Processing XML Streams With Deterministic Automata and Stream Indexes

TODD J. GREEN
University of Pennsylvania
ASHISH GUPTA
University of Washington
GEROME MIKLAU
University of Washington
MAKOTO ONIZUKA
NTT Cyber Space Laboratories, NTT Corporation
and
DAN SUCIU
University of Washington

We consider the problem of evaluating a large number of XPath expressions on a stream of XML packets. We contribute two novel techniques. The first is to use a single Deterministic Finite Automaton (DFA). The contribution here is to show that the DFA can be used effectively for this problem: in our experiments we achieve a constant throughput, independently of the number of XPath expressions. The major issue is the size of the DFA, which, in theory, can be exponential in the number of XPath expressions. We provide a series of theoretical results and experimental evaluations that show that the *lazy* DFA has a small number of states, for all practical purposes. These results are of general interest in XPath processing, beyond stream processing. The second technique is the Streaming IndeX (SIX), which consists of adding a small amount of binary data to each XML packet that allows the query processor to achieve significant speedups. As an application of these techniques we describe the XML Toolkit (XMLTK), a collection of command-line tools providing highly scalable XML data processing.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Classes defined by grammars or automata (e.g., context-free languages, regular sets, recursive sets)*

Dan Suciu was partially supported by the NSF CAREER Grant IIS-0092955, NSF Grants IIS-0140493 and IIS-0205635, a gift from Microsoft, and a Sloan Fellowship.

Authors' addresses: T. H. Green, Department of Computer and Information Science, Levine Hall, 3330 walnut Street, Philadelphia, PA 19104-6389; email tjgreen@cis.upenn.edu; A. Gupta, G. Miklau, and D. Suciu, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350; email:{akgupta,gerome,suciu}@cs.washington.edu; M. Onizuka, Open Source Software Computing Project, NTT Cyber Space Laboratories, NTT Corporation, 1-1 Hikarinooka, Yokosuka-shi, Kanagawa 239-0847, Japan, email: onizuka.makoto@lab.ntt.co.jp.

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: XML processing, stream processing

1. INTRODUCTION

Several applications of XML stream processing have emerged recently: content-based XML routing [Snoeren et al. 2001], selective dissemination of information (SDI) [Altinel and Franklin 2000; Chan et al. 2002; Diao et al. 2003], continuous queries [Chen et al. 2000], and processing of scientific data stored in large XML files [Higgins et al. 1992; Thierry-Mieg and Durbin 1992; Borne n.d.]. They commonly need to process a large collection of XPath expressions (say 10,000 to 1,000,000), on a continuous stream of XML data, at a high sustained throughput.

For illustration, consider XML Routing [Snoeren et al. 2001]. 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 Snoeren et al. [2001] reported very poor performance with publicly available XPath processing tools.

Our goal is to develop techniques for evaluating a large collection of XPath expressions on a stream of XML packets. First we describe a technique that *guarantees* a sustained throughput, which is largely independent of the number of XPath expressions. In contrast, in all other techniques proposed for processing XPath expressions the throughput decreases as the number of XPath expressions increases. [Altinel and Franklin 2000; Chan et al. 2002; Diao et al. 2003]. Second, we describe a lightweight binary data structure, called *Stream IndeX* (SIX), which can be added to the XML packets for further speedups.

The first and main contribution is to show that a Deterministic Finite Automaton (DFA) can be used effectively to process a large collection of XPath expressions, at guaranteed throughput. Our approach 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, but they were thought to be useless for workloads with a large number of XPath expressions, because their size grows exponentially with size of the workload.

Our solution to the state explosion problem consists of constructing the DFA lazily. A *lazy* DFA is one whose states and transitions are computed from the corresponding NFA at runtime, not at compile time. A new entry in the transition table or a new state is computed only when the input data requires the DFA to follow that transition or enter that state. The transitions and states in the lazy DFA form a subset of those in the standard DFA, which we call *eager* DFA in this article. As a consequence, the lazy DFA can sometimes be much smaller than the eager DFA.

We show that, for XML processing, the number of states in the lazy DFA is small and depends only on the structure of the XML data. It is largely independent on the number of XPath expressions in the workload. More precisely, the size of the lazy DFA is at most the size of the data guide [Goldman and Widom 1997] of the XML data, which is typically very small for XML data that has a fairly regular structure. In hindsight, after we first announced this result in Green et al. [2003], this fact may sound obvious, but it was far from obvious before. Previous work in this area [Altinel and Franklin 2000; Chan et al. 2002; Diao et al. 2003] explicitly avoided using DFAs, and developed alternative processing techniques that are slower, but have guaranteed space bounds.

To support the claim that the number of states in the lazy DFA is small, we present here a series of theoretical results characterizing the size of both the eager and the lazy DFA for XPath expressions. These results are of general interest in XPath processing, beyond stream applications.

The second contribution in this article consists of a light-weight technique for speeding up processing XML documents in a network application. The observation here is that, in many applications processing streams of XML messages, the main bottleneck consists of parsing, or tokenizing each message. To address that, some companies use a proprietary tokenized format instead of the XML text representation [Florescu et al. 2003], but this suffers from lack of interoperability. We propose a more lightweight technique that adds a small amount of binary data to each XML document, facilitating access into the document. We call this data a *Stream IndeX* (SIX). The SIX is computed once, when the XML document is first generated, and attached somehow to the document (for example using DIME [Corp. n.d.]). All applications receiving the document that understand the SIX can then access the XML data much faster. If they don't understand the SIX, then they can fall back on the traditional parse/evaluate model. Space-wise, the overhead of a SIX is very small (typical values are, say, 7% of the data, and can be reduced further), so there is little or no penalty from using it. We note that the general principle of adding a small amount of binary data to facilitate access in the XML document also admits other implementations, see [Gupta et al. 2002, 2003].

Finally, we illustrate an application of our techniques by describing the *XML Toolkit* (XMLTK), for highly scalable processing of XML files. Our goal is to provide to the public domain a collection of stand-alone XML tools, in analogy with Unix commands for text files. Current tools include sorting, aggregation, nesting, unnesting, and a converter from a directory hierarchy to an XML file. Each tool performs one single kind of transformation, but can scale to arbitrarily large XML documents in, essentially, linear time, and using only a moderate amount of main memory. By combining tools in complex pipelines users can perform complex computations on the XML files. There is a need for such tools in user communities that have traditionally processed data formatted in line-oriented text files, such as network traffic logs, Web server logs, telephone call records, and biological data. Today, many of these applications are done by combinations of Unix commands, such as `grep`, `sed`, `sort`, and `awk`. All these data formats can and should be translated into XML, but then all the line-oriented

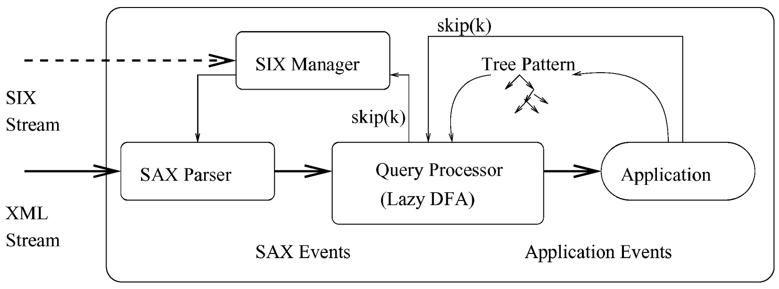


Fig. 1. System's architecture.

Unix commands become useless. Our goal is to provide tools that can process the data after it has been migrated to XML.

Discussion. This article focuses only on linear XPath expressions. Applications rarely have such simple workloads, and are more likely to use XPath expressions with nested predicates. Scalable techniques for such workloads require a separate investigation and are out of the scope of this article. However, the techniques described here are relevant to the general XPath processing problem, for two reasons. First, processing linear expressions is a subproblem in processing more complex workloads, and needs to be addressed somehow. In fact we describe here a simple way to evaluate XPath expressions with nested predicates by decomposing them into linear fragments, and we found this simple technique to work well on small workloads. Second, at a deeper level, it has been shown in Gupta and Suciu [2003] that our results about the DFA extend, although not in a trivial way, to a pushdown automaton, which can process an arbitrarily complex workload of XPath expressions with nested predicates. Thus, the results and techniques discussed in this article can be seen as building blocks for more powerful processors.

Paper organization. We begin with an overview in Section 2 of the processing model and the system's architecture. We describe in detail processing with a DFA in Section 3, then discuss its construction in Section 4 and analyze its size. We describe the SIX in Section 5. We report our experimental results in Section 6 and describe the XML Toolkit in Section 7. Section 8 contains related work, and we conclude in Section 9. The electronic appendix contains some of the proofs and more details on the XML Toolkit.

2. OVERVIEW

2.1 The Event-Based Processing Model

The architecture of our XML stream processing system is shown in Figure 1. The user specifies several correlated XPath expressions arranged in a tree, called the *query tree*. An input stream of XML packets is first parsed by a SAX parser that generates a stream of *SAX events*, or *SAX tokens*; this is sent to the query processor, which 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, or

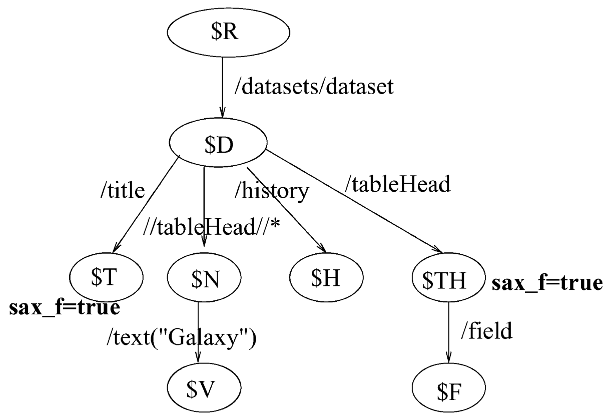


Fig. 2. A Query tree.

computing some values. An optional Stream Index (called *SIX*) may accompany the XML stream to speed up processing (Section 5).

We consider linear XPath expressions, P, given by the following grammar:

$$\begin{aligned} P &::= /N \mid //N \mid PP, \\ N &::= E \mid A \mid * \mid \mathit{text}() \mid \mathit{text}() = S. \end{aligned} \tag{1}$$

Here E and A are element label and attribute label respectively, / denotes the child axis, // denotes the descendant axis, $*$ is the wild card, and S is a string constant. As explained earlier, nested predicates are not discussed here, and have to be decomposed into linear XPath expressions, as shown below.

A query tree, Q, has nodes labeled with variables and the edges with linear path expressions. There is a distinguished variable, $\$R$, which is always bound to the root node of the XML packet. Each node in the tree also carries a Boolean flag, called `sax_f`. When its value is `true`, then the SAX events under that node are forwarded to the application; otherwise they are not forwarded to the application. The `sax_f` can be set on and off at various nodes in the query tree. The `sax_f` flag is used by the stream index, Section 5.

Example 2.1. The following is a query tree (tags taken from the NASA dataset [Borne n.d.]):

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

Figure 2 shows this query tree graphically. Here the application requests the SAX events under `$T`, and `$TH` only. Figure 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

XML Stream	Parser Events: SAX Events	Application Events: SAX and variable events
<datasets>	startElement(datasets)	startVariable($R)
<dataset>	startElement(dataset)	startVariable($D)
<history>	startElement(history)	startVariable($H)
<date>	startElement(date)	
10/10/59	text("10/10/59")	
</date>	endElement(date)	
</history>	endElement(history)	endVariable($H)
<title>	startElement(title)	startVariable($T) startElement(title)
<subtitle>	startElement(subtitle)	startElement(subtitle)
Study	text("Study")	text("Study")
</subtitle>	endElement(subtitle)	endElement(subtitle)
</title>	endElement(title)	endElement(title) endVariable($T)
</dataset>	endElement(dataset)	endVariable($D)
</datasets>	endElement(datasets)	endVariable($R)

Fig. 3. Events generated by a query tree.

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

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

Fig. 4. A query tree Q and an equivalent query set Q' of absolute XPath expressions.

shows the events forwarded to the application. Only some of the SAX events are seen by the application, namely, exactly those that occur within a `$T` or `$TH` variable event.

2.1.1 *Nested Predicates.*

When an XPath expression contains nested predicates, then the application needs to decompose them into linear XPath expressions. For example, given the expression:

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

the application needs to decompose it into four linear XPath expression, which form the query tree Q shown in Figure 4. The query processor will notify the application of five events, `$R`, `$Y`, `$Z`, `$U`, `$X`, and the application needs to do extra work to combine these events, as follows. It uses two Boolean variables, `b1, b2`. On a `$Z` event, it sets `b1` to `true`; on a `$U` event test the following text value and, if it is > 200, then sets `b2` to `true`. At the end of a `$Y` event, it checks whether `b1=b2=true`. Some extra care is needed for the descendant axis, `//`. This simple method works well in the case when there are few XPath expressions, like in the XML Toolkit described in Section 7. Workloads with large numbers of XPath expressions and nested predicates require more complex processing techniques, and this is outside of the scope of this article. We note, however, that the DFA-based processing method that we study in this article has been incorporated into a highly scalable technique for XPath expressions with nested predicates [Gupta and Suciu 2003].

2.1.2 *The Event-Based Processing Problem.* The problem that we address is: given a query tree Q, preprocess it, and then evaluate it on an incoming XML stream. The goal is to maximize the throughput at which we can process the XML stream.

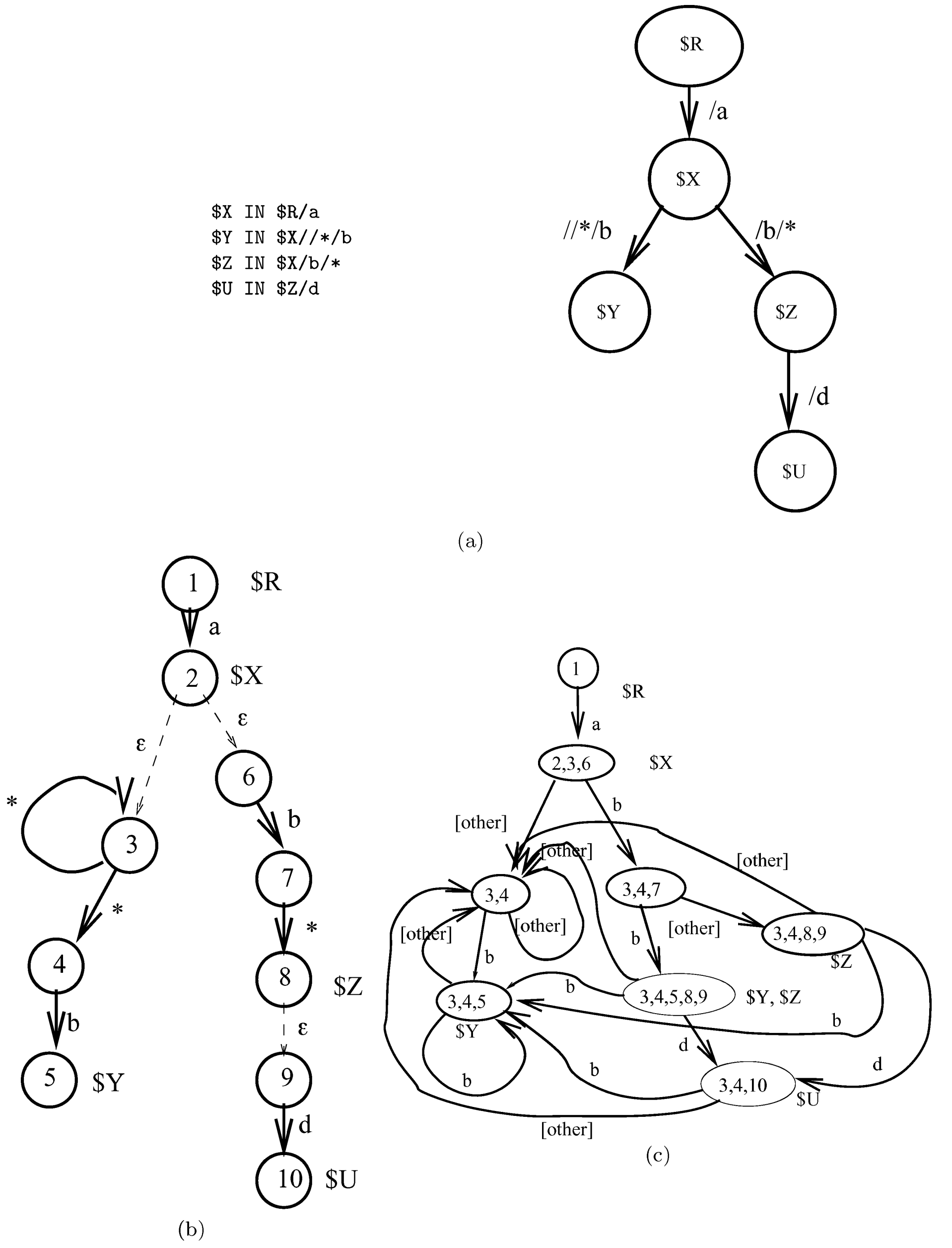
The special case that we will study in Section 4 is that of a query tree in which every XPath expression is absolute, that is, it starts at the root node. In that case we call Q a *query set*, or simply a *set*, because it just consists of a set of absolute XPath expressions. For the purpose of application events only, a query tree Q can be rewritten into an equivalent query set Q', as illustrated in Figure 4. Moreover the DFAs for Q and Q' are isomorphic, so it suffices to study the size of the DFA only for absolute path expressions (Section 4). However, in practice the DFA for Q is somewhat more efficient to compute than that for Q', and for that reason the query processor works on the query tree Q directly.

3. PROCESSING WITH DFAs

3.1 Generating a DFA from a Query Tree

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, for example, to Hopcroft and Ullman [1979]. Our running example will be the query tree P shown in Figure 5(a). Figure 5(b) illustrates the first step: converting the query tree to an NFA, denoted A_n. We follow a popular method for converting XPath expression into an NFA, which was used in Tukwila [Ives et al. 2002], our own work [Green et al. 2003], and in YFilter [Diao et al. 2003]; for a detailed overview of various methods for converting a regular expression to an NFA we refer the reader to Watson's [1993] survey. In Figure 5(b), the transitions labeled $*$ correspond to $*$ or // in P; there is one initial state; there is one terminal state for each variable (`$X`, `$Y`, ...); and there are ε-transitions. The latter 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. This justifies $2 \xrightarrow{\varepsilon} 3$; the other ε-transitions are introduced by compositional rules, which are straightforward and omitted. Notice that, in general, the number of states in the NFA, 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 $A_n(\varepsilon) = \{1\}$, $A_n(ab) = \{3, 4, 7\}$, $A_n(a\omega) = \{3, 4\}$, $A_n(b) = \emptyset$.



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

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

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

$$\begin{aligned} states(A_d) &= \{A_n(w) \mid w \in \Sigma^*\}, \\ \delta(A_n(w), a) &= A_n(wa), a \in \Sigma. \end{aligned} \tag{2}$$

Our running example A_d is illustrated[1] in Figure 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, for example, $\{3, 4, 5, 8, 9\}$ is labeled $Y and $Z. A sax_f flag is defined for each DFA state as follows: its value is true if at least one of the NFA states in that DFA state has sax_f = true; otherwise it is false.

3.2 The DFA at Run Time

One can process an XML stream with a DFA very efficiently. It suffices to maintain a pointer to the current DFA state, and a stack of DFA states. SAX events are processed as follows. On a startElement(e) event we push the current state on the stack, and replace the state with the state reached by following the e transition[2]; on an endElement(e) we pop a state from the stack and set it as the current state. Attributes and text values are handled similarly. At any moment, the states stored in the stack are exactly those at which the ancestors of the current node were processed, and at which one may need to come back later when exploring subsequent children nodes of those ancestors. If the current state has any variables associated to it, then for each such variable $V we send a startVariable($V) (in the case of a startElement) or endVariable($V) (in the case of a endElement) event to the application. If either the current state or the new state we enter has sax_f=true, then we forward the SAX event to the application.

No memory management is needed at run time.[3] Thus, each SAX event is processed in $O(1)$ time, since a transition lookup is implemented as a hash table lookup, and this technique guarantees the throughput at which it can process the stream of XML packets, independently of the number of XPath expressions. The main issue is the size of the DFA, which we discuss next.

4. ANALYZING THE SIZE OF THE DFA

For a general regular expression the size of the DFA may be exponential [Hopcroft and Ullman 1979]. In our setting, however, the expressions are restricted to XPath expressions defined in Section 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 call a DFA *eager* if it is obtained using the standard powerset construction, shown in Equation (2). We call the DFA *lazy* if its states and transitions are constructed at runtime, as we describe in detail in Section 4.2. We shall assume first that the XPath expressions have no predicates of the form text()=S, and, as a consequence, the alphabet Σ is small, then discuss in Section 4.3 the impact of such predicates on the size of

[1]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.
[2]The state's transitions are stored in a hash table.
[3]The stack is a static array, currently set to 1024: this represents the maximum XML depth that we can handle.

the DFA. As explained at the end of Section 2, we will restrict our analysis to absolute XPath expressions, that is, to query sets rather than query trees.

4.1 The Eager DFA

4.1.1 *Single XPath Expression.* A single linear XPath expression can be written as:

$$P = p_0//p_1//\dots//p_k,$$

where each p_i is $N_1/N_2/\dots/N_{n_i}, i = 0,\dots,k$, and each N_j is given by Equation (1) in Section 2.1. We consider the following parameters:

k	$=$	number of //'s,
n_i	$=$	length of p_i, $i = 0,\dots,k$,
m	$=$	max # of $*$'s in each p_i,
n	$=$	length (or depth) of P, that is, $\sum_{i=0,k} n_i$,
s	$=$	alphabet size $=\mid \Sigma \mid$.

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. The proof is in the electronic appendix.

THEOREM 4.1. *Given a linear XPath expression P, define prefix*$(P) = n_0$ *and body*$(P) = (\frac{k^2-1}{2k^2}(n-n_0)^2 + 2(n-n_0) - n_k + 1)s^m$ *when* $k > 0$, *and body*$(P) = 1$ *when* $k = 0$. *Then the eager DFA for P has at most prefix*(P) + *body*(P) *states. In particular, if* $m = 0$ *and* $k \leq 1$, *then the DFA has at most* $(n+1)$ *states.*

We first illustrate the theorem in the case where there are no wild-cards ($m = 0$) and $k = 1$. Then $n = n_0+n_1$ and there are at most $n_0+2(n-n_0)-n_1+1 = n+1$ states in the DFA. For example, if $p = //a/b/a/a/b$, then $k = 1, n = 5$: the NFA and DFA are shown in Figures 6 (a) and (b), respectively, and indeed the latter has six states. This generalizes to $//N_1/N_2/\cdots/N_n$: 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 [Cormen et al. 1990].

When there are wild cards ($m > 0$), the theorem gives an exponential upper bound because of the factor s^m. There is a corresponding exponential lower bound, illustrated in Figures 6(c) and 6(d), showing that the DFA for $p = //a/*/*/*/*$, has 2^5 states. It is easy to generalize this example and see that the DFA for $//a/*/\cdots/*$ has 2^{m+1} states, where m is the number of $*$'s. While a simple hack enables us to $//a/*/\cdots/*$ on an XML document using constant space without converting it into a DFA, this is no longer possible if we modify the expression to $//a/*/\dots/*/b$.

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

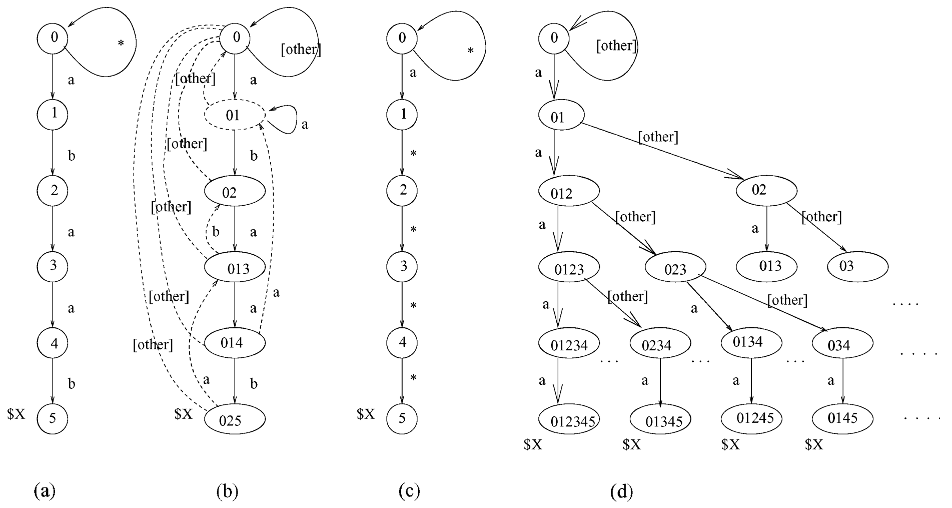


Fig. 6. The NFA (a) and the DFA (b) for `*dfa`. The NFA (c) and the DFA (with back edges removed) (d) for `//a/*/*/*/*`: here the eager DFA has $2^5 = 32$ states.

XQuery already translate a *single* linear XPath expression into DFAs [Ives et al. 2002].

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

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. We generalize this below.

PROPOSITION 4.3. *Consider p XPath expressions:*

$$
\begin{array}{lll}
\$X_1 & \texttt{IN} & \$R//a_1//b \\
 & \dots & \\
\$X_p & \texttt{IN} & \$R//a_p//b
\end{array}
$$

where $a_1, \dots, a_p, b$ are distinct tags. Then the DFA has at least 2^p states.[4]

For all practical purposes, this means that the size of the DFA for a set of XPath expressions is exponential. The theorem below refines the exponential upper bound, and its proof is in the electronic appendix.

[4]Although this requires p distinct tags, the result can be shown with only two distinct tags, and XPath expressions of depths $n = O(\log p)$, using binary encoding of tags.

THEOREM 4.4. *Let Q be a set of XPath expressions. Then the number of states in the eager DFA for Q is at most: $\sum_{P \in Q}(prefix(P)) + \prod_{P \in Q}(1 + body(P))$. In particular, if A, B are constants such that $\forall P \in Q$, $prefix(P) \leq A$, and $body(P) \leq B$, then the number of states in the eager DFA is $\leq p \cdot A + (1 + B)^{p'}$, where p is the number of XPath expressions in Q and p' is the number of such expressions that contain //.*

Recall that $body(P)$ 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 compared with Aho and Corasick's dictionary matching problem [Aho and Corasick 1975; Rozenberg and Salomaa 1997]. 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 4.4 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 4.3 shows. We chose not to strengthen Theorem 4.4 since it would complicate both the statement and proof, with little practical significance.

Sets of XPath expressions like the ones we saw in Example 4.2 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 [Laurikari 2000], but it has never been applied to large numbers of expressions as required in our applications. 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 to the query processor in order to use the lazy DFA, and only serve for the theoretical results.

Formally, let A_l be the lazy DFA. Its states and transitions are described by the following equations, which should be compared to Equation (2) in Section 3.1:

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

$$\delta(A_n(w), a) = A_n(wa), wa \in \mathcal{L}_{data}. \tag{4}$$

Here $\mathcal{L}_{data}$ is the set of all root-to-leaf sequences of tags in the input XML streams. Thus, the size of the lazy DFA is determined by two factors: (1) the number of states, that is, $|\ states(A_l)\ |$, and (2) the size of each state, that is, $|\ A_n(w)\ |$, for $w \in \mathcal{L}_{data}$. Recall that each state in the lazy DFA is represented by a set of states from the NFA, which we call an *NFA table*. In the eager DFA the NFA tables can be dropped after the DFA has been computed, but in the lazy DFA they need to be kept, since we never really complete the construction of the DFA (they are technically needed to apply Equation (4) at runtime). Therefore the NFA tables also contribute to the size of the lazy DFA. We analyze in this section both factors.

4.2.1 *The Number of States in the Lazy DFA.* The first size factor, the number of states in the lazy DFA may be, in theory, exponentially large, and hence is our first concern. 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 [Abiteboul et al. 1999; Buneman et al. 1997] to formalize our notion of schema, where nodes are labeled with tags and edges denote inclusion relationships. A *graph schema* S is a graph with a designated root node, and with nodes labeled with symbols from Σ. Each path from the root defines a word $w \in \Sigma^*$, and the set of all such words forms a regular language denoted $\mathcal{L}_{schema}$. 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 such that for every $i = 0, \ldots, n-1$, there exists an edge from x_i to $x_{(i+1) \bmod n}$. We say that a graph schema is *simple*, if for any two simple cycles $c \neq c'$, we have $c \cap c' = \emptyset$.

We illustrate with the DTD in Figure 7, which also shows its graph schema. This DTD is simple, because the only cycles in its graph schema (shown in Figure 7 (a)) are self-loops. All nonrecursive DTDs are simple. Recall that a simple path in a graph is a path where each node occurs at most once. For a simple graph schema we denote d the maximum number of simple cycles that a simple path can intersect (hence $d = 0$ for nonrecursive schemes), and D the total number of nonempty, simple paths starting at the root: D can be thought of as the number of nodes in the unfolding.[5] In our example $d = 2$, $D = 13$, since the path `book/chapter/section/table/note` intersects two simple cycles, {`table`} and {`note`}, and there are 13 different simple paths that start at the root: they correspond to the nodes in the unfolded graph schema shown in Figure 7 (b). For a query set Q, denote n its depth, that is, the maximum number of symbols in any $P \in Q$ (i.e., the maximum n, as in Section 4.1). We prove the following result in the, electronic appendix:

THEOREM 4.5. *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, on any XML input satisfying the schema, the lazy DFA has at most $1 + D \times (1+n)^d$ states.*

[5]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 Sahuguet [2000]), and hence D is only modestly larger than the number of tags in the DTD.

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

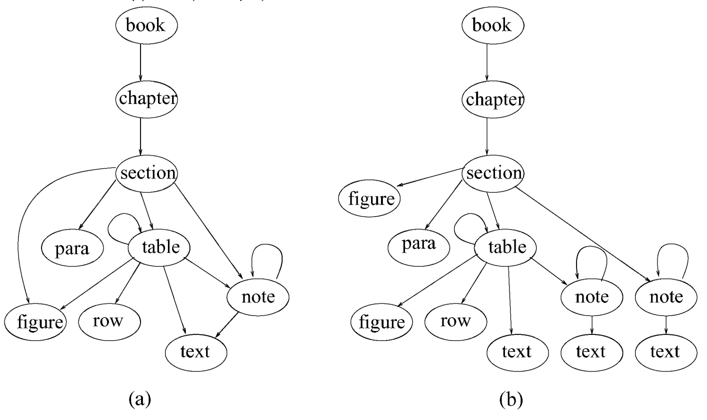


Fig. 7. A simple graph schema for a DTD (a) and its unfolding (b). Here $D = 13$ (since the unfolding has 13 nodes) and $d = 2$ (since two recursive elements may be nested: a `table` may contain a `note`).

The result is surprising, because the number of states does not depend on the number of XPath expressions, only on their depths. In Example 4.2 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 may have $\geq 2^{100000}$ states (see Proposition 4.3). Figure 6(d) shows another example: of the 2^5 states in the eager DFA only nine are expanded in the lazy DFA.

Theorem 4.5 has many applications. First for *nonrecursive* DTDs ($d = 0$), the lazy DFA has at most $1 + D$ states.[6] Second, in *data-oriented* XML instances, recursion is often restricted to hierarchies, for example, 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.

The theorem does not apply, however, to *document-oriented* XML data. These have nonsimple 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. This is illustrated in Figure 8(a). For such cases we give an upper bound on the size of the lazy DFA in terms of data guides [Goldman and Widom 1997]. Given an XML data instance, the data guide G is that schema (a) which is deterministic.[7] (b) which captures exactly the sequence of labels in the data, $\mathcal{L}_{schema} = \mathcal{L}_{data}$, and (c) where G is

[6]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 nonempty, simple path.

[7]For each label $a \in \Sigma$, a node can have at most one child labeled with a.

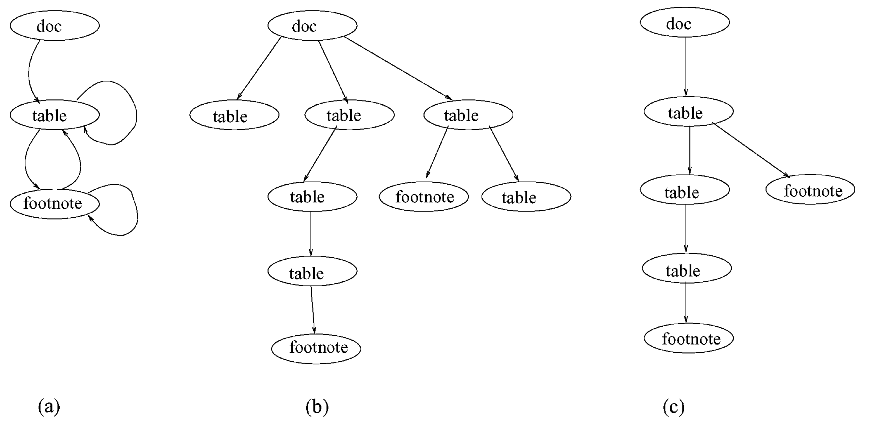


Fig. 8. A nonsimple graph schema (a), an XML instance (b), and its data guide (c).

unfolded, that is, it is a tree. The latter property is possible to enforce since $\mathcal{L}_{data}$ is finite, and hence the data guide has no cycles. Figure 8 illustrates the connection between graph schemas, XML data, and data guides. The graph schema in Figure 8(a) is nonsimple, and shows all possible nestings that are allowed in the data. An actual XML instance in Figure 8(b) uses only some of these nestings. The data guide in Figure 8(c) captures precisely these nestings.

Since data guides are graph schemas with $d = 0$, Theorem 4.5 applies and gives us the following:

COROLLARY 4.6. *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. To understand why, consider the case of XML documents representing structured text, with elements such as `footnote`, `table`, `figure`, `abstract`, `section`, where the DTD allows these elements to be nested arbitrarily. Typical documents will have paths like `section/table`, `section/figure`, `section/figure/footnote`, and hence the dataguide for large enough collection of such documents is quite likely to contain all these paths. However, many other paths are quite unlikely to occur in practice, for example, `table/figure/footnote`, `figure/section/abstract`, and therefore they are unlikely to occur in the dataguide, even though they are technically permitted by the DTD. Thus, the number of nodes in the dataguide is typically much smaller than the theoretical upper bound. This is a general observation, which tends to hold on most practical XML data found in most domains. In order to find a counterexample, one has to go to the domain of Natural Language Processing: Treebank [Marcus et al. 1993] is a large collection of parsed English sentences and its dataguide has $G = 340,000$ nodes, as reported in Liefke and Suciu [2000].

4.2.2 *Size of NFA Tables.* The following proposition ensures that the NFA tables do not increase exponentially:

PROPOSITION 4.7. *Let Q be a set of p XPath expressions, of maximum depth n. Then the size of each NFA table in the DFA for Q is at most $2np$.*

The proof follows immediately from the observation that the NFA for one XPath expression has $n + k \leq 2n$ states; hence each NFA table may contain at most $2np$. 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 Section 6.

4.3 Predicates

We now lift the restriction on predicates, and discuss their impact on the number of states in the DFA. Each linear XPath expression can now end in a predicate `text()=S`, see Equation (1) in Section 2.1. The only difference is that now we can no longer assume that the alphabet Σ is small, since the number of distinct strings `S` in the query workload can be very large. As a matter of notation, we follow the W3C standards and use a rather confusing syntax for the symbol `text()`. An XPath expression may end in a predicate denoted `text()=S`; this matches a SAX event of the form `text(S)`; hence, the predicate becomes a transition labeled `text(S)` in the NFA and the DFA.

For a given set of XPath expressions, Q, let Σ denote the set of all symbols in the NFA for 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, two-tier structure: first an NFA over Σ_t, followed by some Σ_s-transitions into sink states, that is, 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 $\Sigma_s = \{\texttt{text}(\texttt{S}_1), \ldots, \texttt{text}(\texttt{S}_\texttt{q})\}$, and let Q_i, $i = 1, \ldots, q$, be the set of XPath expressions in Q that end in $\texttt{text}() = \texttt{S}_\texttt{i}$; we assume without loss of generality that every XPath expression in Q ends in some predicate in Σ_s, and hence $Q = Q_1 \cup \cdots \cup Q_q$. 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 4.1, Theorem 4.5, and Corollary 4.6, apply to each s_i. We prove the following in the electronic appendix.

THEOREM 4.8. *Given a set of XPath expressions Q, containing q distinct predicates 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. THE STREAM INDEX (SIX)

Parsing and tokenizing the XML document is generally accepted to be a major bottleneck in XML processing. An obvious solution is to represent an XML document in binary, as a string of binary tokens. In an XML message system, the messages are now binary representations of XML, rather than real XML, or they are converted into binary when they enter the system. Some commercial implementations adopt this approach in order to increase performance

[Florescu et al. 2003]. The disadvantage is that all servers in the network must understand that binary format. This defeats the purpose of the XML standard, which is supposed to address precisely the lack of interoperability that is associated with a binary format.

We favor an alternative approach: keep the XML packets in their native text format, and add a small amount of binary data that allows fast access to the document. We describe here one such technique: a different technique based on the same philosophy is described in Gupta et al. [2003].

5.1 Definition

Given an XML document, a *Stream IndeX* (SIX) for that document is an ordered set of byte offsets pairs:

(beginOffset, 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, attached to the XML packet somehow (e.g., using the DIME standard [Corp. n.d.]), then sent along with the XML stream and used by every consumer of that stream (e.g., by every router, in XML routing). A server that does not understand the SIX can simply ignore it.

The SIX is sorted by beginOffset. The query processor starts parsing the XML document and matches SIX entries with XML tags. Depending on the queries that need to be evaluated, the query processor may decide to skip over elements in the XML document, using endOffset. Thus, a simple addition of two integers replaces parsing an entire subelement, generating all SAX events, and looking for the matching end tag. This is a significant savings.

The SIX module (see Figure 1 in Section 2.1) offers a single interface: skip(k), where $k \geq 0$ 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(0) call, issued after reading <c>:

```
XML stream:
<a> <b> <c> <d> </d> </c> <e> </e> </b> <f>  . . .
           |
        skip(0)
parser:
<a> <b> <c>               <e> </e> </b> <f>  . . .
```

while the following shows the effect of a skip(1) call:

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

5.2 Using the SIX

A SIX can be used by any application that processes XML documents using a SAX parser.

Example 5.1. Consider a very simple application counting how many products in a stream of messages have more than 10 complaints:

```
count(/message/product[count(complaint) >= 10]).
```

While looking for `product`, if some other tag is encountered then the application issues a `skip(0)`. Inside a `product`, the application listens for `complaint`: if some other tag is read, then issue a `skip(0)`. If a `complaint` is read then increment the count. If the count is `>=10` then issue `skip(1)`, otherwise `skip(0)`.

A DFA can use a SIX effectively. From the transition table of a DFA state it can see what transitions it expects. If a begin tag does not correspond to any transition and its `sax_f` flag is set to `false`, then it issues a `skip(0)`. As we show in Section 6, this results in dramatic speedups.

5.3 Implementation

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 need to be removed (as we explain next), and skipping over them can be achieved by skipping over their children, yielding largely the same benefit.

The SIX works on arbitrarily large XML documents. 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.[8] The `endOffset` cannot wrap around: elements longer than 2^{32} bytes cannot be represented in the SIX and must be removed.

The SIX is just a piece of binary data that needs to travel with the XML document. Some application decides to compute it and attaches it to the XML document. Later consumers of that document can then benefit from it. In our implementation the SIX is a binary file, with the same name as the XML file and with extension `.six`. In an application like XML packet routing, the SIX needs to be attached somehow to the XML document, for example, by using the DIME format [Corp. n.d.], and identified with a special tag. In both cases, applications that understand the SIX format may use it, while those that don't understand it will simply ignore it.

The SIX for an XML document is constructed while the XML text output is generated, as follows. The application maintains a circular buffer containing a tail of the SIX, and a stack of pointers into the buffer. The application also maintains a counter representing the total number of bytes written so far into the XML output. Whenever the application writes a `startElement` to the XML output, it adds a (`beginOffset`, `endOffset`) entry to the SIX buffer, with

[8]The only XML document for which the SIX cannot be computed is one that has a text value longer than 2^{32} bytes. In that case, the SIX is not computed, and replaced with an error code.

`beginOffset` set to the current byte count, and `endOffset` set to NULL. Then it pushes a pointer to this entry on the stack. Whenever the application writes a `endElement` to the XML output, it pops the top pointer from the stack, and updates the `endOffset` value of the corresponding SIX entry to the current byte offset. In most cases, the size of the entire SIX is sufficiently small for the application to keep it in the buffer. However, if the buffer overflows, then application fetches the bottom pointer on the stack and deletes the corresponding SIX entry from the buffer, then flushes from the buffer all subsequent SIX entries that have their `endOffset` value completed. This, in effect, deletes a SIX entry for a large XML element.

5.4 Speedup of a SIX

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, that is, which 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 Figure 3, we have $n = 8$ (one has to count only the `startElement()` and `text()` SAX events), $n_0 = 4$, and hence $\theta = 0.5$. The maximum speedup from a SIX is $1/\theta$. At one extreme, the expression `/no-such-tag` has $\theta = 0$, and may result in arbitrary large speedups, 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's effectiveness, the $*$'s and //'s should be eliminated, or at least reduced in number, by specializing the XPath expressions with respect to the DTD, using *query pruning*. This is a method, described in Fernandez and Suciu [1998], by which an XPath expression is specialized to a certain DTD. For example the XPath expression `//a` may be specialized to `(/b/c/d/a) | (/b/e/a)` by inspecting how a DTD allows elements to be nested. Query pruning eliminates all $*$'s from the DFA, and therefore increases the effectiveness of the SIX.

6. EXPERIMENTS

We evaluated our techniques in a series of experiments addressing the following questions. How much memory does the lazy DFA require in practice? How efficient is the lazy DFA in processing large workloads of XPath expressions? And how effective is the SIX?

We used a variety of DTDs summarized in Figure 9. All DTDs were downloaded from the Web, except `simple`, which is a synthetic DTD created by us. We generated synthetic XML data for each DTD using the generator from `http://www.alphaworks.ibm.com/tech/xmlgenerator`. For three of the DTDs we also found large, real XML data instances on the Web, which are shown as three separate rows in the table: `protein(real)`, `nasa(real)`, `treebank(real)`.

	File size (kB)	Max. depth	Avg. depth	# of elems. (DTD)	# of elems. (XML)	Recursive?	Simple?
simple	27432	22	19.9	12	350338	Yes	Yes
prov `www.wapforum.org`	25888	22	19.9	3	234531	No	Yes
ebBPSS `www.ebxml.org`	25624	25	10.0	29	356907	Yes	Yes
protein `pir.georgetown.edu`	22952	7	4.6	66	700270	No	Yes
protein(real)	700408	7	5.1		21305818		
nitf	51964	17	8.5	133	439871	Yes	No
nasa `xml.gsfc.nasa.gov`	8000	13	6.6	109	145146	Yes	No
nasa(real)	24488	8	5.5		476646		
treebank	39664	12	11.1	250	830769	Yes	No
treebank(real)	57248	36	7.8		2437666		

Fig. 9. Sources of data used in experiments. Only three real data sets were available.

For example, the row for `protein` represents the synthetic XML data while `protein(real)` the real XML data, and both have the same DTD.

We generated several synthetic workloads of XPath expressions for each DTD, using the generator described in Diao et al. [2003]. It allowed us to tune the probability of $*$ and //, denoted $Prob(*)$ and $Prob(//)$, respectively, and the maximum depth of the XPath expressions, denoted n. In all our experiments below, the depth was $n = 10$.

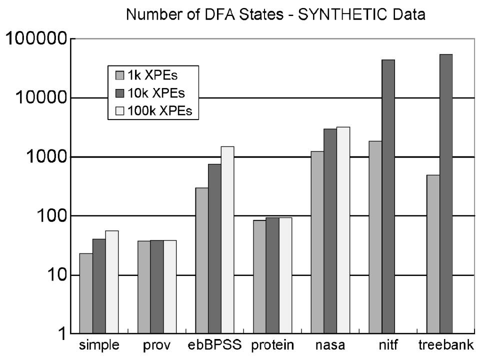
Our system was a Dell Dual P-III 700-Mhz, 2-GB RAM running RedHat 7.1. We compiled the Lazy DFA with the gcc compiler version 2.96 without any optimization options. We also ran a different system, YFilter, which was written in Java: here we used Java version 1.4.2_04.

6.1 Validation of the Size of the Lazy DFA

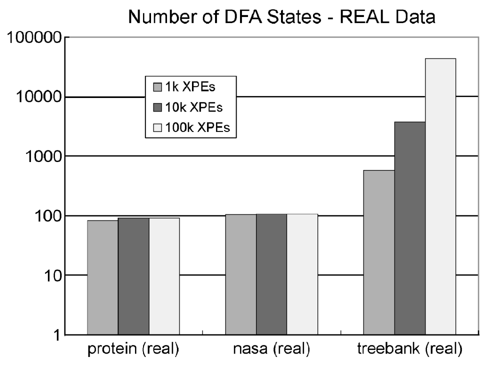
The goal of the first set of experiments was to evaluate empirically the amount of memory required by the lazy DFA. This is as a complement to the theoretical evaluation in Section 4. For each of the datasets we generated workloads of 1k, 10k, and 100k XPath expressions, with $Prob(*) = Prob(//) = 5\%$ and depth $n = 10$.

We first counted the number of states generated in the lazy DFA. Recall that, for simple DTDs, Theroem 4.5 gives the upper bound $1 + D \times (1 + n)^d$ on the number of states in the lazy DFA, where D is the number of elements in the unfolded DTD, d is the maximum nesting depths of recursive elements, and n is the maximum depth of any XPath expression. For real XML data, Corollary 4.6 offers the additional upper bound $1 + G$, where G is the size of the dataguide of the real data instance, which, we claimed, is in general small for a real data instance. By contrast, a synthetic data instance may have a very large dataguide, perhaps as large as the data itself, and therefore the upper bound in Corollary 4.6 is of no practical use.

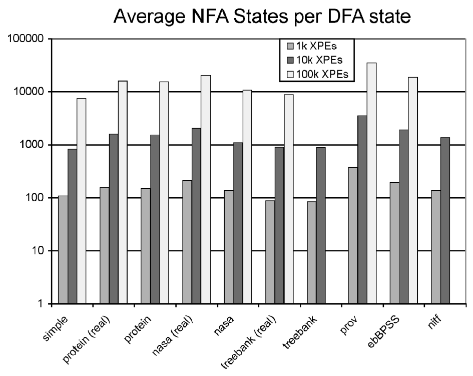
Figure 10(a) shows the number of states in the lazy DFA on *synthetic* XML data. The first four DTDs are simple, and the number of states was indeed smaller than the bound in Theorem 4.5, sometimes significantly smaller. For



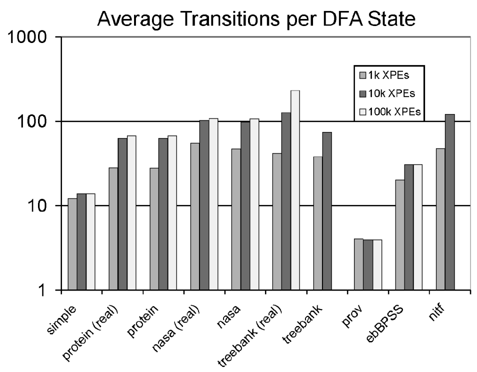
(a)



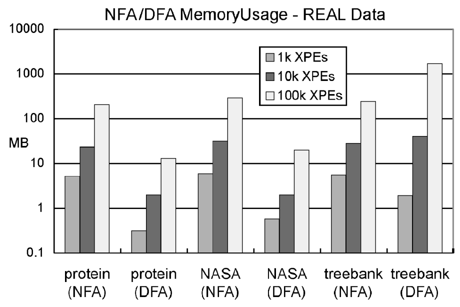
(b)



(c)



(d)



(e)

Fig. 10. Size of the lazy DFA for synthetic data (a), and real data (b); average size of an NFA table (c), and of a transition table (d); total memory used by a lazy DFA (e). 1k XPEs means 1000 XPath expressions.

example ebBPSS has 1479 states for 100k XPath expressions, while the theoretical upper bound, taking[9] $D = 29$, $d = 2$, $n = 10$, is 3510. The last three DTDs were not simple, and Theorem 4.5 does not apply. In two cases (nitf and treebank, for 100,000 expressions) we ran out of memory.

Figure 10(b) shows the number of states in the lazy DFA for *real* data. Here the number of states is significantly smaller than in the previous graph. This is explained by the fact that real XML instances had a small dataguide, which limited the number of states in the lazy DFA. For example, for the real nasa data instance the number of states was 103, 107, and 108, respectively: contrast that to 1470, 2619, 2874 for the synthetic nasa data instance. The only data instance with a large data guide was treebank, where G was about 340,000 and the lazy DFA had 43,438 states on the largest workload (100,000 XPath expressions).

The huge difference between the synthetic and the real data set is striking, and makes one reflect on the limitations of current XML data generators. The lesson for our purposes is that the size of the lazy DFA is small or medium on real data sets, but can be prohibitively large on synthetic data sets.

Next, we measured experimentally the average size of the NFA tables in each DFA state, that is, the average number of NFA states per DFA state. Figure 10(c) shows the experimental results. The average size of the NFA tables grew linearly with p. This is consistent with the theoretical analysis: Proposition 4.7 gives an upper bound of $2np$, and hence $20p$ in our case, where p is the number of XPath expressions. The experiments showed that bound to be overly pessimistic and the real value to be closer to $p/10$, however. Even so, the total size of the NFA tables was large, since this number needed to be multiplied with the number of states in the lazy DFA.

We also measured the average number of transitions per DFA state. These transitions were stored in a hash table at each state in the lazy DFA, and hence they also contributed to the total size. Notice that the number of transitions at a state is bounded by the number of elements in the DTD. Our experimental results in Figure 10(d) confirm that. The transition tables are much smaller than the NFA tables.

Next we measured the total amount of memory used by the lazy DFA, expressed in MB's: this is shown in Figure 10(e). The most important observation is that the total amount of memory used by the lazy DFA grew largely linearly with the number of XPath expressions. This is explained by the fact that the number of states was largely invariant, while the average size of an NFA table at each state grew linearly with the workload. We also measured the amount of memory used by a naive NFA, without any of the state sharing optimization implemented in YFilter. The graph shows that this was comparable to the size of the lazy DFA. On the one hand, the total size of the NFA tables in the lazy DFA was larger than the number of states in the NFA; on the other hand, the DFA made up for this by having fewer transition tables.

None of the experiments above included any predicates on data values. To conclude our evaluation of the memory usage of the lazy DFA, we measured the

[9]We took here D to be the number of elements in the DTD. The real value of D may be larger, due to the unfolding.

	nasa	protein
Xerces C++ SAX Parser	5.449 MB/s	4.238 MB/s
Xerces Java SAX Parser	6.678 MB/s	6.518 MB/s
Xerces C++ SAX2 Parser	2.581 MB/s	1.902 MB/s
Xerces Java SAX2 Parser	6.663 MB/s	6.503 MB/s
Lazy DFA C++ Parser	8.476 MB/s	6.429 MB/s

Fig. 11. The throughput of various XML parsers.

impact of predicates. Recall that the theoretical analysis for this case was done in Section 4.3, and we refer to the notations in that section. We generated a workload of 200,000 XPath expressions with constant values. We used a subset of size 9.12 MB of the `protein` data set, and selected randomly constants that actually occurred in this data. In order to select values randomly from this data instance, we had to store the entire data in main memory. For that reason, we used only a subset of the `protein` data set. The number of distinct constants used was $q = 29740$. The first tier of the automaton had 80 states (slightly less than in Figure 10(b) because we used only a fragment of the `protein` data), while the number of additional states was 63,412. That is, each distinct constant occurring in the predicates contributed to approximatively two new states in the second tier of the automaton. The average size of the NFA tables at these states was at most as large as the average number of XPath expressions containing each distinct constant, that is, 200,000/29,740 $\approx$ 6.7. Since these states had no transition tables, each distinct value occurring in any of the predicates used about 13.4$*$4 $\approx$ 54 bytes of main memory. While nonnegligible, this amount was of the same order of magnitude as the predicate itself.

6.2 Throughput

In our second sets of experiments, we measured the speed at which the lazy DFA processed the real XML data instances `nasa` and `protein`. Our first goal here was to evaluate the speed of the lazy DFA during the *stable phase*, when most or all of its states have been computed, and the lazy DFA reaches its maximum speed. Our second goal was to measure the length of the *warmup phase*, when most time is spent constructing new DFA states. To separate the warmup phase from the stable phase, we measured the instantaneous throughput, as a function of the amount of XML data processed: we measured at 5-MB intervals for `nasa` and 100-MB intervals for `protein`, or more often when necessary.

We compared the lazy DFA to YFilter [Diao et al. 2003], a system that uses a highly optimized NFA to evaluate large workloads of XPath expressions. There are many factors that make a direct comparison of the two systems difficult: the implementation language (C++ for the lazy DFA vs. Java for YFilter), the XML parser (a custom parser vs. the Xerces Java parser), and different coding styles. While a perfect calibration is not possible, in order to get a meaningful comparison we measured the throughput of the Xerces C++ SAX and SAX2 parsers, the Xerces Java SAX and SAX2 parsers, and the parser of the lazy DFA. The results are shown in Figure 11. Contrary to our expectations, the Xerces C++ SAX parser was slightly slower than the Java SAX Parser, while the C++ SAX2 parser was even slower. Assuming that the Java and C++

versions used identical algorithms, this suggests that a Java program should run slightly faster than a C++ program on our platform. On the other hand, the lazy DFA parser was faster on average than the Xerces Java SAX2 parser (used by YFilter); hence, all things being equal, the lazy DFA should run slightly faster than YFilter (at least on nasa). While these numbers underly the difficulty of a direct comparison, they also suggest that any difference in the throughput of the two systems that are attributable to the implementation language and the parser are relatively small. Therefore we report below absolute values of the throughput and do not attempt to normalize them.

In Figures 12(a) and 12(b), we show the results for workloads of varying sizes (500 to 500,000 XPath expressions for nasa, 1000 to 1,000,000 for protein). In all workloads the maximum depth was $n = 10$, and $Prob(*) = Prob(//) = 0.1$. The most important observation is that in both graphs the lazy DFA reached indeed a stable phase, after processing about 5–10 MB of nasa data or 50 MB of protein data, where the throughput was constant, that is, independent of the size of the workload. The throughput in the stable state was about 3.3–3.4 MB/s for nasa and about 2.4 MB/s for protein.

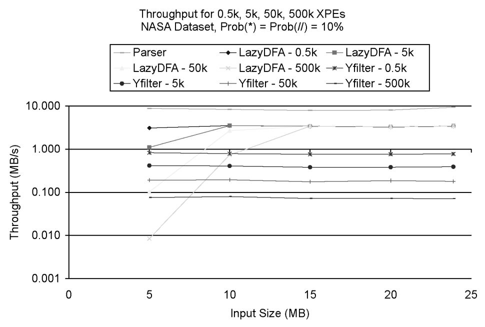
By contrast, the throughput of YFilter decreased with the number of XPath expressions: as the workload increased by factors of 10, the throughput of YFilter decreased by an average factor of 2. In general, however, the throughput of the lazy DFA was consistently higher than that of YFilter, by factors ranging from 4.6 to 48. The throughput was especially higher for large workloads.

The high throughput of the lazy DFA should be balanced by two effects: the amount of memory used and the speed of the warmup phase. To get a sense of the first effect, notice that the lazy DFA used almost the entire 2 GB of main memory on our platform in some of the tests. In one case, when we tried to run it on the nasa dataset with 1,000,000 XPath expressions, we ran out of memory.[10] By contrast, YFilter never used more than 60 MB of main memory on any workload.

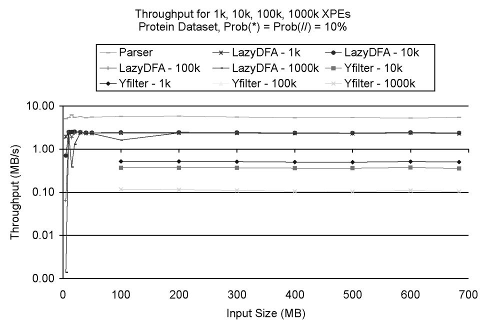
To see the second effect, we report the total running time of the entire data instance in Figure 13(a). The gains of the lazy DFA over YFilter are now smaller, between factors of 1.6 and 8.3. In one case, YFilter was faster than the lazy DFA by a factor of 2. Notice that protein was much larger, allowing the lazy DFA more time to recover from the high warmup cost: here the lazy DFA was always faster. The difference from the graphs in Figure 12 is explained by the fact that the warmup phase is expensive.

Next, we ran similar experiments testing the sensitivity of the lazy DFA to increasing numbers of $*$'s and //'s in the workload of XPath expressions. Figures 12(c) and 12(d) show the variation of the throughput when $Prob(*)$ or $Prob(//)$ varied. We only show here the results for the nasa dataset; those for protein were similar. These graphs show the same general trend as those in Figures 12(a) and 12(b). One interesting observation here is that the warmup

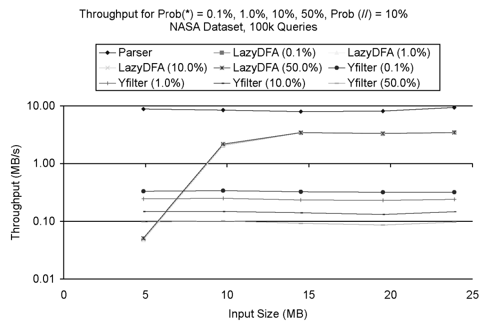
[10]The same test, however, runs fine on a Solaris platform, since the Solaris operating system has a better memory management module. The overall throughput of the lazy DFA was also higher on the Solaris platform. In a preliminary version of this work [Green et al. 2003], we reported experiments on a Solaris platform.



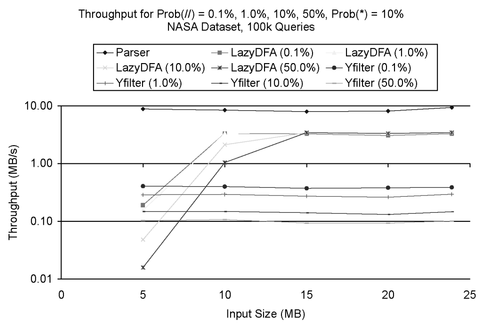
(a)



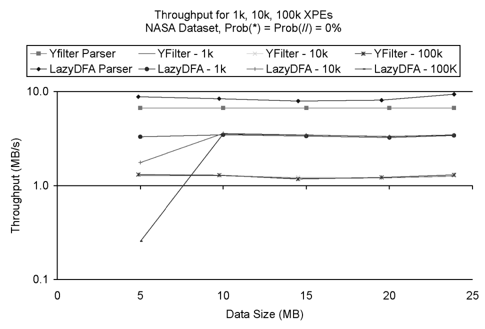
(b)

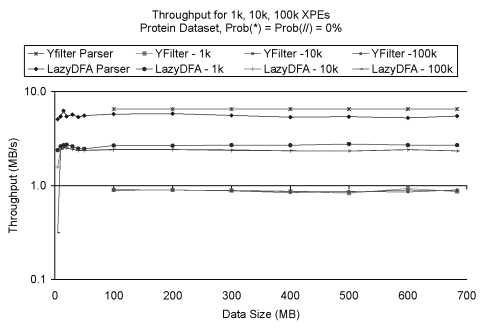


(c)



(d)





ACM Transactions on Computational Logic, Vol. ??, No. 4, 12 2004.

(e) (f)

Fig. 12. The throughput of the lazy DFA and YFilter, as a function of the amount of XML data consumed. Varying workload sizes (a), (b); varying probabilities for ∗ and // (c), (d); workloads without ∗ and // (e), (f). Here 1k XPE means 1000 XPath expressions.

phase of the lazy DFA is not affected by the presence of ∗'s, only by that of //'s.

A type of workload of particular interest in practice is one without any occurrences of ∗ and //. We ran a similar set of experiments on such workloads, and we report the results in Figures 12(e) and 12(f). We also report the absolute running times in Figure 13(b). On such a workload both the NFA optimized

Prob (*) = 10%, Prob (//) = 10%

	`nasa`	
XPEs	lazyDFA	YFilter
500	7.14	30.73
5,000	9.99	60.79
50,000	54.89	129.54
500,000	602.68	323.58

	`protein`	
	lazyDFA	YFilter
1,000	289.52	1,349.65
10,000	285.88	1,872.38
100,000	355.41	2,944.26
1,000,000	3899.58	6,269.34

(a)

$Prob(*) = Prob(//) = 0$:

	nasa		protein	
	lazyDFA	YFilter	lazyDFA	YFilter
1,000	7.126	19.14	253.86	771.95
10,000	8.289	18.81	288.21	769.88
100,000	24.941	18.93	299.87	777.59

(b)

Fig. 13. Absolute running times in seconds for workloads with (a) and without (b) occurrences of $*$ and //.

by YFilter and the DFA become two isomorphic Trie structures. As before, the lazy DFA is slow during the warmup phase, which determined one total running time to be less than, that for YFilter shown in Figure 13(b).

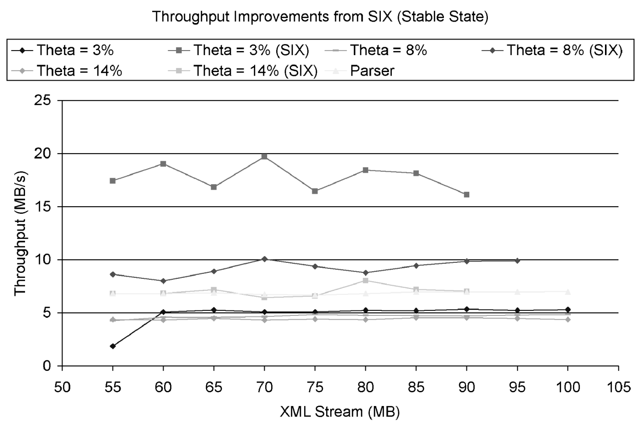
6.3 Evaluation of the SIX

In this set of experiments, we evaluated the SIX on synthetic `nitf` data,[11] with 10,000 XPath expressions using `0.2%` probabilities for both the // and the $*$'s. The justification for these low values is based on the discussion at the end of Section 5.4: the SIX is ineffective for workloads with large numbers of // and $*$, and there exists techniques (e.g., query pruning) for eliminating both // and $*$ by using a schema or a DTD. In order to vary the selectivity parameter θ (Section 5.4), we made multiple, disjoint copies of the `nitf` DTD, and randomly assigned each XPath expression to one such DTD: θ decreased when the number of copies increased. We generated about 50 MB of XML data, then copied it to obtain a 100-MB 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 6.7 MB, or about `7%` of the XML data.

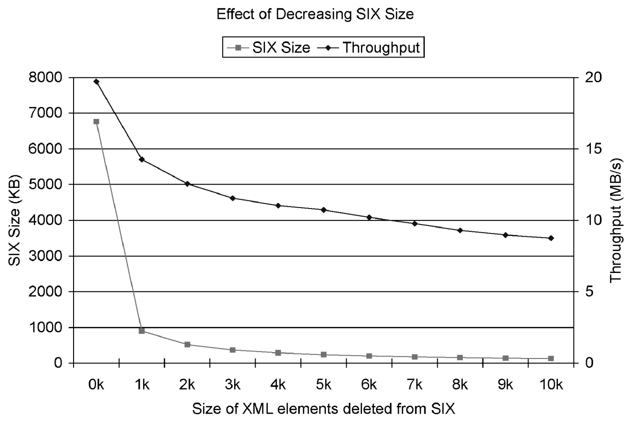
Figure 14(a) shows the throughput with a SIX, and without a SIX, for all three selectivities. Without a SIX, the throughput was constant at around 5 MB/s. This was slightly higher than for the previous experiments because of our optimization of the "failure state" transitions: when the lazy DFA entered the failure state, where all transitions lead back to itself, the lazy DFA processor did not look up the next state in the transition table (which was a hash table, in this case with only one entry), but simply kept the same current state.

When run with a SIX, the throughput increased significantly for low selectivities. For $\theta = 0.03$ the throughput oscillated around 16–19 MB/s, resulting in an average speedup of 3.3. Notice that the throughput of the lazy DFA with a SIX

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



(a)



(b)

Fig. 14. Throughput improvement from the SIX (a), and the effect of decreasing the SIX size by deleting "small" XML elements (b).

was higher in all cases, even significantly higher than the parser's throughput, which was around 6.8 MB/s. This was because the SIX allowed large portions of the XML document to be skipped entirely, and thus could be faster than parsing the entire document.

Next, we measured how much we could 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. Figure 14(b) shows the throughput as a function of the cutoff size for the XML elements. The more elements were deleted from the SIX, the smaller the throughput. However, the SIX size also decreased, and did so much more dramatically. For example, at the 1k data point, when we deleted from the SIX all elements whose size was $\leq$ 1k bytes, the throughput decreased to 14 MB/s from a high of 19 MB/s, but the size of the SIX decreased to a minuscule 898 bytes, from a high of 6.7 KB. Thus we could reduce the SIX more than seven times, yet retain 73%

of the benefit in the throughput. The explanation is that although the number of elements that can be skipped decreases, their average size increases. In other words, we only miss the short elements, which are not very useful to the SIX anyway.

6.4 Discussion

Our experiments demonstrate clearly that the DFA technique is effective at processing XML data at a high, sustained throughput. The most important property is that the throughput remains constant as the number of XPath expressions in the workload increases. This makes the technique attractive for applications that need to guarantee a certain throughput, independently of the size of the workload.

The experiments also show that by computing the DFA lazily one avoids, in most cases of practical interest, an exponential state explosion. We have proven two theoretical upper bounds on the number of states of the lazy DFA. Our experiments confirmed a small number of states in both cases. However, the existence of "bad" cases, that is, data instance that might cause a state explosion in the lazy DFA, is not completely ruled out. One can generate such XML instances syntheticaly, but it is unclear whether such instances exist in practice: the only instance we found that caused the number of states to grow into the tens of thousands was `treebank`, whose complex structure is specific to Natural Language, and is not typical in XML data. Still, it is wise to implement a safety valve in a lazy DFA processor, for example, by deleting all states and restarting from the initial state when it runs out of memory.

On the downside, our experiments have pointed out two limitations in our current implementation of the lazy DFA: a rather high warmup cost, and large memory consumption by the NFA states. We discuss here both limitations and their possible solutions.

6.4.1 *Warmup.* First, let us address the high cost of the warmup phase. During this phase the lazy DFA acts precisely like an NFA, only it has to memorize all states it sees. Currently, our implementation of the NFA is very simple, without any optimizations, and this leads to a high warmup cost. In contrast, YFilter consists of an optimized version of the NFA, and it runs much faster than the lazy DFA during warmup. YFilter first constructs an NFA for each XPath expressions in the workload, then identifies common prefixes and eliminates them. For example, if given the two expressions `/a//b/*/a//c` and `/a//b/*/a/c`, YFilter would optimize the NFA to share states and transitions for their common prefix `/a//b/*/a`, and only branch at the `/c` and `//c` transitions. When extended to large workloads, this optimization results in significant space and time savings over a naive NFA approach. The solution here is to apply the same optimization to the NFA used by the lazy DFA. It suffices to replace the currently naive NFA with YFilter's optimized NFA, and leave the rest of the lazy DFA unchanged. This would speed up the warmup phase considerably, making it comparable to YFilter, and would not affect the throughput in the stable phase.

With or without optimizations, the manipulation of the NFA tables is expensive, and we have put a lot of thought into their implementation. There are three operations done on NFA tables: create, insert, and compare. To illustrate their complexity, consider an example where the lazy DFA ends up having 10,000 states, each with an NFA table with 30,000 entries, and that the alphabet Σ has 50 symbols. Then, during the warmup 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`, and 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, and thus comparisons with negative answers take $O(1)$ in virtually all cases.

6.4.2 *Memory.* Second, we discuss the high memory consumption of the lazy DFA. As our experiments showed, this is due to the NFA tables, not the number of states in the lazy DFA. There are several possible approaches to address this, but studying their effectiveness remains part of future work. The simplest one is to adopt the YFilter optimizations as explained above: in addition to speeding up the warmup phase, this can also decrease the average size of the NFA tables. A second approach is to delete the NFA tables from "completed" DFA states. A completed DFA state is one in which all its transitions have already been expanded. The NFA table in a DFA state is only needed when a new transition is followed, in order to construct the new destination DFA state. Once all such transitions have been expanded, there is no more need for the NFA table.

We notice however that, when run on smaller workloads, the lazy DFA uses far less memory than many other systems. Peng and Chawathe [2003] evaluated the throughput and the memory usage of seven systems, including the XML Toolkit (which is based on the lazy DFA and is described here in Section 7). In their evaluation, the XML Toolkit used by far the least amount of memory, in some cases by several orders of magnitude.

Finally, we discuss here the effectiveness of the SIX. Like any index, it only benefits queries or workloads that retrieve only a very small portion of the data. Our experiments showed the SIX to be effective for workload of 10,000, but on a dataset where we decreased the selectivity artificially. In order to use the SIX in an application like XML packet routine, one needs to cluster XPath expressions into workloads in order to reduce the θ factor for each workload. When this is possible, then the SIX can be very effective.

7. AN APPLICATION: THE XML TOOLKIT (XMLTK)

We describe here an application of our XPath processing techniques to a set of tools for highly performant processing of large XML files. The XML Toolkit is modeled after the Unix tools for processing text files, and is available online at `http://xmltk.sourceforge.net`.

The tools currently in the XML Toolkit are summarized in Figure 15. Every tool inputs/outputs XML data via standard i/o, except `file2xml` which takes a directory as an input and outputs XML to the standard output.

Command	Arguments (fragment) P = XPath expr, N = number	Brief description
xsort	(−c P (−e P (−k P)*)*)*	Sorts an XML file
xagg	(−c P (−a aggFun valP)*)*	Computes the aggregate function aggFun (see Figure 19 in electronic appendix)
xnest	(−e P ((−k P)*) | −n N)*	Groups elements based on key equality or number
xflatten	(−r)? − e P	Flattens collections (deletes tags, but not content)
xdelete	−e P	Removes elements or attributes
xpair	(−e P − g P)*	Replicates an element multiple times, pairing it with each element in a collection
xhead	(−c P (−e P (−n N)?)*)*	Retains only a prefix of a collection
xtail	(−c P (−e P (−n N)?)*)*	Retains only a suffix of a collection
file2xml	−s dir	Generates an XML file for the dir file directory hierarchy

Fig. 15. Current tools in the XML Toolkit.

The xsort tool is by far the most complex one and we describe it in more detail. The others are briefly illustrated in the electronic appendix, but we note that most can be used in quite versatile ways [Avila-Campillo et al. 2002]. When illustrating the tools we will refer to the DBLP database [Ley n.d.]. We used a dataset with 256,599 bibliographic entries.

7.1 Sorting

The command below sorts the entries in the file dbpl.xml in ascending order of their year of publication[12]:

```
xsort -c /dblp -e * -k year/text() dblp.xml > sorted-dblp.xml.
```

The first argument, -c, is an XPath expression that defines the *context*: this is the collection under which we are sorting: in our example this matches the root element, dblp. The second argument, -e, specifies the *items* to be sorted under the context: on the DBLP data this matches elements like the book, inproceedings, article, etc. Finally, the last argument, -k, defines the *key* on which we sort the items; in our example this is the text value of the year element. The result of this command is the file sorted-dblp.xml which lists the four publications in increasing order of the year. In case of publications with the same years, the document order is preserved.

The command arguments for xsort are shown in Figure 15, with some details omitted. There can be several context arguments (-c), each followed by several item arguments (-e), and each followed by several key arguments (-k). The semantics is illustrated in Figure 16. First, all context nodes in the tree are identified (denoted c in the figure): all nodes that are not below some context node are simply copied to the output in unchanged order. Next, for each context node, all nodes that match that context's item expressions are identified

[12]Unix shells interpret the wild cards, so the command should be given like: xsort -c /dblp -e "*" .... We omit the quotation marks throughout the article to avoid clutter.

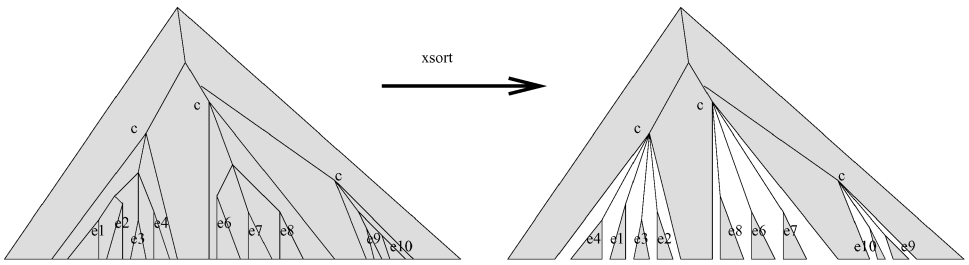


Fig. 16. Semantics of xsort. Under each *context* node the *item* nodes are sorted based on their *key*. Any nodes that are "between" context nodes and item nodes are omitted from the output.

(denoted e1, e2,... in the figure), and a key value is computed for each of them, by evaluating the corresponding key expressions. These item nodes are then sorted according to the key values, and output in increasing order of the keys. Notice that the nodes that are below a context but not below an item are omitted from the output.

We illustrate below several examples of xsort.

7.1.1 *Simple Sorting.* We start with a simple example:

```
xsort -c /dblp -e */author -k text().
```

The answer has the following form, listing all author elements in alphabetical order:

```
<dblp>
  <author>...</author>
  <author>...</author>
  . . .
</dblp>
```

7.1.2 *Sorting with Multiple Key Expressions.* The following example illustrates the use of two keys. Assuming that author elements have a firstname and a lastname subelement, it returns a list of all authors, sorted by lastname first, then by firstname:

```
xsort -c /dblp -e */author
      -k lastname/text() -k firstname/text().
```

7.1.3 *Sorting with Multiple Item Expressions.* When multiple -e arguments are present, items are included in the result in the order of the command line. For example the following command:

```
xsort -c /dblp -e article -e inproceedings -e book -e *
```

lists all articles first, then all inproceedings, then all books, then everything else. Within each type of publication the input document order is preserved.

7.1.4 *Sorting at Deeper Contexts.* By choosing contexts other than the root element, we can sort at different depths in the XML document. A common use

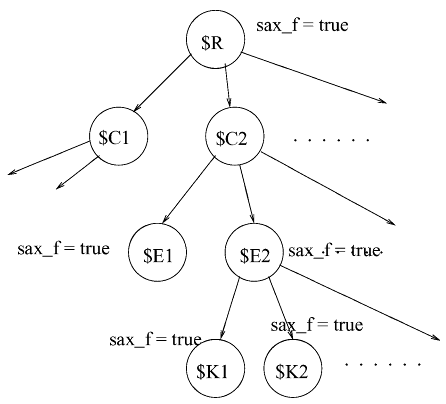


Fig. 17. The query tree generated for the xsort command in Figure 15.

is to normalize the elements by listing their subelements in a standard order. For example, consider:

```
xsort -c /dblp/* -e title -e author -e url -e *.
```

This keeps the order of the publication, but reorders the subelements, as follows: first all `title` elements, then all `author` elements, then all `year` elements, and then everything else.

Notice the use of the "catch all" element `-e *` at the end. We can omit it, and include only selected fields in the result. For example:

```
xsort -c /dblp/* -e title -e author
```

retains only the `title` and `author` in each publication.

The following example sorts authors alphabetically within each publication:

```
xsort -c /dblp/* -e author -k text() -e *.
```

7.1.5 *Sorting with Multiple Context Expressions.* Finally, multiple context arguments can be specified to sort according to different criteria. For example:

```
xsort -c /dblp/book -e publisher -e title -e *
      -c /dblp/* -e title -e *
```

lists `publisher` then `title` first under `books`, and lists `title` first under all other publications.

7.1.6 *Using the XPath Processor.* The XPath expressions in the command line for each tool in the XML Toolkit are converted into a query tree. For illustration, Figure 17 shows the query tree for the `xsort` command. The tree has a root variable, one variable for each context, one variable for each item under each context, and one variable for each key under each item under each context. The `sax_f` flag for all context events is `false`, because we do not need the SAX events that are between contexts and items (these are omitted from the output).

Table I. Experiments with xsort: (a) a Global Sort, and (b) Multiple Local Sorts (Numbers are running times in seconds. A "-" indicates ran out of memory.)

data size (KB)	Xalan (sec)	xsort (sec)
0.41	0.08	0.00
4.91	0.09	0.00
76.22	0.27	0.02
991.79	2.52	0.26
9,671.42	27.45	2.85
100,964.43	—	43.97
1,009,643.71	—	461.36

`xsort -c /dblp -e * -k title/text()`

(a)

data size (KB)	Xalan (sec)	xsort (sec)
0.41	0.08	0.00
4.91	0.10	0.00
76.22	0.29	0.03
991.79	2.78	0.35
9,671.42	29.42	3.54
100,964.43	—	35.52
1,009,643.71	—	358.47

`xsort -c /dblp/* -e title -e author -e year -e *`

(b)

7.1.7 *Implementation.* We briefly describe here the implementation of xsort, which we designed to scale to very large XML files. It sorts one context at a time, copying all other elements (not within a context) to the output file in unchanged order. When sorting one context, it creates a *global key* for each item to be sorted, consisting of the item identification number on the command line, the concatenation of all its keys, and its order number under the current context (to make xsort stable). Next it uses multiway merge-join, with as much main memory as available, and runs for at most two steps. The first step produces the initial runs, using STL's priority queue [ANDIS/ISO 1998], and applying replacement selection [Graefe 1993]. This results in initial runs that may be larger than the main memory: in particular, only one run is produced if the input is already sorted. If more than one run is generated then the second step is executed, which merges all runs to produce the final output. With today's main memories, practically any XML file can be sorted in only two steps. For example, with 128 MB of main memory and disk pages of 4 KB, we can sort XML files of up to 4 TB [Garcia-Molina et al. 2000], and the file size increases quadratically with the memory size. More practical considerations, such as a hard limit of 2 GB on file sizes on most systems, or limits on the number of file descriptors, are more likely to limit the size of the largest file we can sort.

7.1.8 *Experiments.* We evaluated xsort in two experiments,[13] shown in Table I. We compared xsort, with xalan, a publicly available XSLT processor. For xsort, we limited the main memory window to 32 MB. The first represented a global sort which reordered all bibliographic entries: xsort's running time increased linearly, with the exception of an extra factor of 2, when the data size exceeded the memory size. The second table represents local sorts, with small contexts. Here a single pass over the data was always sufficient, and the sorting time increased linearly. The sorting time for xalan also increased linearly, but was an order of magnitude longer than for xsort. Its processing model was DOM-based.

[13]The platform is a Pentium III, 800-MHz, 256-kB cache, 128-MB RAM, 512-MB swap, running RedHat Linux 2.2.18; the compiler is gcc version 2.95.2 with the "-O" command-line option, and Xalan-c 1.3.

8. RELATED WORK

The problem of evaluating large collections of XPath expressions on a stream of XML documents was first introduced in Altinel and Franklin [2000], for a publish-subscribe system called XFilter. Improved techniques have been discussed in XTrie [Chan et al. 2002] (based on a trie), our preliminary version of this work [Green et al. 2003] (based on lazy DFAs), and YFilter [Diao et al. 2003] (based on optimized NFAs). In all methods, except ours, there is a space guarantee that is proportional to the total size of all XPath expressions in the workload, but no guarantee on the throughput. Our method makes the opposite tradeoff.

Two optimizations of the lazy DFA were described in Onizuka [2003]. In one, the XPath expressions are clustered according to their axis types (/ or //) at each depth level. This was shown to reduce the number of DFA states: for example, it was shown that by clustering into eight DFAs, memory usage decreased by a factor of 40 and throughput only by a factor of 8. In the other optimization, NFA tables are allowed to share common subsets, thus saving memory.

More recently, some systems have been described that process more complex XPath expressions [Peng and Chawathe 2003; Gupta and Suciu 2003], or fragments of XQuery [Ludaescher et al. 2002; Diao and Franklin 2003]. A complete XQuery engine for streaming data was described in Florescu et al. [2003].

A related problem is the event detection problem described in Nguyen et al. [2001]. Each event is a set of atomic events, and they trigger queries defined by other sets of events. The technique used here is also a variation on the Trie data structure.

Ives et al. [2002] described a general-purpose XML query processor that, at the lowest level, uses an event based processing model, and showed how such a model can be integrated with a highly optimized XML query processor. We were influenced by Ives et al. [2002] in designing our stream processing model. Query processors like that in Ives et al. [2002] can benefit from an efficient low-level stream processor. Specializing regular expressions with respect to schemas is described in Fernandez and Suciu [1998] and McHugh and Widom [1999].

The conversion problem from regular expression to an NFA was intensively studied in the 1960s and 1970s: see Watson [1993] for a review. The most popular method is due to Thompson [1968] and has been adopted by most textbooks.

Empirical studies of the (eager) DFA construction time have been done in the automaton community [Watson 1996], for NFAs with up to 30 to 50 states.

9. CONCLUSION

We have described two techniques for processing linear XPath expressions on streams of XML packets: using a Deterministic Finite Automaton, and a Stream IndeX (SIX). The main problem with the DFA is that the worst-case memory requirement is exponential in the size of the XPath workload. We have presented a combination of theoretical results and experimental validations that together prove that the size of the *lazy* DFA remains small, for all practical purposes. Some of the theoretical results offer insights into the structure of XPath expressions that is of independent interest. We also validated lazy DFAs on streaming

XML data and showed that they indeed have a very high throughput, which is independent of the number of XPath expressions in the workload. The SIX is a simple technique that adds some small amount of binary data to an XML document, which helps speed up a query processor by several factors. Finally, we described a simple application of these XPath processing techniques: the XML Toolkit, a collection of command-line tools for highly scalable XML data processing.

Electronic Appendix. The Electronic Appendix for this article can be accessed in the ACM Digital Library. The appendix contains the proofs of many theorems from the main body of the article, and a description of several tools in the XML toolkit.

ACKNOWLEDGMENTS

We thank Peter Buneman, AnHai Doan, Zack Ives, and Arnaud Sahuguet for their comments on earlier versions of this article. We also thank Iliana Avila-Campillo, Yana Kadiyska, Jason Kim, and Demian Raven for their contributions to the XML Toolkit, Cristine Haertl for pointing out a bug in the second example in Figure 6 and suggesting the fix, and the anonymous reviewers for their useful comments.

REFERENCES

ABITEBOUL, S., BUNEMAN, P., AND SUCIU, D. 1999. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, San Fransisco, CA.

AHO, A. AND CORASICK, M. 1975. Efficient string matching: an aid to bibliographic search. *Commun. Assoc. Comput. Mach. 18*, 333–340.

ALTINEL, M. AND FRANKLIN, M. 2000. Efficient filtering of XML documents for selective dissemination. In *Proceedings of VLDB* (Cairo, Egipt). 53–64.

ANDIS/ISO. 1998. *C++ Standard*. ANDIS/ISO, Geneva, Switzerland.

AVILA-CAMPILLO, I., GREEN, T. J., GUPTA, A., ONIZUKA, M., RAVEN, D., AND SUCIU, D. 2002. XMLTK: An XML toolkit for scalable XML stream processing. In *Proceedings of PLANX*.

BORNE, K. D. n.d. NASA's astronomical data center. ADC XML resource page. Available online at http://xml.gsfc.nasa.gov/.

BUNEMAN, P., DAVIDSON, S., FERNANDEZ, M., AND SUCIU, D. 1997. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory* (Delphi, Greece). Springer Verlag, Berlin, Germany, 336–350.

BUNEMAN, P., NAQVI, S. A., TANNEN, V., AND WONG, L. 1995. Principles of programming with complex objects and collection types. *Theoret. Comput. Sci. 149*, 1, 3–48.

CHAN, C., FELBER, P., GAROFALAKIS, M., AND RASTOGI, R. 2002. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the International Conference on Data Engineering*.

CHEN, J., DEWITT, D., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM/SIGMOD Conference on Management of Data*. 379–390.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

CORP., M. n.d. DIME—direct Internet message encapsulation specification index page. IETF Internet draft. Available online at `http://msdn.microsoft.com/webservices/understanding/gxa/default.aspx`.

DIAO, Y., ALTINEL, M., FRANKLIN, M., ZHANG, H., AND FISCHER, P. 2003. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst. 28*, 4, 467–516.

DIAO, Y. AND FRANKLIN, M. 2003. Query processing for high-volume XML message brokering. In *Proceedings of VLDB* (Berlin, Germany).

FERNANDEZ, M. AND SUCIU, D. 1998. Optimizing regular path expressions using graph schemas. In *Proceedings of the International Conference on Data Engineering*. 14–23.

FLORESCU, D., HILLARY, C., KOSSMANN, D., P.LUCAS, RICCARDI, F., WESTMANN, T., CAREY, M., SUNDARARAJAN, A., AND AGRAWAL, G. 2003. The bea/xqrl streaming xquery processor. In *Proceedings of VLDB* (Berlin, Germany). 997–1008.

GARCIA-MOLINA, H., ULLMAN, J. D., AND WIDOM, J. 2000. *Database System Implementation*. Prentice Hall, Upper Saddle River, NJ.

GOLDMAN, R. AND WIDOM, J. 1997. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of Very Large Data Bases*. 436–445.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv. 25*, 2 (June), 73–170.

GREEN, T. J., MIKLAU, G., ONIZUKA, M., AND SUCIU, D. 2003. Processing XML streams with deterministic automata. In *Proceedings of ICDT*. 173–189.

GUPTA, A. K., HALEVY, A. Y., AND SUCIU, D. 2002. View selection for XML stream processing. In *Proceedings of the International Workshop on the Web and Database* (Web DB). 83–88.

GUPTA, A. AND SUCIU, D. 2003. Stream processing of XPath queries with predicates. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

GUPTA, A., SUCIU, D., AND HALEVY, A. 2003. The view selection problem for XML content based routing. In *Proceedings of the PODS*.

HIGGINS, D. G., FUCHS, R., STOEHR, P. J., AND CAMERON, G. N. 1992. The EMBL data library. *Nucleic Acids Res. 20*, 2071–2074.

HOPCROFT, J. AND ULLMAN, J. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

IVES, Z., HALEVY, A., AND WELD, D. 2002. An XML query engine for network-bound data. *VLDB J. 11*, 4, 380–402.

LAURIKARI, V. 2000. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Proceedings of SPIRE*. 181–187.

LEY, M. n.d. Computer science bibliography (dblp). Available online at `http://dblp.uni-trier.de`.

LIEFKE, H. AND SUCIU, D. 2000. XMill: An efficent compressor for XML data. In *Proceedings of SIGMOD* (Dallas, TX). 153–164.

LUDAESCHER, B., MUKHOPADHYAY, P., AND PAPAKONSTANTINOU, Y. 2002. A transducer-based XML query processor. In *Proceedings of VLDB*. 227–238.

MARCUS, M., SANTORINI, B., AND MARCINKIEWICZ, M. A. 1993. Building a large annotated corpus of English: The Penn Treenbak. *Computat. Ling. 19*.

MCHUGH, J. AND WIDOM, J. 1999. Query optimization for XML. In *Proceedings of VLDB* (Edinburgh, U.K.). 315–326.

NGUYEN, B., ABITEBOUL, S., COBENA, G., AND PREDA, M. 2001. Monitoring XML data on the Web. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (Santa Barbara, CA). 437–448.

ONIZUKA, M. 2003. Light-weight xpath processing of XML stream with deterministic automata. In *Proceedings of the CIKM*. 342–349.

PENG, F. AND CHAWATHE, S. 2003. XPath queries on streaming data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 431–442.

ROZENBERG, G. AND SALOMAA, A. 1997. *Handbook of Formal Languages*. Springer Verlag, Berlin, Germany.

SAHUGUET, A. 2000. Everything you ever wanted to know about DTDs, but were afraid to ask. In *Proceedings of WebDB*, D. Suciu and G. Vossen, Eds. Sringer Verlag, Berlin, Germany, 171–183.

SNOEREN, A., CONLEY, K., AND GIFFORD, D. 2001. Mesh-based content routing using XML. In *Proceedings of the 18th Symposium on Operating Systems Principles*.

THIERRY-MIEG, J. AND DURBIN, R. 1992. Syntactic Definitions for the ACEDB Data Base Manager. Tech. rep. MRC-LMB xx.92. MRC Laboratory for Molecular Biology, Cambridge, U.K.

THOMPSON, K. 1968. Regular expression search algorithm. *Commun. Assoc. Comput. Mach. 11*, 6, 419–422.

WATSON, B. W. 1993. A taxonomy of finite automata construction algorithms. Computing Science report 93/43. University of Technology Eindhoven, Eindhoven, The Netherlands.

WATSON, B. W. 1996. Implementing and using finite automata toolkits. *J. Nat. Lang. Eng. 2*, 4 (Dec.), 295–302.

Received July 2003; revised May 2004; accepted August 2004