag{3}$$

Here $\mathcal{L}_{data}$ is the set of all root-to-leaf sequences of tags in the input XML streams. Assuming that the XML stream conforms to a schema (or DTD), denote $\mathcal{L}_{schema}$ all root-to-leaf sequences allowed by the schema: we have $\mathcal{L}_{data} \subseteq \mathcal{L}_{schema} \subseteq \Sigma^*$.

We use *graph schema* [1, 5] to formalize our notion of schema, where nodes are labeled with tags and edges denote inclusion relationships. Define a simple cycle, c, in a graph schema to be a set of nodes $c = \{x_0, x_1, \ldots, x_{n-1}\}$ which can be ordered s.t. for every $i = 0, \ldots, n-1$, there exists an edge from x_i to $x_{i+1 \mod n}$. We say that a graph schema is *simple*, if for any two cycles $c \neq c'$, we have $c \cap c' = \emptyset$.

We illustrate with the DTD in Fig. 7, which also shows its graph schema [1]. This DTD is simple, because the only cycles in its graph schema (shown in Fig. 7 (a)) are self-loops. All non-recursive DTDs are simple. For a simple graph schema we denote d the maximum number of cycles that a simple paths can intersect (hence $d = 0$ for non-recursive schemes), and D the total number of nonempty, simple paths: D can be thought of as the number of nodes in the unfolding[11]. In our example $d = 2$, $D = 13$, and the unfolded graph schema is shown in Fig. 7 (b). For a query set Q, denote n its depth, i.e. the maximum number of symbols in any $P \in Q$ (i.e. the maximum n, as in Sec. 4.1). We prove the following result in the appendix:

Theorem 3. *Consider a simple graph schema with d, D, defined as above, and let Q be a set of XPath expressions of maximum depth n. Then the lazy DFA has at most $1 + D \times (1+n)^d$ states.*

The result is surprising, because the number of states does not depend on the number of XPath expressions, only on their depths. In Example 2 the depth is $n = 2$: for the DTD above, the theorem guarantees at most $1 + 13 \times 3^2 = 118$ states in the lazy DFA. In practice, the depth is larger: for $n = 10$, the theorem guarantees ≤ 1574 states, even if the number of XPath expressions increases to, say, 100,000. By contrast, the eager DFA has $\geq 2^{100000}$ states (see Prop. 1). Fig. 6 (d) shows another example: of the 2^5 states in the eager DFA only 9 are expanded in the lazy DFA.

[11] The constant D may, in theory, be exponential in the size of the schema because of the unfolding, but in practice the shared tags typically occur at the bottom of the DTD structure (see [23]), hence D is only modestly larger than the number of tags in the DTD.

Theorem 3 has many applications. First for *non-recursive* DTDs ($d = 0$) the lazy DFA has at most $1 + D$ states[12]. Second, in *data-oriented* XML instances, recursion is often restricted to hierarchies, e.g. departments within departments, parts within parts. Hence, their DTD is simple, and d is usually small. Finally, the theorem also covers applications that handle documents from *multiple* DTDs (e.g. in XML routing): here D is the sum over all DTDs, while d is the maximum over all DTDs.

```
<!ELEMENT book (chapter*)>
<!ELEMENT chapter (section*)>
<!ELEMENT section ((para|table|note|figure)*)>
<!ELEMENT table ((table|text|note|figure)*)>
<!ELEMENT note ((note|text)*)>
```

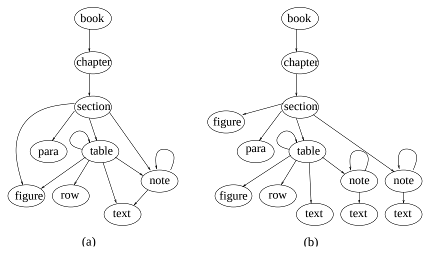


Fig. 7. A graph schema for a DTD (a) and its unfolding (b).

DTD Source	DTD Names	(DTD Statistics)		Data size MB
		No. elms	Simple ?	
`[synthetic]`	`simple.dtd`	12	Yes	-
`www.wapforum.org`	`prov.dtd`	3	Yes	-
`www.ebxml.org`	`ebBPSS.dtd`	29	Yes	-
`pir.georgetown.edu`	`protein.dtd`	66	Yes	684
`xml.gsfc.nasa.gov`	`nasa.dtd`	117	No	24
`UPenn Treebank`	`treebank.dtd`	249	No	56

Fig. 8. Sources of data used in experiments. Only three real data sets were available.

The theorem does not apply, however, to *document-oriented* XML data. These have non-simple DTDs : for example a `table` may contain a `table` or a `footnote`, and a `footnote` may also contain a `table` or a `footnote` (hence, both {`table`} and {`table`, `footnote`} are cycles, and they share a node). For such cases we give an upper bound on the size of the lazy DFA in terms of *Data Guides* [11]. The data guide is a special case of a graph schema, with $d = 0$, hence Theorem 3 gives:

Corollary 1. *Let G be the number of nodes in the data guide of an XML stream. Then, for any set Q of XPath expressions the lazy DFA for Q on that XML stream has at most $1 + G$ states.*

An empirical observation is that real XML data tends to have small data guides, regardless of its DTD. For example users occasionally place a `footnote` within a `table`, or vice versa, but do not nest elements in all possible ways allowed by the schema. All XML data instances described in [16] have very small data guides, except for Treebank [17], where the data guide has $G = 340,000$ nodes.

[12] This also follows directly from (3) since in this case $\mathcal{L}_{schema}$ is finite and has $1 + D$ elements: one for $w = \varepsilon$, and one for each non-empty, simple paths.

Using the Schema or DTD If a Schema or DTD is available, it is possible to specialize the XPath expressions and remove all $*$'s and $//$'s, and replace them with general Kleene closures: this is called *query pruning* in [10]. For example for the schema in Fig. 7 (a), the expression `//table//figure` is pruned to `/book/chapter/section/(table)+/figure`. This offers no advantage to computing the DFA lazily, and should be treated orthogonally. Pruning may increase the number of states in the DFA by up to a factor of D: for example, the lazy (and eager) DFA for `//*` has only one state, but if we first prune it with respect to a graph schema with D nodes, the DFA has D states.

Size of NFA tables A major component of the space used by the lazy DFA are the sets of NFA states that need to be kept at each DFA state. We call these sets *NFA tables*. The following proposition is straightforward, and ensures that the NFA tables do not increase exponentially:

Proposition 2. *Let Q be a set of p XPath expressions, of maximum depths n. Then the size of each NFA table in the DFA for Q is at most $n \times p$.*

Despite the apparent positive result, the sets of NFA states are responsible for most of the space in the lazy DFA, and we discuss them in Sec. 6.

4.3 Validation of the Size of the Very Lazy DFA

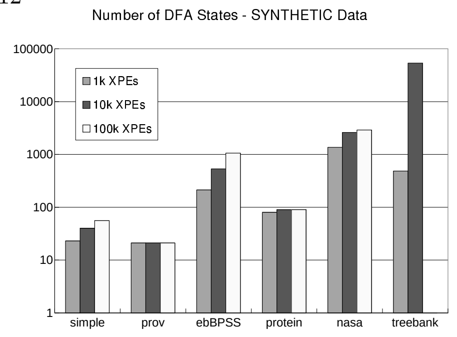
We ran experiments measuring the size of the lazy DFA for XML data for several publicly available DTDs, and one synthetic DTD. We generated synthetic data for these DTDs[13]. For three of the DTDs we also had access to real XML instances. The DTDs and the available XML instances are summarized in Fig. 8: four DTDs are simple, two are not; `protein.dtd` is non-recursive. We generated three sets of queries of depth $n = 20$, with 1,000, 10,000, and 100,000 XPath expressions[14], with `5%` probabilities for both the $*$ and the $//$.

Fig. 9(a) shows the number of states in the lazy DFA for the *synthetic* data. The first four DTDs are simple, or non-recursive, hence Theorem 3 applies. They had significantly less states than the upper bound in the theorem; e.g. `ebBPSS.dtd` has 1058 states, while the upper bound is 12,790 ($D = 29$, $d = 2$, $n = 20$). The last two DTDs were not simple, and neither Theorem 3 nor Corollary 1 applies (since synthetic data has large data guides). In one case (Treebank, 100,000 expressions) we ran out of memory.

Fig. 9(b) shows the number of states in the lazy DFA for *real* data. This is much lower than for synthetic data, because real data has small dataguides, and Corollary 1 applies; by contrast, the dataguide for synthetic data may be as large as the data itself. The `nasa.dtd` had a dataguide with 95 nodes, less than the number of tags in the DTD (117) because not all the tags occurred in the data. As a consequence, the lazy DFA had at most 95 states. Treebank has a data guide with 340,000 nodes; the largest lazy DFA here had only 44,000 states.

[13] Using `http://www.alphaworks.ibm.com/tech/xmlgenerator`.

[14] We used the generator described in [9].



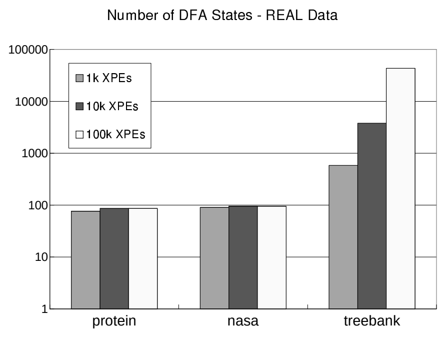


Fig. 9. Size of the lazy DFA for (left) synthetic data, and (right) real data. 1k means 1000 XPath expressions. For 100k XPath expressions for the `treebank` DTD with synthetic data we ran out of memory.

We also measured experimentally the average size of the NFA tables in each DFA state and found it to be around $p/10$, where p is the number of XPath expressions (graph shown in the appendix). Proposition 2 also gives an upper bound $O(p)$, but the constant measured in the experiments is much lower than that in the Theorem. These tables use most of the memory space and we address them in Sec. 6. Finally, we measured the average size of the transition tables per DFA state, and found it to be small (less than 40).

4.4 Constant Values

Finally, we comment on the impact of constant values on the number of states in the DFA. Each linear XPath expression can now end in a `text(S)` predicate, see Eq.(1). For a given set of XPath expressions, Q, let Σ denote the set of all symbols in Q, including those of the form `text(S)`. Let $\Sigma = \Sigma_t \cup \Sigma_s$, where Σ_t contains all element and attribute labels and ω, while Σ_s contains all symbols of the form `text(S)`. The NFA for Q has a special, 2-tier structure: first an NFA over Σ_t, followed by some Σ_s-transitions into sink states, i.e. with no outgoing transitions. The corresponding DFA also has a two-tier structure: first the DFA for the Σ_t part, denote it A^t, followed by Σ_s transitions into sink states. All our previous upper bounds on the size of the lazy DFA apply to A^t. We now have to count the additional sink states reached by `text(S)` transitions. For that, let $\Sigma_s = \{\texttt{text(S}_\texttt{1}\texttt{)}, \ldots, \texttt{text(S}_\texttt{q}\texttt{)}\}$, and let Q_i, $i = 1, \ldots, q$, be the set of XPath expressions in Q that end in $\texttt{text(S}_\texttt{i}\texttt{)}$; we assume w.l.o.g. that every XPath expression in Q ends in some predicate in Σ_s, hence $Q = Q_1 \cup \ldots \cup Q_q$. Denote A_i the DFA for Q_i, and A_i^t its Σ_t-part. Let s_i be the number of states in A_i^t, $i = 1, \ldots, q$. All the previous upper bounds, in Theorem 1, Theorem 3, and Corollary 1 apply to each s_i. We prove the following in the appendix.

Theorem 4. *Given a set of XPath expressions Q, containing q distinct constant values of the form `text(S)`, the additional number of sink states in the lazy DFA due to the constant values is at most $\sum_{i=1,q} s_i$.*

5 Experiments

This section validates the throughput achieved by lazy DFAs in stream XML processing. Our execution environment consists of a dual 750MHz SPARC V9 with 2048MB memory, running SunOS 5.8. Our compiler is gcc version 2.95.2, without any optimization options.

We used the NASA XML dataset [19] and concatenated all the XML documents into one single file, which is about 25MB. We generated sets of 1k (= 1000), 10k, 100k, and 1000k XPath expression using the XPath generator from [9], and varied the probability of $*$ and // to 0.1%, 1%, 10%, and 50% respectively. We report the throughput as a function of each parameter, while keeping the other two constant. For calibration and comparison we also report the throughput for parsing the XML stream, and the throughput of XFilter [3], which we re-implemented, without list balancing.

Figure 10 shows our results. In (a) we show the throughput as a function of the number of XPath expressions. The most important observation is that in the stable state (after processing the first 5-10MB of data) the throughput was constant, about 5.4MB/s. Notice that this is about half the parser's throughput, which was about 10MB/s; of course, the XML stream needs to be parsed, hence 10MB/s should be seen as an upper bound on our platform. We observed in several other experiments with other datasets (not shown here) that the throughput is constant, i.e. independent on the number of XPath expressions. By contrast, the throughput of XFilter decreased linearly with the number of XPath expressions. The lazy DFA is about 50 times faster than XFilter on the smallest dataset, and about 10,000 times faster than XFilter on the largest dataset. Figure 10 (b) and (c) show the throughput as a function of the probability of $*$, and of the probability of // respectively.

The first 5MB-10MB of data in Fig. 10 represent the *warm-up phase*, when most of the states in the lazy DFA are constructed. The length of the warm-up phase depends on the size of the lazy DFA that is eventually generated. For the data in our experiments, the lazy DFA had the same number of states for 1k, 10k, 100k, and 1000k (91, 95, 95, and 95 respectively). However, the size of the NFA tables grows linearly with the number of XPath expressions, which explains the longer tail: even if few states remain to be constructed, they slow down processing. In our throughput experiments with other datasets we observed that the lengths of the warm-up phase is correlated to the number of states in the lazy DFA.

6 Implementation Issues

Implementing the NFA tables In the lazy DFA we need to keep the set of NFA states at each DFA state: we call this set an *NFA table*. As shown in Prop. 2 the size of an NFA table is linear in the number of XPath expressions p, and about $p/10$ in our experiments. Constructing and manipulating these tables during the warm-up phase is a significant overhead, both in space and in time.

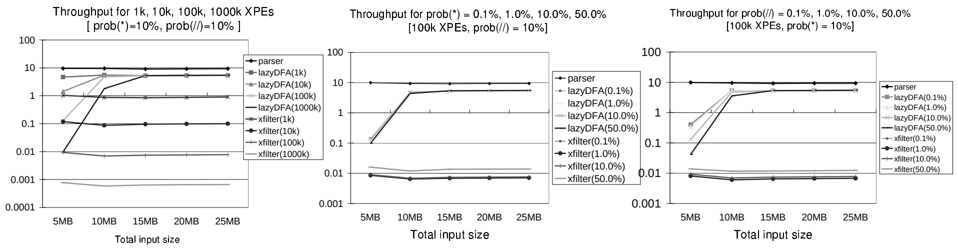


Fig. 10. Experiments illustrating the throughput of the DFA v.s. XFilter [3], as a function of the amount of XML data consumed. (left) varying number of XPath expressions (1k = 1000). (middle) varying probability of $*$. (right) varying probability of $//$.

We considered many alternative implementations for the NFA tables. There are three operations done on these sets: create, insert, and compare. For example a complex data set might have 10,000 DFA states, each containing a table of 30,000 NFA states and 50 transitions. Then, during warm-up phase we need to *create* $50 \times 10,000 = 500,000$ new sets; *insert* $30,000$ NFA states in each set; and *compare*, on average, $500,000 \times 10,000/2$ pairs of sets, of which only 490,000 comparisons return `true`, the others return `false`. We found that implementing sets as *sorted arrays* of pointers offered the best overall performance. An insertion takes $O(1)$ time, because we insert at the end, and sort the array when we finish all insertions. We compute a hash value (signature) for each array, thus comparisons with negative answers take $O(1)$ in virtually all cases.

Optimizing space To save space, it is possible to delete some of the sets of NFA tables, and recompute them if needed: this may slow down the warm-up phase, but will not affect the stable state. It suffices to maintain in each DFA state a pointer to its predecessor state (from which it was generated). When the NFA table is needed, but has been deleted (a *miss*), we re-compute it from the predecessor's set; if that is not available, then we go to *its* predecessor, eventually reaching the initial DFA state for which we always keep the NFA table.

Updates Both *online* and *offline* updates to the set of XPath expressions are possible. In the *online* update, when a new XPath expression is inserted we construct its NFA, then create a new lazy DFA for the union of this NFA and the old lazy DFA. The new lazy DFA is very efficient to build (i.e. its warm-up is fast) because it only combines two automata, of which one is deterministic and the other is very small. When another XPath expression is inserted, then we create a new lazy DFA. This results in a hierarchy of lazy DFAs, each constructed from one NFA and another lazy DFA. A state expansion at the top of the hierarchy may cascade a sequence of expansions throughout the hierarchy. Online deletions are implemented as invalidations: reclaiming the memory used by the deleted XPath expressions requires garbage-collection or reference count. *Offline* updates can be done by a separate (offline) system, different from the production system. Copy the current lazy DFA, A_l, on the offline system, and also copy there the new query tree, P, reflecting all updates (insertions, deletions, etc). Then construct the eager DFA, A_d, for P, but only expand states that have a corresponding state in A_l, by maintaining a one-to-one correspondence from A_d to A_l and only expanding a state when this correspondence can be extended to the new state. When completed, A_d is moved to the online system and processing resumes normally. The idea is that A_d will be no larger than A_l and, if there are only few updates, then A_d will be approximately the same as A_l, meaning that the warm-up cost for A_d is minimal.

7 Related Work

Two techniques for processing XPath expressions have been proposed. XFilter [3], its successor YFilter [9] and XTrie [6] evaluate large numbers of XPath expressions with what is essentially a highly optimized NFA. There is a space guarantee which is proportional to the total size of all XPath expressions. An optimization in XFilter, called list balancing can improve the throughput by factors of 2 to 4. XTrie identifies common strings in the XPath expressions and organizes them in a Trie. At run-time an additional data structure is maintained in order to keep track of the interaction between the substrings. The throughput in XTrie is about 2 to 4 times higher than that in XFilter with list balancing.

In [20] the authors describe a technique for event detection. Events are sets of atomic events, and they trigger queries defined by other sets of events. The technique here is also a variation on the Trie data structure. This is an efficient event detection method that can be combined with lazy DFAs in order to process XPath expressions with filters.

Reference [15] describes a general-purpose XML query processor that, at the lowest level, uses an event based processing model, and show how such a model can be integrated with a highly optimized XML query processor. We were influenced by [15] in designing our stream processing model. Query processors like [15] can benefit from an efficient low-level stream processor. Specializing regular expressions w.r.t. schemes is described in [10, 18].

8 Conclusion

The challenge in fast XML stream processing with DFAs is that memory requirements have exponential bounds in the worst case. We proved useful theoretical bounds and validated them experimentally, showing that memory usage is manageable for *lazy* DFAs. We also validated lazy DFAs on stream XML data and found that they outperform previous techniques by factors of up to 10,000.

Acknowledgment We thank Peter Buneman, AnHai Doan, Ashish Gupta, Zack Ives, and Arnaud Sahuguet for their comments on earlier versions of this paper. Suciu was partially supported by the NSF CAREER Grant 0092955, a gift from Microsoft, and an Alfred P. Sloan Research Fellowship.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.
2. A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.
3. M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination. In *Proceedings of VLDB*, pages 53–64, Cairo, Egipt, September 2000.
4. 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 *Proceedings of PLANX*, October 2002.
5. P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.
6. C. 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*, 2002.
7. J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *Proceedings of the ACM/SIGMOD Conference on Management of Data*, pages 379–390, 2000.
8. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.
9. Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *Proceedings of the International Conference on Data Engineering*, San Jose, California, February 2002.
10. M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the International Conference on Data Engineering*, pages 14–23, 1998.
11. R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of Very Large Data Bases*, pages 436–445, September 1997.
12. T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata. Technical Report 02-10-03, University of Washington, 2002. Available from `www.cs.washington.edu/homes/suciu`.
13. D. G. Higgins, R. Fuchs, P. J. Stoehr, and G. N. Cameron. The EMBL data library. *Nucleic Acids Research*, 20:2071–2074, 1992.
14. J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.
15. Z. Ives, A. Halevy, and D. Weld. An XML query engine for network-bound data. Unpublished, 2001.
16. H. Liefke and D. Suciu. XMill: an efficent compressor for XML data. In *Proceedings of SIGMOD*, pages 153–164, Dallas, TX, 2000.
17. M. Marcus, B. Santorini, and M.A.Marcinkiewicz. Building a large annotated corpus of English: the Penn Treenbak. *Computational Linguistics*, 19, 1993.
18. J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, pages 315–326, Edinburgh, UK, September 1999.
19. NASA's astronomical data center. ADC XML resource page. http://xml.gsfc.nasa.gov/.

20. B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 437–448, Santa Barbara, 2001.
21. D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *Proc. the International Conference on Data Engineering*, 2003.
22. G. Rozenberg and A. Salomaa. *Handbook of Formal Languages.* Springer Verlag, 1997.
23. A. Sahuguet. Everything you ever wanted to know about dtds, but were afraid to ask. In D. Suciu and G. Vossen, editors, *Proceedings of WebDB*, pages 171–183. Sringer Verlag, 2000.
24. A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using XML. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.
25. J. Thierry-Mieg and R. Durbin. Syntactic Definitions for the ACEDB Data Base Manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge,CB2 2QH, UK, 1992.

A Appendix

A.1 Proof of Theorem 1

Proof. Let A be the NFA for an XPath expression $P = p_0//p_1//\ldots//p_k$ (see notations in Sec. 4.1). Its set of states, Q, has n elements, in one-to-one correspondence with the symbol occurrences in P, which gives a total order on these states. Q can be partitioned into $Q_0 \cup Q_1 \cup \ldots \cup Q_k$, which we call *blocks*, with the states in $Q_i = \{q_{i0}, q_{i1}, q_{i2}, \ldots, q_{in_i}\}$ corresponding to the symbols in p_i; we have $\sum_{i=0,k} n_i = n$. The transitions in A are: states q_{i0} have self-loops with wild-cards, for $i = 1, \ldots, k$; there are ε transitions from $q_{i-1n_{i-1}}$ to q_{i0}, $i = 1, \ldots, k$; and there are normal transitions (labeled with $\sigma \in \Sigma$ or with wild-cards) from $q_{i(j-1)}$ to q_{ij}. Each state S in the DFA A_0 defined as $S = A(w)$ for some $w \in \Sigma^*$ ($S \subseteq Q$), and we denote $S_i = S \cap Q_i$, $i = 0, 1, \ldots, k$. Our goal is to count the number of such states.

Before proving the theorem formally, we illustrate the main ideas on $P = /a/b//a/*/a//b/a/*/c$, whose NFA, A, is shown in Fig.11. Some possible states are:

$$S_1 = A(a) = \{q_{01}\}$$
$$S_2 = \{q_{10}, q_{12}\} = A(a.b.e.e.a.b)$$
$$S_3 = \{q_{10}, q_{12}, q_{20}, q_{21}\} = A(a.b.e.e.a.b.a.e.e.a.b)$$
$$S_4 = \{q_{10}, q_{11}, q_{20}, q_{22}\} = A(a.b.e.e.a.b.a.e.e.b.a)$$

The example shows that each state S is determined by four factors: (1) whether it consists of states from block Q_0, like S_1, or from the other blocks, like S_2, S_3, S_4; there are n_0 states in the first category, so it remains to count the states in the second category only. (2) the highest block it reaches, which we call the *depth*: e.g. S_2 reaches block Q_1, S_3, S_4 reach block Q_2; there are k possible blocks. (3) the highest internal state in each block it reaches: e.g. for S_3 the highest internal state is q_{12}, in block Q_1, while for S_4 the highest internal states is q_{22}, in Q_2: there are at most $n - n_0$ choices here. (4) the particular values of the wildcards that allowed us to reach that highest internal state (not illustrated here); there are s^m such choices.

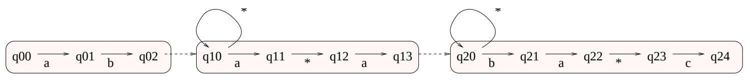


Fig. 11. The NFA for $/a/b//a/*/a//b/a/*/c$.

We now make these arguments formal.

Lemma 1. *Let $S = A(w)$ for some $w \in \Sigma^*$. If there exists some $q_{0j} \in S$, for $j = 0, \ldots, n_0 - 1$, then $S = \{q_{0j}\}$.*

Proof. There are no loops at, and no ε transitions to the states $q_{00}, q_{01}, \ldots, q_{0j}$, hence we have $\mid w \mid = j$. Since there are no ε transitions *from* these states, we have $S = A(w) = \{q_{0j}\}$

This enables us to separate the sets $S = A(w)$ into two categories: those that contain some q_{0j}, $j < n_0$, and those that don't. Notice that the state q_{0n_0} does not occur in any set of the first category, and occurs in exactly one set of the second category, namely $\{q_{0n_0}, q_{10}\}$, if $k > 0$ (because of the ε transition between them), and $\{q_{0n_0}\}$, if $k = 0$ respectively. There are exactly n_0 sets S in the first category. It remains to count the sets in the second category, and we will show that there are at most $k + k(n - n_0)s^m$ such sets, when $k > 0$, and exactly one when $k = 0$: then, the total is $n_0 + k + k(n - n_0)s^m \leq k + nks^m$, when $k > 0$, and is $n_0 + 1 = n + 1$ when $k = 0$. We will consider only sets S of the second kind from now on. When $k = 0$, then the only such set is $\{q_{0n_0}\}$, hence we will only consider the case $k > 0$ in the sequel.

Lemma 2. *Let $S = A(w)$. If $q_{de} \in S$ for some $d > 0$, then for every $i = 1, \ldots, d$ we have $q_{i0} \in S$.*

Proof. This follows from the fact the automaton A is linear, hence in order to reach the state q_{de} the computation for w must go through the state q_{i0}, and from the fact that q_{i0} has a self loop with a wild card.

It follows that for every set S there exists some d s.t. $q_{i0} \in S$ for every $1 \leq i \leq d$ and $q_{i0} \notin S$ for $i > d$. We call d the *depth* of S.

Lemma 3. *Let $S = A(w)$. If $q_{de} \in S$ for some $d > 0$, then for every $i = 1, \ldots, d-1$ and every $j \leq n_i$, if we split w into $w_1.w_2$ where the length of w_2 is j, then $q_{i0} \in A(w_1)$.*

Proof. If the computation for w reaches q_{de}, then it must go through $q_{i0}, q_{i1}, \ldots, q_{in_i}$. Hence, if we delete fewer than n_i symbols from the end of w and call the remaining sequence w_1 then the computation for w_1 will reach, and possible pass q_{i0}, hence $q_{i0} \in A(w_1)$ because of the selfloop at q_{i0}.

We can finally count the maximum number of states $S = A(w)$. We fix a w for each such S (choosing one nondeterministically) and further associate to S the following triple (d, q_{tr}, v): d is the depth; $q_{tr} \in S$ is the state with the largest r, i.e. $\forall . q_{ij} \in S \Rightarrow j \leq r$ (in case when there are several choices for t we pick the one with the largest t); and v is the sequence of the last r symbols in w. We claim that the triple (d, q_{tr}, v) uniquely determines S. First we show that this claim proves the theorem. Indeed there are k choices for d. For the others, we consider separate the choices $r = 0$ and $r > 0$. For $r = 0$ there is a single choice for q_{tr} and v: namely v is the empty sequence and $t = d$; for $r > 0$ there are $n_1 + n_2 + \ldots + n_d < n - n_0$ choices for q_{tr}, and at most s^m choices for v since these correspond to choosing symbols for the wild cards on the path from q_{t1} to q_{tr}. The total is $\leq k + k(n - n_0)s^m$, which, as we argued, suffices to prove the theorem. It only remains to show that the triple (d, q_{tr}, v) uniquely determines S. Consider two states, S, S', resulting in the same triples (d, q_{tr}, v). We have $S = A(w.v), S' = A(w'.v)$ for some sequences u, u'. It suffices to prove that $S \subseteq S'$ (the inclusion $S' \subseteq S$ is shown similarly). Let $q_{ij} \in S$. Clearly $i \leq d$ (by Lemma 1), and $j \leq r$. Decompose v into $v_1.v_2$, where v_2 contains the last $j-1$ symbols in v: this is possible since v's length is $r \geq j$. Since $q_{ij} \in A(w.v)$ the last j symbols in $w.v$ correspond to a transition from q_{i0} to q_{ij}. These last j symbols belong to v, because v's length is $r \geq j$, hence we can split v into $v_1.v_2$, with the length of v_2 equal to j, and there exists a path from q_{i0} to q_{ij} spelling out the word v_2. By Lemma 3 we have $q_{i0} \in A(w'.v_1)$ and, following the same path from q_{i0} to q_{ij} we conclude that $q_{ij} \in A(u'.v_1.v_2) = S'$.

A.2 Proof of Theorem 2

Proof. Given a set of XPath expressions Q, one can construct its DFA, A, as follows. First, for each $P \in Q$, construct the DFA A_P. Then, A is given by the product automaton $\prod_{P \in Q} A_Q$. From the proof of Theorem 1 we know that the states of A_P form two classes: a prefix, which is a linear chain of states, with exactly *prefix*(P) states, followed by a more complex structure with *suffix*(P) states. In the product automaton each state in the prefix of some A_P occurs exactly once: these account for $\sum_{P \in Q}$ *prefix*(P) states in A. The remaining states in A consists of sets of states, with at most one state from each A_P: there are at most $\prod_{P \in Q}(1 + \textit{suffix}(P))$ such states.

A.3 Proof of Theorem 3

Proof. Given an unfolded graph schema S, the set of all root-to-leaf sequences allowed by S, $\mathcal{L}_{schema} \subseteq \Sigma^*$, can be expressed as:

$$\mathcal{L}_{schema} = \{\varepsilon\} \cup \bigcup_{x \in \text{nodes}} \mathcal{L}_{schema}(x)$$

where $\mathcal{L}_{schema}(x)$ denotes all sequences of tags up to the node x in S. Our goal is to compute the number of states, as given by Eq.(3), with $\mathcal{L}_{data}$ replaced by $\mathcal{L}_{schema}$. Since the graph schema is simple and each simple path intersects at most d cycles, we have:

$$\begin{aligned}\mathcal{L}_{schema}(x) = \{&w_0.z_1^{m_1}.w_1 \ldots z_d^{m_d}.w_d \mid \\ &m_1 \geq 1, \ldots, m_d \geq 1\} \qquad (4)\end{aligned}$$

where $w_0, \ldots, w_d \in \Sigma^*$ and $z_1, \ldots, z_d \in \Sigma^+$. (To simplify the notation we assumed that the path to x intersects exactly d cycles.) We use a pumping lemma to argue that, if we increase some m_i beyond n (the depth of the query set), then no new states are generated by Eq.(3). Let $u.z^m.v \in \mathcal{L}_{schema}(x)$ s.t. $m > n$. We will show that $A_n(u.z^m.v) = A_n(u.z^n.v)$. Assume $q \in A_n(u.z^n.v)$. Following the transitions in A_n determined by the sequence $u.z^n.v$ we notice that at least one z in z^n must follow a selfloop (since n is the depth), corresponding to a // in one of the XPath expressions. It follows that $u.a^m.v$ has the same computation in A_n: just follow that loop an additional number of times, hence $q \in A_n(u.z^m.v)$. Conversely, let $q \in A_n(u.z^m.v)$ and consider the transitions in A_n determined by the sequence $u.z^m.v$. Let q' and q'' be the beginning and end states of the z^m segment. Since all transitions along the path from q' to q'' are either z or wild cards, it follows that the distance form q' to q'' is at most n; moreover, there is at least one self-loop along this path. Hence, z^n also determines a transition from q' to q''.

As a consequence, there are at most $(1+n)^d$ sets in $\{A_n(w) \mid w \in \mathcal{L}_{schema}(x)\}$ (namely corresponding to all possible choices of $m_i = 0, 1, 2, \ldots, n$, for $i = 1, \ldots, d$ in Eq.(4)). It follows that there are at most $1 + D(1+n)^d$ states in A_l.

A.4 Proof of Theorem 4

Proof. Each state in A_i^t can have at most one transition labeled `text(Si)`: hence, the number of sink states in A_i is at most s_i. The automaton for $Q = Q_1 \cup \ldots \cup Q_q$ is can be described as the cartesian product automaton $A = A_1 \times \ldots \times A_q$, assuming each A_i has been extended with the global sink state $\emptyset$, as explained in Sec. 3.1. The Σ_s-sink states[15] in A will thus consists of the disjoint union of the Σ_s-sink states from each A_i, because the transitions leading to Σ_s-sink states in A_i and A_j are incompatible, when $i \neq j$. Hence, there are $\sum_i s_i$ sink states.

A.5 Additional Experimental Results

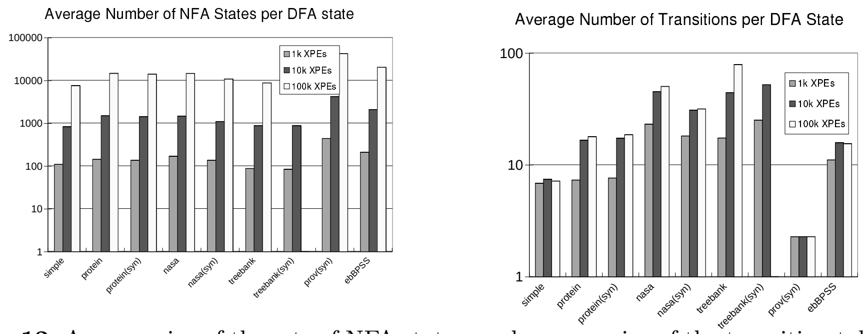


Fig. 12. Average size of the sets of NFA states, and average size of the transition table

Lazy DFA Size The two graphs in Figure 12 provide further experimental evidence on the size of the lazy DFA. Recall that each state in the lazy DFA is

[15] We call them that way in order not to confuse them with the $\emptyset$ sink state.

identified by a set of states from the underlying NFA. Storing this list of states for each state in the lazy DFA is a major contributor to memory use in the system. The first graph shows that across our collection of datasets, the number of NFA states in a lazy DFA state averaged about 10% of number of XPath expressions evaluated.

Another measure of the complexity of the lazy DFA is the number of transitions per lazy DFA state. The second graph reports this averaged quantity for each dataset.

Fallback Methods

Memory Guarantees: Combined Processing Methods The upper bounds on the size of the lazy DFA given so far apply in most practical cases, but not in all cases. When they don't apply we can combine lazy DFAs with an alternative evaluation method, which is slower but has hard upper bounds on the amount of space used. Two such methods are `xfilter` [3] and `xtrie` [6], both of which process the XML stream as we do, by interpreting SAX events. Either can be used in conjunction with a lazy DFA so as to satisfy the following properties. (1) The combined memory used by the lazy DFA+fall-back is that of the fall-back module plus a constant amount, M, determined by the user. (2) Most of the SAX events are processed only by the lazy DFA; at the limit, if the lazy DFA takes less than M space, then it processes *all* SAX events alone. (3) Each SAX event is processed at most once by the lazy DFA and at most once by the fall-back module (it may be processed by both); this translates into a worst case throughput that is slightly less than that of the fall-back module alone.

We describe such a combined evaluation method that is independent of any particular fall-back method. To synchronize the lazy DFA and the fall-back module, we use two stacks of SAX events, S_1 and S_2, and a pointer p to their longest common prefix. S_1 contains all the currently open tags: every `start(element)` is pushed into S_1, every `end(element)` is popped from S_1 (and the pointer p may be decreased too). S_2 is a certain snapshot of S_1 in the past, and is updated as explained below; initially S_2 is empty. Start processing the XML stream with the lazy DFA, until it consumes the entire amount of memory allowed, M. At this point continue to run the lazy DFA, but prohibit new states from being constructed. Whenever a transition is attempted into a non-existing state, switch to the fall-back module, as follows. Submit `end(element)` events to the fall-back module, for all tags from the top of S_2 to the common-prefix pointer, p, then submit `start(elements)` from p to the top of S_1. Continue normal operation with the fall-back module until the computation returns to the lazy DFA. At this point we copy S_1 into S_2 (we only need to copy from p to the top of S_1), reset p to the top of both S_1 and S_2, and continue operation in the lazy DFA (a depth counter will detect that). The global effect is that the lazy DFA processes some top part of the XML tree, while the fall-back module processes some subtrees: the two stacks are used to help the fall-back move from one subtree to the next.

We experimented with the `treebank` dataset, for which the lazy DFA is potentaily large, and found a 10/90 rule: by expanding only 10% of the states of the lazy DFA one can process 90% of all SAX events. Thus, only 10% need to be handled by the fall-back module. This is shown in Figure 13, which illustrates the number of SAX events processed by fallback methods when the lazy DFA is limited to a fixed number of states. The fixed number of DFA states is expressed as a percentage of the total required states for processing the input data. For example, when 100% of the lazy DFA states are allowed, then no SAX events are processed by fallback methods. The point at 12%, for example, shows that with only 12% of the lazy DFA states most of the SAX events are handled by the DFA, and only a few (less than 10%) by the fall-back method.

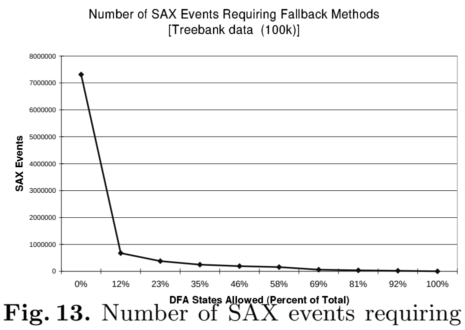


Fig. 13. Number of SAX events requiring fallback methods for lazy DFA limited to fixed number of states.