Paper Number: 323

Processing XML Streams with Deterministic Automata and Stream Indexes

Authors: Todd J. Green, Gerome Miklau, Makoto Onizuka, Dan Suciu (VLDB PC member)

Contact author:
Dan Suciu
suciu@cs.washington.edu
Department of Computer Science and Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350
206-685-1934

Track: Infrastructure for Information Systems — Research

Primary Topic: Database and database services in new context – Internet and WWW

Secondary Topic: Novel/Advanced Applications

Processing XML Streams with Deterministic Automata and Stream Indexes

Todd J. Green* Gerome Miklau† Makoto Onizuka‡ Dan Suciu†

Abstract

We consider the problem of evaluating a large number of XPath expressions on an XML stream. We make two key contributions: we convert the entire set of XPath expressions into a single Deterministic Finite Automata (DFA), and we use a Stream IndeX (SIX). The size of the DFA grows exponentially in the number of expressions, so it was previously believed that they cannot be used for such large sets. Our first contribution consists of a collection of techniques, theoretical results, and experimental validations that show DFAs can be used successfully with large sets of XPath expressions. Experiments show a constant throughput of about 5.4MB/s for up to 1,000,000 XPath expressions. The second contribution consists of a novel technique to index streaming data, that can be used in network-bound XML streams. Our results show that it can lead to dramatic performance improvements: with a SIX of only 6% the size of the data, we report four-fold performance improvements.

1 Introduction

Several applications of XML stream processing have emerged recently: content-based XML routing [19], selective dissemination of information (SDI) [2, 5], continuous queries [6], and processing of scientific data stored in large XML files [10, 20, 16]. They commonly need to process large numbers of XPath expressions (say 10,000 to 1,000,000), on continuous XML streams, at network speed.

As a motivating example, consider XML Routing [19]. 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 [19] reports very poor performance with publicly-available tools.

Our goal is to develop techniques for evaluating large numbers of XPath expressions on XML streams. We *guarantee* a sustained throughput that is independent of the number of XPath expressions: for up to 1,000,000 XPath expressions we measured a constant throughput of about 5.4MB/s. (Note that our reference parser measures peak throughput of 9.6MB/s) In contrast, previous techniques have the disadvantage that the throughput decreases as the number of XPath expressions increases. [2, 5].

We make two core contributions to XML stream processing: lazy Deterministic Finite Automata (DFA) and the Stream IndeX (SIX). To process at guaranteed throughput we convert all XPath expressions into a single DFA. This was thought impossible before because the number of XPath expressions is very large and the size of the DFA grows exponentially in this number. In fact, previous work in this area [2, 5] explicitly avoided using DFAs, and developed alternative processing techniques that are slower, but have guaranteed space bounds. Our *first* contribution consists of a collection of techniques, theoretical results, and experiments that validate DFAs for use in stream XML processing. The techniques use *lazy DFAs*, i.e. constructing the DFA states on an as-needed basis. We provide a complete theoretical analysis of the size of the eager and lazy DFA. In particular we prove that the number

* Xyleme (work done at U. Washington)
† University of Washington
‡ NTT Cyber Space Laboratories, NTT Corporation (work done while visiting University of Washington)

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

of states in the lazy DFA has a small upper bound that is independent of the number of XPath expressions. This is a surprising result because the number of states in the eager DFA is exponential in that of XPath expressions. We validate this result experimentally, showing that the space used by the lazy DFA is manageable in practice. We also conduct thorough performance experiments, confirming that an XML stream can be processed at constant throughput independent of the number of XPath expressions.

Our *second* contribution consists of a novel technique to index streaming data, called a Stream IndeX (SIX). A stream index is computed only once, by the data producer, and is sent along with the data stream. Every application that processes the data stream may use the index to improve its performance. A stream index differs significantly from traditional index structures: it needs to be much smaller than the data stream in order not to use extra network bandwidth, and it must arrive *just in time* to be of any use to the application. In our experiments the SIX is only 6% the size of the XML stream (and can be further reduced), and increased the throughput by up to a factor of four (to 27MB/s). To our knowledge the SIX is the first attempt to index streaming data, and our results prove that such an index can lead to dramatic performance improvements.

The techniques described in this paper have been incorporated into a public software package[1].

Paper Organization We begin with an overview of the system in Sec. 2. We describe in detail processing with a DFA in Sec. 3, then discuss its construction in Sec. 4 and analyze their size both theoretically and experimentally. We describe the stream index in Sec. 5. Experiments are discussed in Sec. 6. We discuss other issues in Sec. 7, and related work in Sec 8. Finally, we conclude in Sec. 9.

2 Overview

2.1 The Event-Based Processing Model

We define a simple event-based XML processing model in which a query is modeled as a tree, called the *query tree*, consisting of several XPath expressions. 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 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, computing some values, etc. Optionally, a binary SIX stream may accompany the XML stream to speed up the stream processing.

[1] Available at `www.cs.washington.edu/homes/suciu/XMLTK`.

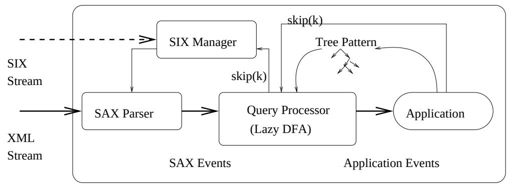


Figure 1: System's Architecture

An application starts by defining a query tree. This is a tree in which the nodes are labeled with variables and the edges with linear XPath expressions, P, given by the following grammar:

$$\begin{array}{lll} P & ::= & /N \mid //N \mid PP \\ N & ::= & E \mid A \mid text(S) \mid * \end{array} \qquad (1)$$

Here E, A, S are an element constant, an attribute constant, and a string constant respectively, and $*$ is wild card. The function `text(S)` matches a text node whose value is the string `S`. While filters and order predicates are not explicitly allowed, we show below that they can be expressed. In addition to the query tree, the application specifies the variables for which it requests SAX events. There is a distinguished variable, $\$R$, that is always bound to the root.

Example 2.1 The following is a query tree[2]:

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

Fig. 2 shows this query tree graphically, where the SAX events are requested for `$T` and `$TH`. 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 two columns show the application events. Only some of the SAX events are forwarded to the application, namely exactly those that occur within a `$T` or `$TH` variable event.

Independent Expressions A common case of a query tree is a *set of XPath expressions*, of 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 e_i may start with // instead of /). These are typical of many applications (e.g. XML routing), and are convenient for analyzing performance as a function of p. We will interchangeably refer to a query tree as a set of XPath expressions in this paper.

Filters Query trees do not support filters directly but they can be expressed by linearizing them and

[2] The tags are from an XML astronomical database [16].

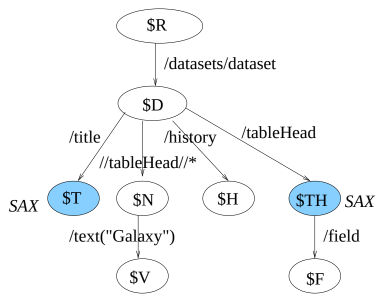


Figure 2: A Query Tree

XML Stream	Parser SAX Events	Application Events	
		Variable Events	SAX 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($T)`	
			`start(title)`
`<subtitle>`	`start(subtitle)`		`start(subtitle)`
`Study`	`text(Study)`		`text(Study)`
`</subtitle>`	`end(subtitle)`		`end(subtitle)`
`</title>`	`end(title)`		`end(title)`
		`end($T)`	
...			
`</dataset>`	`end(dataset)`	`end($D)`	
...	...		
`</datasets>`	`end(datasets)`	`end($R)`	

Figure 3: Events generated by a Query Tree

adding extra logic at the application level. For example the XPath expression:

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

is linearized into the query tree:

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

with SAX events requested for `$U` and `$X`. The application is thus notified about all relevant events, but needs to implement extra logic to define the filters' semantics. Specifically it needs two boolean variables, `b1, b2`; on a `$Z` event, it sets `b1` to true; on a `$U` event it tests the subsequent value and, if it is > 200, then sets `b2` to true. The application needs to buffer all SAX events below `quantity` and, at the end of a `$Y` event check whether `b1`=`b2`=`true`. Buffering is thus left to the application, if needed. Other predicates such as document order can be handled similarly.

2.2 Summary of Performance Results

The techniques described here evaluate a large set of XPath expressions on an XML stream at guaranteed throughput. A brief overview of their performance is shown in Table 1. The first two lines are for reference, and show the throughput obtained by the `xerces` SAX parser (available from the Apache foundation [3]) and by another publicly available parser, `xmill` [13] (the latter is restricted to ASCII characters and is optimized for speed). We used the `xmill` parser in our system, hence the 9.6MB/s should be viewed as an upper bound.

The next entry shows that 10,000 XPath expressions can be processed at 5.4MB/s. Our system converts all expressions into a single Deterministic Finite Automaton (DFA), then evaluates the DFA on the XML stream. Notice that the throughput approaches the throughput of the parser. Significantly, the throughput is independent of the number of XPath expressions, as shown in the next two rows. This property distinguishes our technique from others proposed in the literature [2, 5], where the throughput decreases as the number of XPath expression increases (in absolute numbers our performance is also significantly better). To our knowledge, this is the first attempt to guarantee the throughput. The main challenge of utilizing a DFA, and the obstacle to its prior use, is that its size may grow exponentially in the number of XPath expressions. One of this paper's important contributions is to show how DFAs can be made to work..

Operation	Throughput
`xerces` parser	3.9 MB/s
`xmill` parser	9.6 MB/s
10,000 XPath exprs	5.4 MB/s
100,000 XPath exprs	5.4 MB/s
1,000,000 XPath exprs	5.4 MB/s
10,000 XPath exprs with SIX	27.0 MB/s

Table 1: Illustration of the main results in the paper.

The last row in the table shows the performance obtained with a Stream IndeX (SIX): now the throughput is 27MB/s, which is almost a factor of four improvement. Notice that the throughput is greater than the parser's: the SIX contains offsets in the XML stream that enable the system to skip portions, and avoid parsing them. The SIX needs to be computed only once, at the data producer, and can then be used by all consumers downstream. It only increases the stream by 6%, or less. As we said, the SIX is, to our knowledge, the first approach to "index" streaming data, and we find it to be highly effective.

3 Processing with DFAs

3.1 Background on DFA

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: first convert the query tree into a Nondeterministic Finite Automaton (NFA), then convert it into a DFA. We review here briefly the basic techniques for both steps and refer the reader to [11] for more details. Our running example will be the query tree P shown in Fig. 4(a). The NFA, denoted A_n, is illustrated in Fig. 4(b). Transitions labeled $*$ correspond to $*$ or // in P; there is one initial state; terminal states are labeled with variables (`$X`, `$Y`, ...); and there are ε-transitions [3]. It is straightforward to generalize this to any query tree. Importantly, 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 $A_n(\varepsilon) = \{1\}$, $A_n(a.b) = \{3, 4, 7\}$, $A_n(a.\omega) = \{3, 4\}$, $A_n(b) = \emptyset$.

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

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

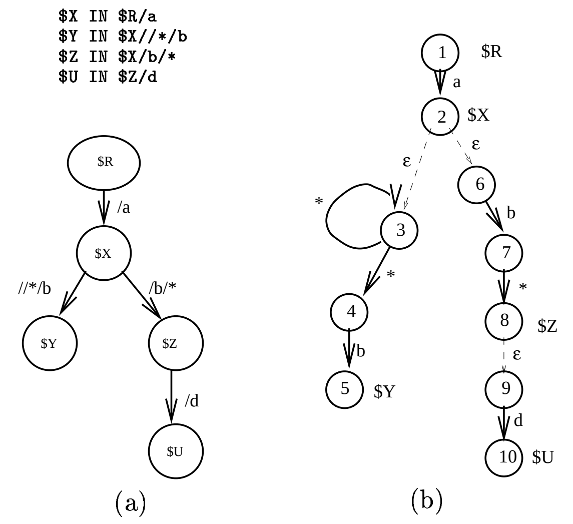
For our running example A_d is illustrated[4] in Fig. 5. Each state has unique transitions, and one optional `[other]` transition, denoting any symbol in Σ *except* the explicit transitions at that state: this is different from $*$ in A_n which denotes *any* symbol. For example `[other]` at state $\{3, 4, 8, 9\}$ denotes either a or ω. 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 configuration consisting of 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 it with the state reached by following the `element` transition[5]; on a `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[6]. Thus, each SAX event is processed in $O(1)$ time, and we can guarantee the throughput, independent of the number of XPath expressions. The trade-off is the size of the DFA, an issue that we address next.

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

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



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

Figure 4: (a) A query tree; (b) its NFA, A_n.

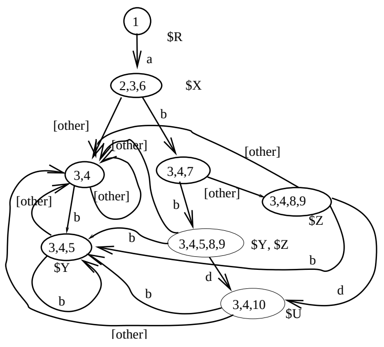


Figure 5: The DFA, A_d, for Fig. 4.

4 Analyzing the Size of the DFA

For a general regular expression the size of the DFA may be exponential [11]. In our setting, however, the expressions are restricted to XPath expressions as in Sec. 2.1. We analyze and discuss here their eager and lazy DFAs.

4.1 The Eager DFA

We start with the eager DFA.

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

[6]The stack is a static array; we assume a bound on the maximum depth of the XML stream.

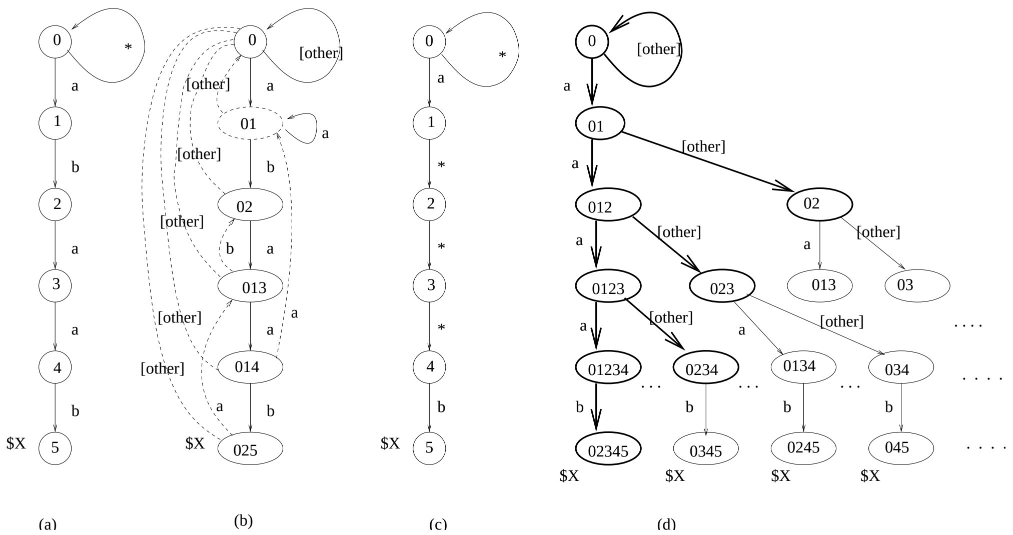


Figure 6: The NFA (a) and the DFA (b) for `//a/b/a/a/b`. The NFA (c) and the DFA (with back edges removed) (d) for `//a/*/*/*/b`: here the eager DFA has $2^5 = 32$ states, while the lazy DFA, assuming the DTD `<!ELEMENT a (a*|b)>`, has at most 9 states.

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:

Parameter	**Meaning**
k	number of $//$'s
s	alphabet size: $\mid \Sigma \mid$
n	length of P: $\sum n_i$
m	max # of $*$'s in each p_i

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). We prove:

Theorem 4.1 *The DFA for a linear XPath expression has at most $k + kns^m$ states, when $k > 0$, and $n + 1$ states, when $k = 0$.*

The details of the proof are deferred to the appendix. We first illustrate the theorem in the case when 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 that correspond to the *prefix function* in Knuth-Morris-Pratt's string matching algorithm [7].

When there are wild cards ($m > 0$), the theorem gives an exponential upper bound. This is unavoidable: Fig. 6 (c), (d) illustrate the NFA and DFA for $p = //a/*/*/*/b$, and the DFA 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[7], where m is the number of $*$'s.

Thus, the theorem shows that the wild cards (more precisely, the maximum number of wild cards between two consecutive occurrences of $//$) are the only source of exponential size for the eager DFA in the case of one linear XPath expression. One expects this number to be small in most practical applications; arguably users write expressions like `/catalog//product//color` rather than `/catalog//product/*/*/*/*/*/*/*/*/*/color`. Indeed, some implementations of XQuery already translate a *single* linear XPath expressions into DFAs [12].

Multiple XPath Expressions For multiple linear XPath expressions, the DFA grows exponentially in their number.

Example 4.2 Consider four XPath expressions:

```
$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. Generalizing, we get:

Proposition 4.3 *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.*[8]

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

[8]Although this requires p distinct tags, the result can be shown with only 2 distinct tags, using standard techniques.

Examples like 4.2 are common in the applications we consider, and rule out the eager DFA. The solution to this problem is to construct the DFA lazily, discussed below.

Size of NFA tables In addition to the number of DFA states, we also analyze the size of their internal structure: the tables of NFA states. Given a set of p XPath expressions, let k, n, s, m be the largest values of the parameters in Th. 4.1 for each expression. Then, we can prove using Th. 4.1:

Theorem 4.4 *The maximum size of an NFA table in each DFA state is at most $p(k + kns^m)$.*

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 need to be computed.

This idea has been used before in text processing, but it has never been applied to such large numbers 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 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):

$$states(A_l) \quad = \quad \{A(w) \mid w \in \mathcal{L}_{data}\} \qquad (3)$$

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, 4] to formalize our notion of schema, where nodes are labeled with tags and edges denote inclusion relationships. We call a graph schema *k-cyclic* if the length of the longest simple cycle is k. For example, the DTD:

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

is 1-cyclic, because the only cycles in its graph schema (shown in Fig. 7 (a)) are self-loops. Non-recursive DTDs are 0-cyclic. For 1-cyclic graph schema we denote d the maximum number of loops on any path, and D the number of nodes in its unfolding[9]. In our example $d = 2$ and $D = 13$: the unfolded graph schema is shown in Fig. 7 (b). For a query tree, denote n its depth, i.e. the maximum number of symbols on any path from root to a leaf (this generalizes the parameter n in Sec. 4.1). We prove the following result in the Appendix:

Theorem 4.5 *Consider a 1-cyclic graph schema with d, D, defined as above, and let P be a query tree of maximum depth n. Then the lazy DFA has at most $D \times n^d$ states.*

The result is surprising, because the number of states does not depend on the number of XPath expressions. In Example 4.2 the depth is $n = 2$: for the DTD above, the theorem guarantees at most $13 \times 2^2 = 52$ states in the lazy DFA, even if the number of Xpath expressions increases to 100,000. In practice, the depth is larger: for $n = 10$, the theorem guarantees ≤ 1300 states. By contrast, the eager DFA has $\geq 2^{100000}$ states. Fig. 6 (d) shows another example: of the 2^5 states in the eager DFA only 9 are expanded in the lazy DFA.

Theorem 4.5 has many applications. First for *non-recursive* DTDs ($d = 0$) the lazy DFA has at most D states[10]. Second, in *data-oriented* XML instances, recursion is often restricted to hierarchies, e.g. departments within departments, parts within parts. Hence, their DTD is 1-cyclic, 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.

The theorem does not apply, however, to *document-oriented* XML data. These have k-cyclic DTDs for $k > 1$: for example a `table` may occur within a `footnote`, and a `footnote` may occur within a `table`. For such cases we give an upper bound on the size of the lazy DFA in terms of *Data Guides* [9]. The data guide is a special case of a graph schema, with $d = 0$, hence Theorem 4.5 gives:

Corollary 4.6 *Let G be the number of nodes in the data guide of an XML stream. Then, for any query tree P, the lazy DFA for P on that XML stream has at most 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 don't

[9]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 [18]), hence D is only modestly larger than the number of tags in the DTD.

[10]This also follows directly from (3) since in this case $\mathcal{L}_{schema}$ is finite and has D elements.

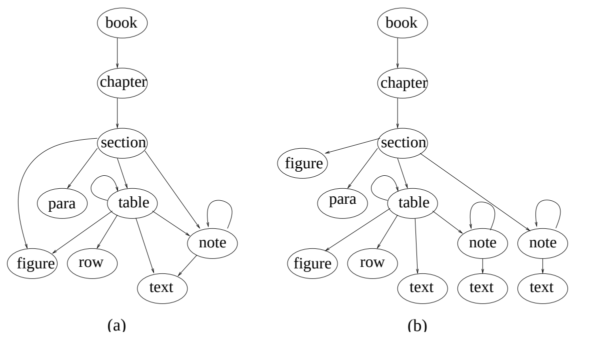


Figure 7: A graph schema for a DTD (a) and its unfolding (b).

DTD Source	DTD Names	(Element count in DTD, 1-cyclic, Data Size)		
		#	1-cyclic	MB
[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`	108	No	24
UPenn Treebank	`treebank.dtd`	249	No	56

Figure 8: Sources of data used in experiments.

nest such elements to arbitrary depth. All XML data instances described in [13] have very small data guides, except for Treebank [14], where the data guide has $G = 340,000$ nodes.

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: this is called *query pruning* in [8]. For example for the schema in Fig. 7 (a), the expression `//table//figure` is pruned to `/book/chapter/section/(table)+/figure`. This is no substitute for computing the DFA lazily, and should be treated orthogonally. We can prove that the eager DFA of the pruned XPath expressions has at least D times more states than the lazy DFA: 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.

4.3 Validation of the Size of the Lazy DFA

We validated experimentally the size of the lazy DFA for XML instances corresponding to about 20 publicly available DTDs, and one synthetic DTD. We generated synthetic data for these DTDs[11]. In three cases we also had access to large, real XML instances of a DTD. We generated three sets of queries of depth $n = 20$, with 1,000, 10,000, and 100,000 XPath expressions[12], with 5% probabilities for both the $*$ and the //. We omit here the small and the non-recursive DTDs, for which the lazy DFA was very small. The remaining DTDs are described in Fig. 8: three are 1-cyclic, two had a higher cyclicity parameter, k; `protein.dtd` is non-recursive, but we still report it because we had real data for it. We also had real data for `nasa.dtd` and `treebank.dtd`.

Fig. 4.3(a) shows the number of states in the lazy DFA for the *synthetic* data. The first four DTDs are 1-cyclic, or non-recursive, hence Theorem 4.5 applies. They had significantly less states than the bound in the theorem; e.g. `ebBPSS.dtd` has 1058 states, while the bound is 116,000 ($D = 29$, $d = 2$, $n = 20$). The last two DTDs were not 1-cyclic, and neither Theorem 4.5 nor Corollary 4.6 applies (since synthetic data has large data guides). In one case (Treebank, 100,000 expressions) we ran out of memory.

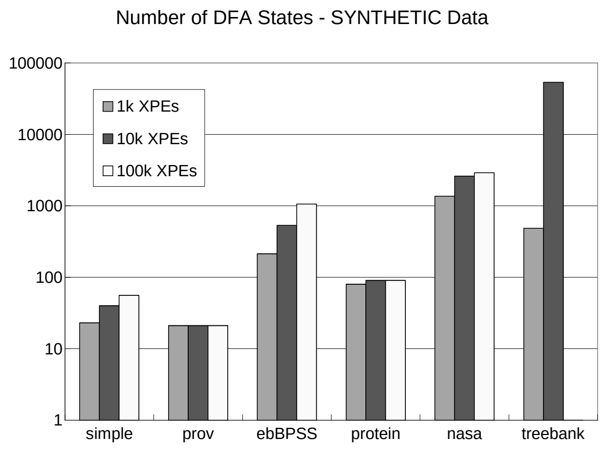
Fig. 4.3(b) shows the same two DTDs (plus `protein.dtd`), but with *real* data: it differs significantly from the synthetic data. Real data has small dataguides, and Corollary 4.6 applies: indeed all lazy DFAs had a small, or at least manageable number of states. Even here the theoretical upper bound was never reached: Treebank has a data guide with 340,000 nodes but the lazy DFA had at most 44,000. The reason why Fig. 4.3(b) differs so much from (a) is that real data has small data guides, while for synthetic data the size of the data guide is proportional to the size of the data.

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). This is much lower than the theoretical upper bound, Theorem 4.4. These tables use most of the memory space and we address them in Sec. 7. Finally, we measured the average size of the transition tables per DFA state, and found it to be small (less than 40).

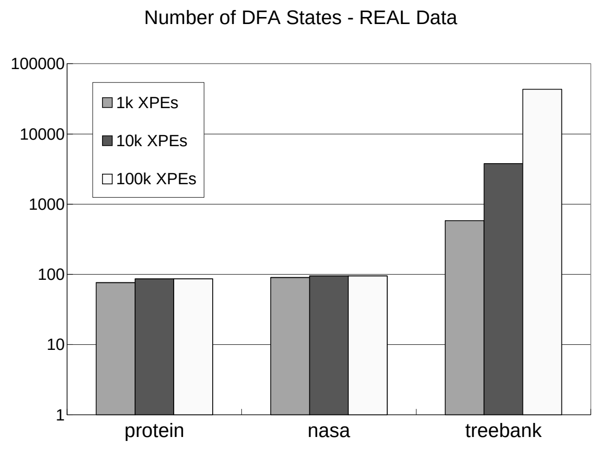
4.4 Memory Guarantees

Lazy DFAs offer guaranteed throughput, while Theorem 4.5 and Corollary 4.6 offer space guarantees under generous assumptions. When these assumptions don't hold (like for our synthetic data), we can use as fall back an alternative evaluation method that guarantees space but not throughput. Two such methods are `xfilter` [2] and `xtrie` [5], and both process the XML stream 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

[11]Using `http://www.alphaworks.ibm.com/tech/xmlgenerator`.
[12]We used the generator described in [2].



(a)



(b)

Figure 9: Size of the lazy DFA for (a) synthetic data, and (b) real data.

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. 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), and continue operation in the lazy DFA. The 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 various datasets (the graph for `treebank` is in the appendix) and found a 10%/90% rule: with only 10% of the states in the lazy DFA one can process 90% of all SAX events: only 10% need to be handled by the fall-back module.

5 The Stream IndeX (SIX)

An index for streaming data differs dramatically from one for stored data: it needs to be only a small fraction of the data (otherwise it consumes network bandwidth), and needs to arrive just in time for the application to use it. The *Stream IndeX* (SIX) defined here is, to our knowledge, the first attempt to index streaming data. A SIX consist of pairs of byte offsets:

$$(\texttt{beginOffset}, \texttt{endOffset})$$

where `beginOffset` is the byte offset of some begin tag, and `endOffset` of the corresponding end tag (relative to the begin tag). Both numbers are represented in binary, to keep the SIX small. The SIX is computed only once, by the producer of the XML stream, then send along with the XML stream and used by every consumer of that stream (e.g. by every router, in XML routing).

The SIX is sorted by `beginOffset`, allowing it to arrive synchronously with the XML stream. The system matches SIX entries with XML tags and, if it decides to skip the current element then it uses `endOffset` to skip characters in the XML stream without even parsing the content. This is a significant saving because parsing alone has limited throughput. The SIX module offers a single interface: `skip(k)`, where $\texttt{k} \geq 0$ denotes the number of open XML elements that need to be skipped. Thus `skip(0)` means "skip to the end of the most recently opened XML element". The example below illustrates the effect of a `skip(1)` call, issued after reading `<c>`:

```
XML stream:
<a> <b> <c> <d> </d> </c> <e> </e> </b> <f>  . . .
            |
         skip(1)
parser:
<a> <b> <c>                             <f>  . . .
```

It is easy to couple a SIX with a DFA. From the transition table of a DFA state we can see what transitions it expects. If a begin tag does not correspond to any transitions then we issue a `skip(0)`. As we show in Sec. 6 this results in dramatic speed-ups. Applications may have extra knowledge that allows them to make more aggressive skips, and issue `skip(k)` with $k > 0$.

The SIX is very robust: arbitrary entries may be removed without compromising consistency. Entries for very short elements are candidates for removal because they provide little benefit. Very large elements may also be removed, as we explain below. The SIX works with arbitrarily long (even infinite) XML streams. After exceeding 2^{32} bytes in the input stream, `beginOffset` wraps around; the only constraint is that each window of 2^{32}-bytes in the data has at least one entry in the SIX. `endOffset` does not wrap around: elements longer than 2^{32} bytes cannot be represented in the SIX and are removed.

The SIX can be easily constructed either by the application generating the XML stream or by some other application that parses the entire stream. We omit details for lack of space.

The effectiveness of the SIX depends on the selectivity. Given a query tree P and an XML stream let n be the total number of XML nodes, and let n_0 be the number of *selected* nodes, i.e. that match at least one variable in P. Define the *selectivity* as $\theta = n_0/n$. Examples: the selectivity of the XPath expression `//*` is 1; the selectivity of `/a/b/no-such-tag` is 0 (assuming `no-such-tag` does not occur in the data); referring to Fig.3, we have $n = 8$, $n_0 = 4$, hence $\theta = 0.5$. The maximum speed-up from a SIX is $1/\theta$. At one extreme, the expression `/no-such-tag` has $\theta = 0$, and may result in arbitrary large speed-ups, since every XML packet is skipped entirely. At the other extreme the SIX is ineffective when $\theta \approx 1$.

The presence of $*$'s and, especially, //'s may reduce the effectiveness of the SIX considerably, even when θ is small. For example the XPath expression `//no-such-tag` has $\theta = 0$, but the SIX is ineffective since the system needs to inspect every single tag while searching for `no-such-tag`. In order to increase the SIX' effectiveness, the $*$'s and //'s should be eliminated, or at least reduced in number, by specializing the XPath expressions w.r.t. the DTD (using *query pruning*, see Sec. 4.2). With this in mind, we used low probabilities for the $*$'s and //'s in the experiments in Sec. 6.2.

6 Experiments

Our experiments are meant to validate the following: (1) the throughput achieved by lazy DFA's in stream XML processing, and (2) the improvements obtained from the Stream IndeX.

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 an XML data generator from IBM[13] and the XPath expression generator from [2]. We executed the throughput experiments only once (since they had plenty of time to stabilize) and executed the SIX experiments three times, reporting the average. For comparison with XFilter, we re-implemented it following [2]; we did not implement list balancing.

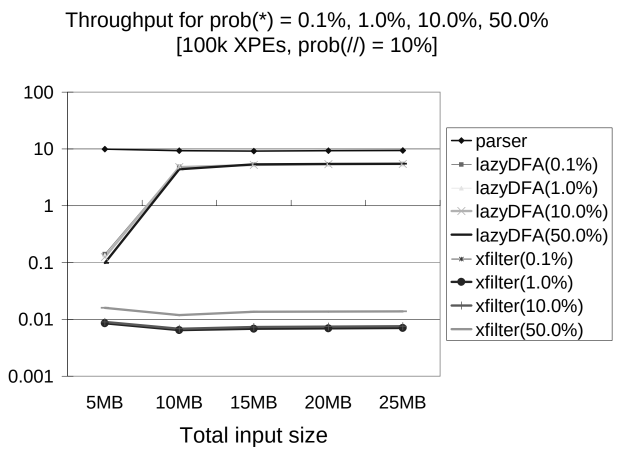
6.1 Throughput

We ran our experiments on the NASA XML dataset [16] and concatenated all the XML documents into one single file, which is about 25MB. The query sets had 1000, 10000, 100000, and 1000000 XPath expressions (denoted 1k, 10k, 100k, and 1000k), with the probability of $*$ and // equal to 0.1%, 1%, 10%, and 50% respectively: a total of 64 query sets. 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, described in [2].

Figure 10 shows 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 the same, about 5.4MB/s. We observed in several other experiments with other datasets (not shown here) that the throughput is constant, with the only measurable exception in the case of query trees with very low selectivity where the throughput was slightly higher, because we optimized the transitions in the "sink state" (Sec. 3.1). We found the constant throughput to be a very stable and predictable property. By contrast, the throughput of XFilter decreased linearly with the number of XPath expressions. Figure 11 shows the throughput as a function of the probability of $*$, and of the probability of // respectively. They show the same behavior for the lazy DFA. XFilter's performance does not depend on these parameters.

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 number of NFA tables grows linearly with the number of XPath expressions (Fig. 13), 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.

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



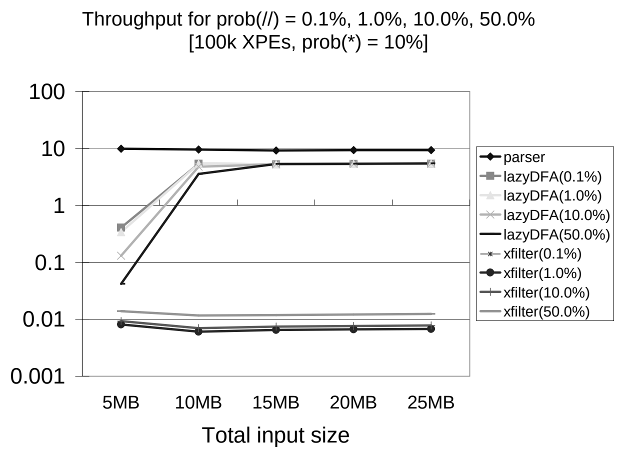


Figure 11: Experiments illustrating the throughput.

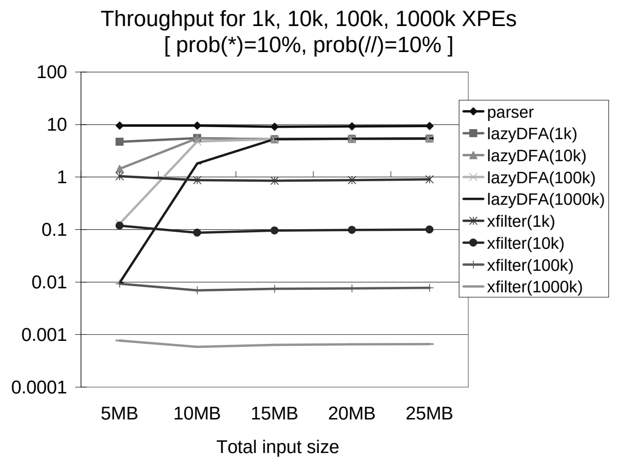


Figure 10: Experiments illustrating the throughput. 10k means 10,000, 100k means 100000 etc.

6.2 Stream IndeX

We evaluated the SIX on synthetic NITF data[14], with 10000 XPath expressions using 0.2% probabilities for both the // and the $*$'s (see Sec. 5 for a justification). In order to vary the selectivity parameter θ (Sec. 5), we made multiple copies of the NITF DTD, and randomly assigned each XPath expression to one such DTD: θ decreases when the number of copies increases. We generated about 50MB of XML data, then copied it to obtain a 100MB data set. The reason for the second copy is that we wanted to measure the SIX in the stable phase, while the lazy DFA warms up too slowly when using a SIX, because it sees only a small fragment of the data. The size of complete SIX for the entire dataset was 6.7MB, or about 6% of the XML data.

Fig. 6.2 (a) shows the throughput with a SIX, and without a SIX, for all three selectivities. Without a SIX the throughput was constant at around 7.3MB/s (slightly higher than for the previous experiments because of our optimization of the "sink state" transitions), while with a SIX the throughput increased significantly for low selectivities. For $\theta = 0.03$ the throughput was about 27MB/s, hence the speed-up is 3.7. Even slower selectivities increased the throughput even further, and are not shown here.

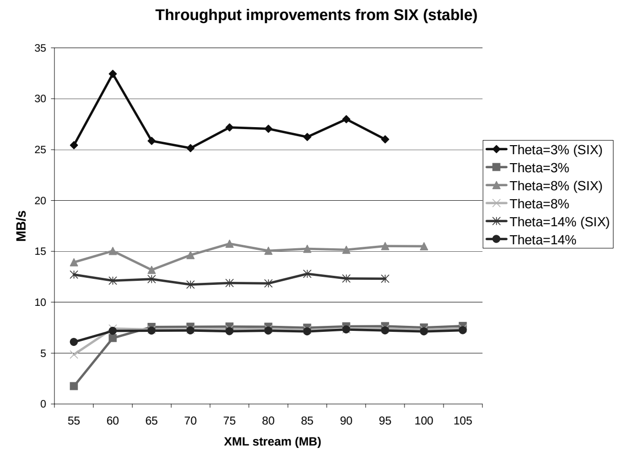
Next, we measured how much we can decrease the SIX by removing entries corresponding to small XML elements. Reducing the size is important for a stream index, since it competes for network bandwidth with the data stream. Fig. 6.2 (b) shows the throughput as a function of the cut-off size for the XML elements. The more elements are deleted from the SIX, the smaller the throughput. However, the SIX size also decreases, and does so much more dramatically. For example at the $2k$ data point, when we deleted from the SIX all elements whose size is $\leq 2k$ bytes, the throughput decreases to 18.6MB/s from a high of 27MB, but the size of the SIX decreases to a minuscule 522bytes, from a high of 6.7KB. Thus we can reduce the SIX more than ten times, but only pay a 28% penalty in the throughput.

7 Discussion

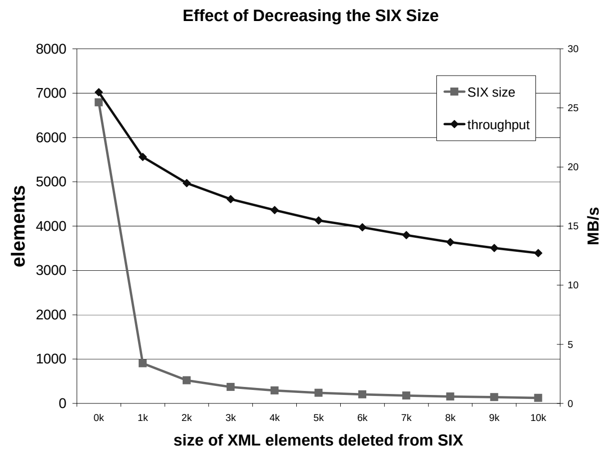
Recall from Sec. 4.3 that the NFA tables in the DFA states grow linearly with the number of XPath expressions. During the warm-up phase this affects both speed and space. We address these issues here, and also discuss updates of XPath expressions.

Optimizing speed during warm-up 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 set 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 *sorted arrays* of pointers offered the best overall performance. An insertion takes $O(1)$ time, because we only sort the array

[14] http://www.nitf.org/site/nitf-documentation/



(a)



(b)

Figure 12: Throughput improvement from the SIX (a), and the effect of decreasing the SIX size by deleting "small" XML elements (b).

when we finish all insertions. We maintain a checksum for each array, and use it as a pre-filter for the comparison: virtually all negative answers take $O(1)$ time. In addition, we build a hash table of all sets with the checksums as key.

Optimizing space during warm-up We consider releasing 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. We 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 released (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. We are currently implementing this method.

Updates We consider both *online* and *offline* to the set of XPath expressions, and start with online. 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, one of which is deterministic and the other 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 another lazy DFA and one NFA. A state expansion at the top of the hierarchy may cascade a sequence of expansions throughout the hierarchy. Online deletions are implemented by invalidation: reclaiming the memory used by the deleted XPath expressions requires *garbage-collection*. *Offline* updates can be done one a separate (offline) system, different from the production system. We copy the current lazy DFA, A_l, on the offline system, and also copy there the new query tree, P, with all updates incorporated. Then we 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. A_d will not be larger than A_l; also, if there are only few updates, then A_d will be approximately the same as A_l, meaning there is no warm-up cost for A_d on the online system.

8 Related Work

Two techniques for processing XPath expressions have been proposed recently. In XFilter [2] a large number of XPath expressions are interpreted with what is essentially a highly optimized NFA. There is a space guarantee which is proportional to the total size of all XPath expressions. However, there is no throughput guarantee, since the number of operations per input XML tag may, in the worst case, be as high as the number of XPath expressions.

In the recently proposed XTrie [5], the XPath expressions are preprocessed to find common substrings, which are organized in a Trie. At run-time an additional data structure is maintained in order to keep track of the interaction between the substrings. This technique works best when the XPath expressions are ordered, i.e. branches are matched in the XML document in the order listed in the query. Like XFilter, the XTrie also gives a guaranteed upper bound for the total space used, and are more efficient than XFilter.

In [17] 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 problem is complementary to ours: we are concerned here about how to generate the atomic events but leave to the application their interpretation.

Reference [12] 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 [12] in designing our stream processing model. Query processors like [12] can benefit from an efficient low-level stream processor. Specializing regular expressions w.r.t. schemas is described in [8, 15].

9 Conclusion

We have described two complementary techniques for evaluating large numbers of XPath expressions over XML streams. Using lazy DFAs we can process up to 1,000,000 XPath expressions with a guaranteed throughput approaching that of simple parsing. Adding our Stream IndeX we achieve four times higher throughput, or more than double of our reference parser.

The challenge in fast stream processing with DFAs is that memory requirements have exponential bounds in the worst case. We proved useful theoretical bounds under generous assumptions and validated them experimentally. When these assumptions don't hold, worst case exponential size increases are still possible (although we only witnessed such size increases for synthetically-generated data). For these cases, we have provided fall-back techniques that allow lazy DFAs to be used efficiently in all cases, although at some sacrifice of peak throughput.

The Stream Index is, to our knowledge, the first attempt to index streaming data as opposed to stored data, and we found it to be very effective. We also found it to be a robust, and flexible structure, offering a large spectrum of space/speed trade-off.

Acknowledgment We thank Peter Buneman, AnHai Doan, Zack Ives, and Arnaud Sahuguet for their comments.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[2] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination. In *Proceedings of VLDB*, pages 53–64, Cairo, Egipt, September 2000.

[3] Apache. Xerces C++ parser, 2001. http://xml.apache.org.

[4] 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.

[5] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressiosn. In *Proceedings of the International Conference on Data Engineering*, 2002.

[6] 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.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[8] 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.

[9] 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.

[10] D. G. Higgins, R. Fuchs, P. J. Stoehr, and G. N. Cameron. The EMBL data library. *Nucleic Acids Research*, 20:2071–2074, 1992.

[11] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[12] Z. Ives, A. Halevy, and D. Weld. An xml query engine for network-bound data. Unpublished, 2001.

[13] H. Liefke and D. Suciu. XMill: an efficent compressor for XML data. In *Proceedings of SIGMOD*, pages 153–164, Dallas, TX, 2000.

[14] M. Marcus, B. Santorini, and M.A.Marcinkiewicz. Building a large annotated corpus of English: the Penn Treenbak. *Computational Linguistics*, 19, 1993.

[15] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, pages 315–326, Edinburgh, UK, September 1999.

[16] NASA's astronomical data center. ADC XML resource page. http://xml.gsfc.nasa.gov/.

[17] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proceedingso of the ACM SIGMOD Conference on Management of Data*, pages 437–448, Santa Barbara, 2001.

[18] 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.

[19] A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using XML. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.

[20] 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 4.1

Proof: Let A be the NFA for p. 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 $Q_0 \cup Q_1 \cup \ldots \cup Q_k$, 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, 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 $\sigma \in \Sigma$) 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 $S_i = S \cap Q_i$, $i = 0, 1, \ldots, k$. Before counting the number of such states S we prove a few lemmas about the structure of the sets S.

Lemma A.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. As further clarification on these two categories, 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 A.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 A.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 A.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 A.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 4.5

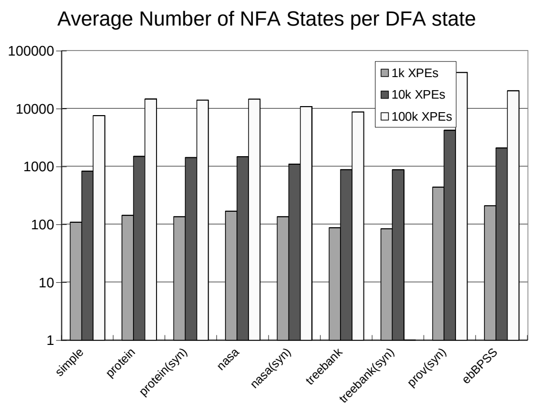
Proof: Given an unfolded graph schema and $\mathcal{L}_{dtd} \subseteq \Sigma^*$, we have:

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

where $\mathcal{L}_{schema}(x)$ denotes all sequences of tags up to the node x in the unfolded graph schema. Since the graph schema is 1-cyclic, and has at most d nested loops, we have:

$$\mathcal{L}_{schema}(x) = \{w_0.a_1^{m_1}.w_1 \ldots a_d^{m_d}.w_d \mid m_1 \geq 1, \ldots, m_d \geq 1\} \tag{4}$$

where $a_1, \ldots, a_d \in \Sigma$ and $w_0, \ldots, w_d \in \Sigma^*$. We use a pumping lemma to argue that, if we increase some m_i beyond n (the depth of the query tree), then no new states are generated by Eq.(3). Let $u.a^m.v \in \mathcal{L}_{schema}(x)$ s.t. $m > n$. We will show that $A_n(u.a^m.v) = A_n(u.a^n.v)$. Assume $q \in A_n(u.a^n.v)$. Following the transitions in A_n determined by the sequence $u.a^n.v$ we notice that at least one a in a^n must follow a selfloop (since n is the depth). It follows that $u.a^m.v$ has the same computation



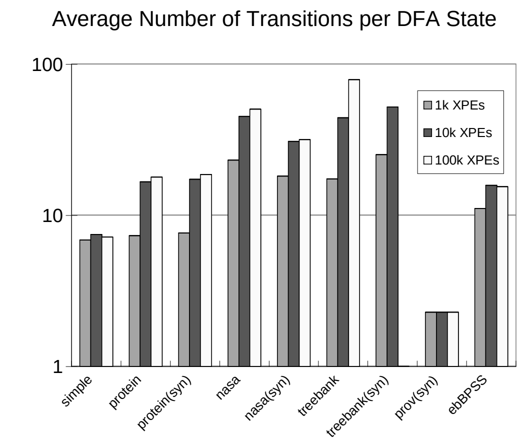


Figure 13: Average size of the sets of NFA states, and average size of the transition table

in A: just follow that loop an additional number of times, hence $q \in A_n(u.a^m.v)$. Conversely, let $q \in A_n(u.a^m.v)$ and consider the transitions in A determined by the sequence $u.a^m.v$. Let q' and q'' be the beginning and end states of the a^m segment. Since all transitions along the path from q' to q'' are either a 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, a^n also determines a transition from q' to q''.

As a consequence, there are at most n^d sets in $\{A_n(w) \mid w \in \mathcal{L}_{schema}(x)\}$ (namely corresponding to all possible choices of $m_i = 1, 2, \ldots, n$, for $i = 1, \ldots, d$ in Eq.(4)). It follows that there are at most Dn^d states in A_1. □

A.3 Additional Experimental Results

A.3.1 Lazy DFA Size

The two graphs in Figure 13 provide further experimental evidence on the size of the lazy DFA. Recall that each state in the lazy DFA is 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.

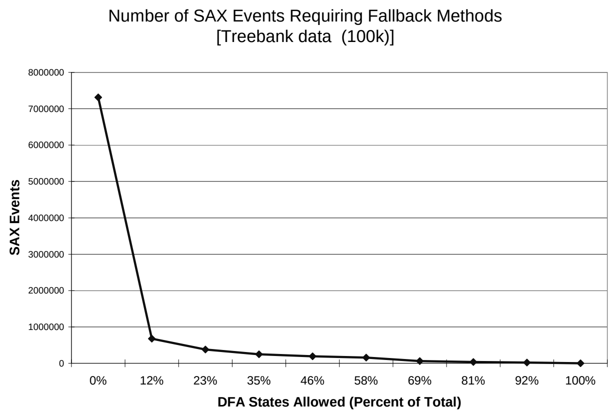


Figure 14: Number of SAX events requiring fallback methods for lazy DFA limited to fixed number of states.

A.3.2 Fallback Methods

Figure 14 shows 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.