From Searching Text to Querying XML Streams

Dan Suciu

University of Washington, USA
suciu@cs.washington.edu
http://www.cs.washington.edu/homes/suciu

Abstract. XML data is queried with XPath expressions, which are a limited form of regular expressions. New XML stream processing applications, such as content-based routing or selective dissemination of information, require thousands or millions of XPath expressions to be evaluated simultaneously on the incoming XML stream at a high, sustained rate. Conceptually, the XPath evaluation problem is analogous to the text search problem, in which one or several regular expressions need to be matched to a given text, but the number of regular expressions here is much larger, while the "text" is much shorter, since it corresponds to the depth of the XML stream. In this paper we examine techniques that have been proposed for XML stream processing, which are variations of either a non-deterministic or a deterministic finite automata (NFA and DFA). For the latter, we describe a series or theoretical results establishing lower and upper bounds on the number of DFA states for sets of XPath expressions.

1 Databases, Text, and XML

Data in relational databases is *structured*. It has a schema, which is usually stored in a part of the of the database called the catalog, while the data values are stored separately, in tables, following a layout that is completely described by the schema. User queries, expressed in SQL, refer both to the schema components, such as relation names and their attributes, and to the data values, in the form of equality predicates, inequality predicates, or string matches. Research on query processing has focused on join processing techniques, join ordering, and indexes.

Text documents are *unstructured*. There is no schema, only the text, and the data consists of some large collection of documents. A query consists of a regular expression, often as simple as a single word, and the answer consists of the set of text documents that match the given query. Indexes are used here too, and they are conceptually similar to, although technically different from those in relational databases (e.g. inverted files v.s. B^+-trees). Research on query processing has focused, among other things, on how to process efficiently regular expressions on a text document, and has produced celebrated results such as Knuth-Morris-Pratt's string search algorithm, suffix trees, and suffix arrays. These techniques have often been based on, and even expanded automata theory.

A new kind of data is *semistructured data*. Although considered in one form or another for a long time, semistructured has gained main-stream acceptance only recently, since the introduction of XML. Like in structured data here we have schema components (the tags and attributes in XML), and data values are organized along these components. But here the schema is embedded with the data, thus allowing each data item to describe its own local schema. This allows much more freedom in designing the structure, and often leads to structures that were explicitly disallowed in the relational data, such as nested collections, multiple or missing subelements, elements of the same type but with different structures, heterogeneous collections, etc. Hence the term *semistructured*. In the past, researchers have studied instances of data that we would call today semistructured, either in the form of SGML documents, or as *structured documents*, i.e. documents with a predefined grammar. Many interesting research results have been produced in this context, and they are definitely relevant today [9, 8, 5, 24, 20]. What is different today are the new applications in which XML is being used, which create new challenges for efficient query processing.

An Application of XML Stream Processing For illustration, consider XML Routing [25, 12, 13]. Here a network of *XML routers* forwards a continuous stream of XML packets from data producers to consumers. The "packet" is really an instance of a semistructured data, only expressed in XML. Each router in the network receives incoming XML packets, and forwards each packet to a subset of its output links (other routers or clients). In order to determine where to forward a given packet, the router needs to evaluate a large number of XPath filters on that packet, which usually correspond to clients' subscription queries, on the stream of XML packets. For example: "if the packet satisfies the expression:

```
/Envelope[Header/@dest="Lisabon"][@priority>100]/Body//*[@keyword="SPIRE"]
```

then forward the packet to servers S_{64} and S_{108}".

Data processing in XML packet routing 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. However the performance requirements are high, because routers often need to process packets at the rate of the network, for example one may want to process XML packets at a rate of, say, 10MB/s. In one experimental system [25] publicly available tools were used to parse the XML documents and evaluate the XPath expressions, and the resulting performance was quite poor: about 2.6KB/s for 10,000 XPath expressions[1].

The XPah expressions used here are very similar to queries considered before, in the context of structured text processing. For example all XPath expressions in this paper can be translated into the PAT algebra [23], for which efficient processing techniques are known. So, one may ask: what is new here ?

The answer lies in the different assumptions on the data and the queries. First we need to compute on a *stream* of incoming XML packets. Stream data processing is only now emerging as a mainstream research topic [3]. One issue in stream data is that we cannot use an index, at least not in the traditional sense,

[1] The authors in [25] report parsing a 262 bytes XML document in $64.2\mu s$, and evaluating one short XPath expression in 10 μs. This translates into a throughput of $262/100064.2 = 2.6$KB/s for 10,000 XPath expressions.

which makes some of the most efficient processing techniques for PAT algebra expressions useless. The second is the scale of the queries. Although each predicate is simple, like the one above, there may be lots of them, perhaps thousands or millions. How can we evaluate 10^6 XPath expressions on an incoming XML stream, and do this at, say, 10MB/s ? Finally, XML stream applications often have additional knowledge about the data or the queries that we did not have in processing structured text, and which we need to exploit to improve query processing. For example we may know the schema of the XML data, or we may know certain selectivities for some of the predicates used in the XPath expressions and we could use them to do cost-based optimizations like in relational database systems.

This paper defines the XML stream processing problem, and describes some of the techniques that have been considered in this context, with a strong bias toward the author's own work, in the context of the XML Toolkit project [11, 10, 14].

Outline The XML stream processing problem is described and discussed in Sec. 2. Two of the existing processing techniques, based on NFAs and DFAs respectively, are discussed in Sec. 3. An analysis of the number of states in the DFA in presented in Sec. 4. The lazy DFA is described, analyzed, and evaluated in Sec. 5. We conclude in Sec. 6.

2 XML Stream Processing

We start by describing the architecture of an XML stream processing system [10], to illustrate the context in which XPath expressions are processed. This defines a simple event-based XML processing model, extending the well-known SAX parsing model[2]. Several correlated XPath expressions are arranged in a tree, called the *query tree*. An input XML stream is first parsed by a SAX parser that generates a stream of *SAX events* (Fig. 1); this is input to the query processor that evaluates the XPath expressions and generates a stream of *application events*. The application is notified of these events, and usually takes some action such as forwarding the packet, notifying a client, or computing some values. In [10], an optional Stream Index (called SIX) may accompany the XML stream to speed up processing [10].

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

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

Here E, A, and S are an element constant, an attribute constant, and a string constant respectively, and $*$ is the wild card. The function `text(S)` matches a

[2] Simple API for XML, `http://www.megginson.com/SAX/`.

text node whose value is the string S. In addition to the query tree, the application specifies the variables for which it requests SAX events. There is a distinguished variable, $\$R$, which is always bound to the root.

Example 1. The following is a query tree[3]:

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

Fig. 2 shows this query tree graphically, where the SAX events are requested for $T and $TH. Fig. 3 shows the result of evaluating this query tree on an XML input stream: the first column shows the XML stream, the second shows the SAX events generated by the parser, and the last two columns show the application events. Only some of the SAX events are forwarded to the application, namely exactly those that occur within a $T or $TH variable event.

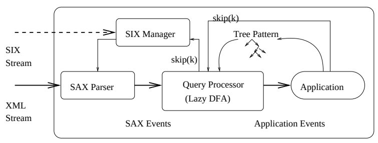


Fig. 1. System's Architecture

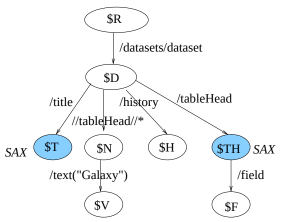


Fig. 2. A Query Tree

XML Stream	Parser SAX Events	Application Events: Variable Events	Application Events: SAX Events
<datasets>	start(datasets)	start($R)	
<dataset>	start(dataset)	start($D)	
<history>	start(history)	start($H)	
<date>	start(date)		
5/1/59	text("5/1/59")		
</date>	end(date)		
</history>	end(history)	end($H)	
<title>	start(title)	start($T)	
			start(title)
<subtitle>	start(subtitle)		start(subtitle)
Study	text(Study)		text(Study)
</subtitle>	end(subtitle)		end(subtitle)
</title>	end(title)		end(title)
		end($T)	
...			
</dataset>	end(dataset)	end($D)	
...	...		
</datasets>	end(datasets)	end($R)	

Fig. 3. Events generated by a Query Tree

Filters XPath expressions are rarely linear, but instead have branches called *filters*. It is possible to implement filters on top of query trees, using a simple and direct method [11], whose throughput decreases with the number of filters in the query tree. In order to support filters efficiently one needs to combine

[3] The tags are from an XML astronomical database [19].

the techniques for linear XPath expressions with event detection methods, e.g. like those described in [21]. We will not further discuss filters here, and assume instead that every XPath expressions is linear. Notice, however, that XPath expressions can already be arranged in a tree.

The XML Stream Processing Problem The problem that we address is: given a query tree Q, pre-process it, then evaluate it on an incoming XML stream. The goal is to maximize the throughput at which we can process the XML stream. A special case of a query tree, Q, is one in which every node is either the root or a leaf node, i.e. has the form: $\mathtt{\$X_1}$ in $\mathtt{\$R/e_1}, \mathtt{\$X_2}$ in $\mathtt{\$R/e_2}, \ldots, \mathtt{\$X_p}$ in $\mathtt{\$R/e_p}$ (each $\mathtt{e_i}$ may start with // instead of /): we call Q a *query set*, or simply a *set*. Query sets are typical of many applications (e.g. XML routing), and are convenient for analyzing the problem as a function of their cardinality, p. It actually suffices to restrict the theoretical analysis to query sets only, since every query tree Q can be converted into an equivalent query set Q'. An implementation does not have to do that, but it suffices for us to restrict the theoretical analysis in Sec. 4 to query sets.

3 XML Stream Processing Techniques

Current approaches to processing sets of XPath expressions on an XML stream are based either on a nondeterministic finite automaton (NFA) or on a deterministic finite automaton (DFA). We review here both of them.

3.1 Processing with NFAs

Recall that the query tree may be a very large collection of XPath expressions. The first step is to translate the XPath expressions into one, single Nondeterministic Finite Automaton, denoted A_n. The automaton can then be executed directly on the input XML stream, using a stack of sets of NFA states. As usual we will denote $\delta(q, a)$ the transition function returning a set of states, i.e. $q' \in \delta(q, a)$ if there exists a transition labeled a from q to q'. We also extend δ to words, by defining $\delta(q, wa) = \delta(\delta(q, w), a)$. Notice that a here may be a tag, attribute label, string, or wild-card ($*$).

To illustrate, consider the query tree P shown in Fig. 4(a). The NFA, denoted A_n, is shown in Fig. 4(b). Transitions labeled $*$ correspond to $*$ or // in P; there is one initial state; there is one terminal state for each variable (`$X`, `$Y`, ...); and there are ε-transitions [4]. It is straightforward to generalize this to any query tree. Importantly, the number of states in A_n is proportional to the size of P.

To execute the NFA on the input XML stream we maintain a set of current states, S, and a stack of sets of states, ST. On a `start(`a`)` SAX event, with tag a, we push S on ST and replace S with the set of all successor states: $\{q' \mid q' \in \delta(q, a), q \in S\}$, where δ is the transition function. The transition function

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

is implemented as a hash table at each state. On an `end(a)` SAX event we pop one set from the stack, and that becomes the current S.

Let us analyze the time and space complexity of the NFA evaluation method. The space of the NFA is proportional to the size of P. In addition we need a stack, whose depth is that of the XML input document, i.e. can be assumed to be bound by a constant, for all practical purposes. The elements on the stack are sets of states, whose cardinality is bounded by the size of P. Hence the space complexity is linear in the size of P. Consider now the time complexity. The time to process one SAX events can be as large as the number of states in A_n, hence is linear in the size of P. As a consequence, the more XPath expressions we have, the slower we can process the input XML stream.

Application: XFilter The XFilter system [2] is designed for efficient processing of XPath expressions on XML streams, with applications in Selective Dissemination of Information. We explain XFilter here putting emphasis on its relationship to NFAs, and refer the reader to [2] for details. More precisely, we will describe it as a sequence of optimization steps over the direct NFA evaluation method. The first step is to replace δ with a new transition function δ' defined by: $\delta'(q,a) = \{q' \mid \exists k.k \geq 0 \land q' \in \delta(q, *^k.a)\}$. That is, multiple $*$ transitions are collapsed, and, in the process, some states may be eliminated. Due to the special form of the linear XPath expressions, for every state q' there exists now a unique "predecessor" state q s.t. $q' \in \delta'(q,a)$ for some symbol a, and we denote $q = \delta^{-1}(q')$. Also define a "relative position" predicate for each state q', describing the number of symbols needed to reach q' from $\delta^{-1}(q')$ (the k in the definition of δ', plus one). The predicate can be either $= n$, or $\geq n$, for some $n \geq 1$. To illustrate δ', consider the the NFA in Fig. 4 (b). We transform into a new NFA having only the states $1, 2, 5, 7, 10$; the new transition function is $\delta'(1,a) = \{2\}$, $\delta'(2,b) = \{5,7\}$, $\delta'(7,d) = \{10\}$, while the distance predicates for the states $2, 5, 7, 10$ are $= 1, \geq 2, = 1, = 2$ respectively (i.e. we can go from state 1 to state 2 by following exactly one symbol, we can go from state 2 to state 5 by following ≥ 2 symbols, etc). When applied to query sets rather than query trees, each state q has a unique transition, i.e. there is a unique a for which $\delta'(q,a) \neq \emptyset$, and, moreover, $\delta'(q,a)$ contains exactly one state, q'. Hence, we can drop the a argument from δ', and simply write $q' = \delta'(q)$.

The next optimization in XFilter is the way in which it represents at runtime the current set of states and the stack of sets of states, which we denoted S and ST respectively. The basic idea is to store in S and ST the values $\delta'(q)$ rather than q: this, as will see, speeds up execution at runtime. More precisely, XFilter maintains one hash table indexed by the symbols a (tags and attribute labels) occurring in all the XPath expressions. The hash table entry for a symbol a is a list, called *candidate list*, CL, which is updated at runtime as follows. The CL corresponding to a contains, at any given moment, all pairs $(\delta'(q,a), D)$, where q is any NFA state that has been visited by the current sequence of open XML tags, and D is a depth predicate of the form $= d$ or $\geq d$, representing the depth where a has to occur in order for the NFA to make the transition $q \rightarrow \delta'(q,a)$: the depth is computed by adding the depth at which q was reached to the relative position predicate for q'.

At runtime XFilter works as follows. On a `start(`a`)` SAX event, occurring at depth d, the CL for entry a is inspected, and for each pair (q', D), the predicate D is checked on d: if it is true, then we compute the successor of q', say $\delta'(q', b)$, and insert the pair (q'', D') in the CL at entry b. Thus, while processing an a event, we may need to update the CL's at several entries b. However, only the states at entry a need to be inspected, and this is an improvement over the direct NFA evaluation method. To see this, recall that in the direct NFA processing we have a set of states S and need to compute $\{q' \mid q' \in \delta'(q', a), q \in S\}$. When iterating q over S, many states may not have an a transition at all. For example, given the XPath expressions $/b/a$, $/b/c$, $/b/d$, $/b/e$, after seeing a `start(`b`)` event S has three states. If `start(`a`)` is the next event, then the direct NFA evaluation method needs to scan over all three states in S, and discover that only one has an a transition, while XFilter checks directly the CL at entry a, which has a single state. An `end(`a`)` event is processed similarly only in reverse, i.e. we need to find all matching elements in the CL for entry a, and remove the appropriate $\delta'(q', b)$ elements.

Another technique, XTrie [4], introduces additional optimization that further increase the throughput by a factor of four or so.

3.2 Processing with DFAs

Alternatively, we can evaluate the XPath expressions by constructing a DFA from the NFA. We review this construction here briefly and refer the reader to a standard textbook [15] for more details.

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, i.e. $\delta(q_0, w)$ where q_0 is the initial state in A_n. In our example (Fig. 4 (b)) we have $\Sigma = \{a, b, d, \omega\}$, and $A_n(\varepsilon) = \{1\}$, $A_n(a.b) = \{3, 4, 7\}$, $A_n(a.\omega) = \{3, 4\}$, $A_n(b) = \emptyset$.

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

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

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

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

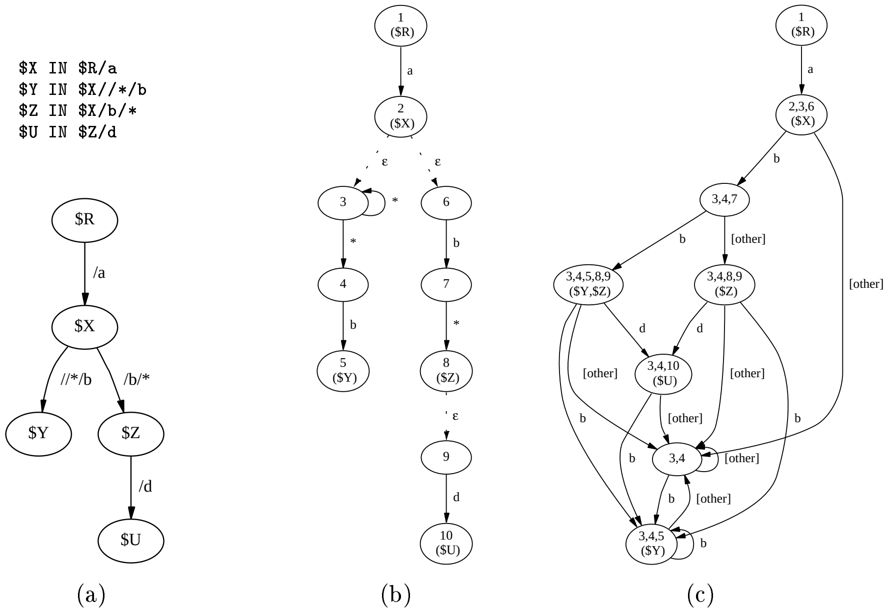


Fig. 4. (a) A query tree; (b) its NFA, A_n, and (c) its DFA, A_d.

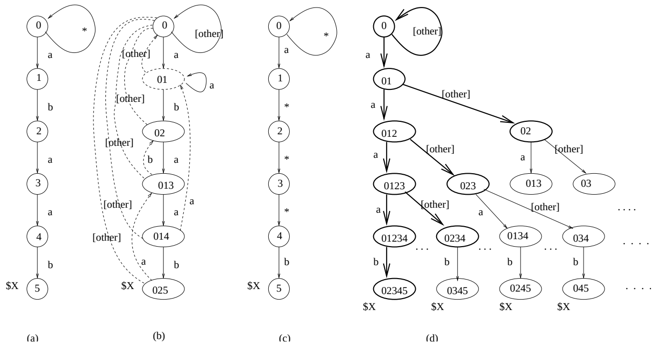
Processing an XML stream with a DFA is similar to that of an NFA, only simpler, since now we have a single current state, and the stack contains one single state per entry. Both a `start(element)` and a `end(element)` take now $O(1)$ time to process, and, assuming an upper bound on the depth of the XML document, we can preallocate the stack statically and there is no need to do any dynamic memory management. The space taken by the DFA, however, can be prohibitive since, in general, the number of states in a DFA can be exponentially larger that that in the NFA. The solution, as we will see, is to compute the DFA lazily.

Application: the XML Toolkit The XML Toolkit [10] is an application using a lazy DFA to process large collections of XPath expressions on an input XML stream. The toolkit, available from `xmltk.sourceforge.net`, implements an API to a processor for pattern trees, as described in Sec. 3. In addition it implements a number of XML Unix commands that scale to large XML files, like `xsort` (for sorting an XML document), `xagg` (for aggregation), `xnest`, `xflatten`.

4 Analyzing the Size of the DFA

While the size of a DFA for a general regular expression may be exponential [15], this general statement needs to be revisited in our context for two reasons. First the regular expressions we are considering are restricted to XPath expressions given by the grammar in Sec. 2. Second, in the XML applications that we target the regular expressions are over the structure of an XML stream, and, hence they are essentially evaluated on short sequences (whose length is bound by the depth of the XML tree). For example it can be in the range 20-30. By contrast, regular expressions in text processing need to inspect large documents, of at least a few thousands of characters.

Upper and lower bounds on the number of states in the DFA for XPath expressions were given in [11]. We review here those results.



Parameter	Meaning
k	number of //'s
$s = \mid \Sigma \mid$	alphabet size
n_i	length of N_i, $i = 0, \ldots, k$
$n = \sum_{i=0,k} n_i$	length of P
m	max # of $*$'s in each p_i

(e)

Fig. 5. The NFA (a) and the DFA (b) for `//a/b/a/a/b`. The NFA (c) and the DFA (with back edges removed) (d) for `//a/*/*/*/b`: here the eager DFA has $2^5 = 32$ states, while the lazy DFA, assuming the DTD `<!ELEMENT a (a*|b)>`, has at most 9 states. Parameters for a linear XPath expression (e): $P = p_0//p_1//\ldots//p_k$, with $p_i = N_1/N_2/\ldots/N_{n_i}$, $i = 0, \ldots, k$.

Single XPath Expression We start with one linear XPath expression, which has the form $P = p_0//p_1//\ldots//p_k$ where each p_i is $N_1/N_2/\ldots/N_{n_i}$, $i = 0, \ldots, k$, and each N_j is given by (1). This case has already been considered in traditional text processing. For example, consider $P = //a/b/a/a/b$ the NFA and DFA are shown in Fig. 5 (a) and (b), and they have the same number of states. The DFA corresponds to Knuth-Morris-Pratt's string matching algorithm [6] applied to the word *abaab*. However, the construction in the KMP algorithm does not extend to all linear XPath expressions, and in general we may have some exponential blowup:

Theorem 1 **([11]).** *Given a linear XPath expression P with the parameters in Fig. 5 (e), define:*

$$\mathit{prefix}(P) = n_0$$
$$\mathit{suffix}(P) = k + k(n - n_0)s^m$$

Then the eager DFA for P has at most prefix(P)+ suffix(P) states. In particular, when $k = 0$ the DFA has at most n states, and when $k > 0$ the DFA has at most $k + k(n - n_0)s^m$ states.

When no wild-cards occur ($m = 0$), then the number of states in the DFA is at most $k + k(n - n_0)$. When wild cards are present ($m > 0$), then the nubmer of states in the DFA can be exponentially larger than that of the NFA's This is illustrated in Fig. 5 (c), (d) showing that the DFA for $p = //a/*/*/*/b$, has 2^5 states. It generalizes to the DFA for $//a/*/\ldots/*/b$ having 2^{m+2} states, where m is the number of $*$'s.

Thus, for the case of only one linear XPath expression, the number of wild cards between two consecutive occurrences of // is the only source of exponential size for the eager DFA of one linear XPath expression. One expects this number to be small in most practical applications; arguably users are more likely to write expressions like `/catalog//product//color` rather than expressions like `/catalog//product/*/*/*/*/*/*/*/*/*/color`. Indeed, some implementations of XQuery already translate a *single* linear XPath expression into DFAs [16].

Multiple XPath Expressions The case when P consists of several XPath expressions also has an analogy in text processing: it is related to Aho and Corasick's dictionary matching problem [1, 22]. We are given a dictionary consisting of p words, $\{w_1, \ldots, w_p\}$, and have to retrieve all their occurrences in a given text. This translates into constructing the DFA for the set of XPath expressions $Q = \{//w_1, \ldots, //w_p\}$. Hence, the dictionary matching problem corresponds to the 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. However, when we allow allow multiple occurrences of //, as we need in XML applications, the number of DFA states grows exponentially in the size of P, as illustrated in the following example.

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

It is easy to generalize this exaple, and prove that p XPath expressions my have a DFA with 2^p states [11]. In terms of an upper bound, we have:

Theorem 2 ([11]). *Let Q be a set of XPath expressions. Then the number of states in the eager DFA for Q is at most:*

$$\sum_{P \in Q} (\mathit{prefix}(P)) + \prod_{P \in Q} (1 + \mathit{suffix}(P))$$

In particular, if A, B are constants s.t. $\forall P \in Q$, prefix$(P) \leq A$ and suffix$(P) \leq B$, then the number of states in the eager DFA is $\leq p \times A + B^{p'}$, where p' is the number of XPath expressions $P \in Q$ that contain //.

Recall that *suffix*(P) already contains an exponent, which we argued is small in practice. The theorem shows that the extra exponent added by having multiple XPath expressions is precisely the number of expressions with //'s.

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

5 Processing XML Streams with Lazy DFAs

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

This idea has been used before in text processing, but it has never been applied to such large number of expressions as required in XML stream processing (say, 100,000): a careful analysis of the size of the lazy DFA is needed to justify its feasibility. Unlike our analysis of the eager DFA, in Sec. 4, when analyzing the lazy DFA we need to take into account both the set of XPath expressions, and the XML data instance. Two results have been shown in [11], which apply to two different classes of XML data.

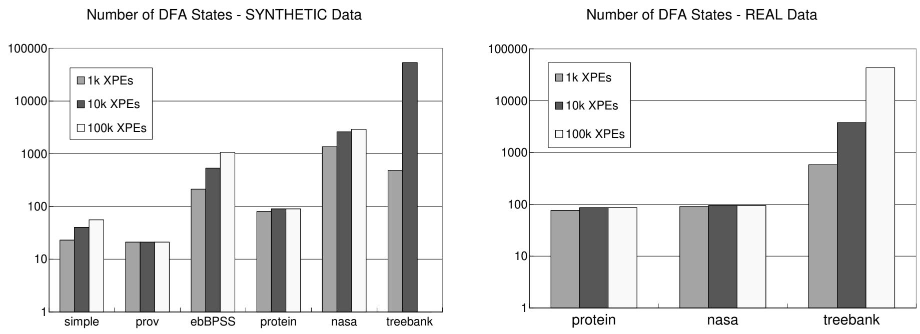
Data-centric XML Documents Call a DTD *simple* if any two distinct cycles in the DTD graph have disjoint sets of elements. For example non-recurisve DTDs are simple, because they have no cycles at all. Data-centric XML instances are either non-recursive, or have a limited form of recursion that usually corresponds to a hierarchy (part-subpart, or unit-subunit, etc). Thus, they have self-recursive elements, where an element a may contain a subelement a, but not mutual recursion, where a may contain b and b may contain a. These are also simple DTDs, since all cycles contain a single element and, hence, distinct ccyles are disjoint. Here we have:

Theorem 3 ([11]). *Let D be the number of simple paths in a simple DTD, and let d be the maximum number cycles on any simple path. Let Q be a set of XPath expressions of maximum depth n. Then the lazy DFA has at most $1 + D \times (1+n)^d$ states.*

Document-oriented XML instances encoding document-centric information, however, often have non-simple DTDs: for example a `table` may contain a `table` or a `footnote`, and a `footnote` may also contain a `table` or a `footnote`. In such cases one can give an upper bound on the size of the lazy DFA in terms of *Data Guides* [7]. For a given XML instance, the data guide G is defined to be the trie of all root-to-leaf element sequences. An empirical observation is that real

XML data instances tends to have small data guides, regardless of its DTD. For example users occasionally place a `footnote` within a `table`, or vice versa, but don't nest all available element in all possible combinations. All XML data instances described in [17] have very small data guides, except for Treebank [18], where the data guide has $G = 340,000$ nodes. This is useful for the lazy DFA because of the following:

Corollary 1 ([11]). *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.*



(a) (b)

Fig. 6. Size of the lazy DFA for (a) synthetic data, and (b) real data. 1k means 1000 XPath expressions. For 100k XPath exrpession for the `treebank` DTD with synthetic data we ran out of memory.

5.1 Validation of the Size of the Lazy DFA

An experimental evaluation of the size of the lazy DFA was conducted in [11]. We briefly review here those findings on six DTDs, of which four are simple (`protein.dtd` is even non-recursive), and two are not. Three sets of queries of depth $n = 20$ were generated, with 1,000, 10,000, and 100,000 XPath expressions[6], with 5% probabilities for both the $*$ and the //. Next, synthetic data was generated for all six DTDs[7], and the three query sets were executed on the data. The number of states in the lazy DFA is shown in Fig. 6 (a). For three DTDs (`protein`, `nasa`, and `treebank`) real XML data was also availble: the number of states in the lazy DFA on the real data is shown in Fig. 6 (b).

[6] With the generator described in [2].

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

The first four DTDs in Fig. 6 (a) are simple, and indeed the number of states in the lazy DFA is small. The last two DTDs are non-simple, and the number of states was significantly larger. In fact, we ran out of memory on `treebank` while processing the largest query set (100,000 XPath expressions).

By contrast Fig. 6 (b) shows that the number of states in the lazy DFA is significantly smaller when ran on *real* XML data, rather than synthetic one. The explanation is that real XML data tends to have a relatively small dataguide, hence Corollary 1 applies, while synthetic data has (by definition) a large data guide.

5.2 The Throughput of Lazy DFAs

As expected, at runtime the DFA outperforms any NFA-based approach. The graph in Figure 7, taken from [11], illustrates the point. It shows the measured throughput, as a function of the amount of data consumed, for the lazy DFA, for XFilter [2] (without list balancing), and for the parser only. The data is the NASA XML dataset [19], and is about 25MB. The four sets of XPath expressions were synthetically generated, and contain 1k (= 1000), 10k, 100k, and 1000k XPath expressions respetively. In all cases, the probability of both a $*$ and a // is 10%. After a "warm-up" period, when most of the lazy DFA states are being constructed, the throughput of the lazy DFA stabilizes at about 5.4MB/s. This is about 1/2 the parser's througput. Importantly, the stable throughput is independent on the number of XPath expressions in the query set. By constrast, XFilter does not need a warm-up phase, however the throughput degrades as we increase the number of XPath expressions, until it is about 50,000 smaller than that of the lazy DFA.

6 Conclusions

XML stream processing provides a new class of applications for automata-based processing techniques. Unlike traditional text processing applications, which have to search words or regular expressions in long text documents, here the match is on the XML's structure, and usually applies to a short sequence of elements, bounded by the detph of the XML document. However, the number of expressions that we need to evaluate on the input XML stream is much larger than what traditional text-based processing has previously attempted to process.

This paper has argued that lazy Deterministic Finite Automata are the most efficient technique for processing sets of *linear* XPath expressions, and that they work on *most* XML data streams. Both emphasis highlight limitations, and point to future work. First, the techniques described here apply only to linear XPath expression. In practice one encounters almost always expressions with filters. For example, consider `/hr/person[name/text()="John"][age/text()="48"]`. The lazy DFA can detect both branches `/hr/person/name/text("John")` and `/hr/person/age/text("48")`, but we need to identify persons that match *both* branches. While processing linear XPath expression is a neceessary first step,

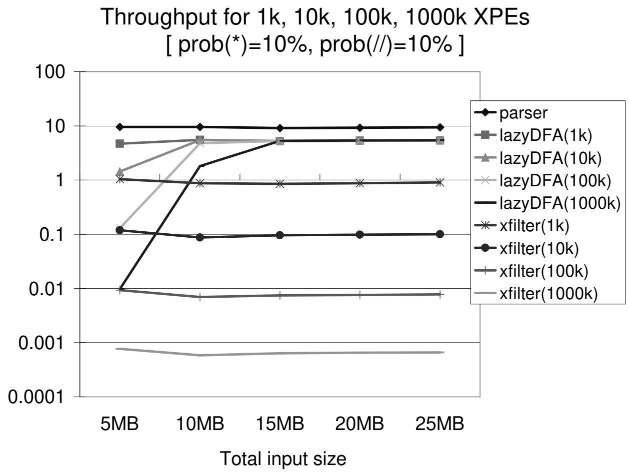


Fig. 7. Experiments illustrating the throughput of the DFA v.s. XFilter [2], as a function of the amount of XML data consumed. The number of XPath expressions varies from 1k (1000), to 1000k ($= 10^6$).

it is insufficient yet for practical purposes. The challenge is to extend the lazy DFA idea to branches, and obtain similarly high throughputs. Second, despite the positive results in Theorem 3 and Corollary 1, they fall short of offering a hard upper bound guarantee on the number of states in the lazy DFA in all cases. Production systems need hard guarantees, and they can only be obtained by offering a safety valve that adds an NFA-based proceesing component to the lazy DFAs. Designing, optimizing, and evaluating such mixed methods is part of future work.

Acknowledgment Most of the results mentioned in this paper have been obtained in work done in collaboration with T.J. Green, A. Gupta, G. Miklau, and M. Onizuka. The author was partially supported by the NSF CAREER Grant 0092955, a gift from Microsoft, and an Alfred P. Sloan Research Fellowship.

References

1. A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.
2. M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination. In *Proceedings of VLDB*, pages 53–64, Cairo, Egipt, September 2000.
3. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the ACM SIGART/SIGMOD Symposium on Principles of Database Systems*, pages 1–16, June 2002.
4. C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the International Conference on Data Engineering*, 2002.

5. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In R. Snodgrass and M. Winslett, editors, *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, pages 313–324, Minneapolis, Minnesota, May 1994.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.
7. R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of Very Large Data Bases*, pages 436–445, September 1997.
8. G. Gonnet, R. Baeza-Yates, and T. Snider. Lexicographical indices for text: inverted files vs. PAT trees. In W. B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice-Hall, 1992.
9. G. Gonnet and F. Tompa. Mind your grammar: A new approach to modelling text. In *Proceedings of 13th International Conference on Very Large Databases*, pages 339–346, 1987.
10. T. J. Green, A. Gupta, M. Onizuka, and D. Suciu. XMLTK: an XML toolkit for scalable XML stream processing, 2002. manuscript.
11. T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes, 2002. manuscript.
12. M. Gudgin, M. Hadley, J. Moreau, and H. Nielsen. SOAP version 1.2 part 1: Messaging framework, 2001. available from the W3C, `http://www.w3.org/2000/xp/Group/`.
13. M. Gudgin, M. Hadley, J. Moreau, and H. Nielsen. SOAP version 1.2 part 2: Adjuncts, 2001. available from the W3C, `http://www.w3.org/2000/xp/Group/`.
14. A. Gupta, A. Halevy, and D. Suciu. View selection for XML stream processing. In *WebDB'2000*, 2002.
15. J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.
16. Z. Ives, A. Halevy, and D. Weld. An XML query engine for network-bound data. Unpublished, 2001.
17. H. Liefke and D. Suciu. XMill: an efficent compressor for XML data. In *Proceedings of SIGMOD*, pages 153–164, Dallas, TX, 2000.
18. M. Marcus, B. Santorini, and M.A.Marcinkiewicz. Building a large annotated corpus of English: the Penn Treenbak. *Computational Linguistics*, 19, 1993.
19. NASA's astronomical data center. ADC XML resource page. http://xml.gsfc.nasa.gov/.
20. G. Navarro and R. Baeza-Yates. Proximal nodes: a model to query document databases by content and structure. *ACM Transactions on Information Systems*, 15(4):400–435, October 1997.
21. B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 437–448, Santa Barbara, 2001.
22. G. Rozenberg and A. Salomaa. *Handbook of Formal Languages.* Springer Verlag, 1997.
23. A. Salminen and F. W. Tompa. PAT expressions: An algebra for text search. In *Papers in Computational Lexicography: COMPLEX'92*, pages 309–332, 1992.
24. A. Salminen and F. W. Tompa. PAT expressions: An algebra for text search. *Acta Linguistica Hungarica*, 41(1-4):277–306, 1994.
25. A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using XML. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.