ELSEVIER

Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Journal of Discrete Algorithms 2 (2004) 17–32

JOURNAL OF DISCRETE ALGORITHMS

www.elsevier.com/locate/jda

From searching text to querying XML streams

Dan Suciu

University of Washington, Department of Computer Science, 114 Sieg Hall, Box 352350, Seattle, WA 98195, USA

Abstract

XML data is queried with a limited form of regular expressions, in a language called XPath. 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. In its simplest approximation, 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. At a finer level, it is related to the tree pattern matching problem. However, unlike the traditional setting, 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 and describe a few open problems.

Keywords: XML; XPath; Automata theory; Query languages; Data streams

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

E-mail address: suciu@cs.washington.edu (D. Suciu).
URL: http://www.cs.washington.edu/homes/suciu.

1570-8667/$ – see front matter
doi:10.1016/S1570-8667(03)00063-7

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 vs. 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 data 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 and data are stored *together*, allowing each data item to describe its own local schema. There is more freedom in designing the structure, and it 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 [8,14,15,27,30]. 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. As an application of semistructured data, consider XML Routing [17,18,31]. 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, 10 MB/s. In one experimental system [31] publicly available tools were used to parse the XML documents and evaluate the XPath expressions, and the resulting performance was quite poor: about 2.6 KB/s for 10,000 XPath expressions.[1]

[1] The authors in [31] report parsing a 262 bytes XML document in 64.2 μs, 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.

Semistructured data vs. text processing. The XPath 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 [29], 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 [5]. One issue in stream data is that we cannot use an index, at least not in the traditional sense, 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, 10 MB/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 [4,16,19,20]. It then describes a few theoretical open problems in algorithms, inspired from XML stream processing.

2. XML stream processing

We briefly review here the syntax of XML and XPath, and define the XML stream processing problem.

XML, short for *eXchange Markup Language* [6,10], is a syntax for describing hierarchical data. The subset of XML that we consider in this paper contains only elements and data values, i.e., no attributes or processing instructions, and can be described by the following grammar:

```
X ::= string | <tag> X X ...X </tag>
```

Each XML document is a tree. A node, `X`, is either a `string`, or a begin-tag, `<tag>`, followed by a sequence of nodes, followed by an end-tag, `</tag>`. We call the node a *data value* node in the first case, and an *element node* in the second case. In addition, each XML data contains an extra node, the *document root*, which is right above the top most element node. To illustrate, the following example describes a list of persons:

```
<people>
   <person> <name> Smith </name>
            <address> <city> Seattle </city>
                      <state> WA </state> </address>
   </person>
```

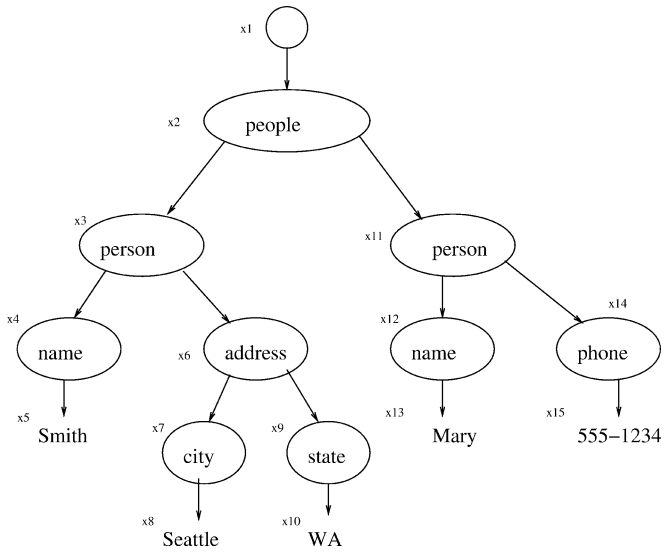

Fig. 1. An XML tree.

```
    <person> <name> Mary </name>
             <phone> 555-1234 </phone>
    </person>
</people>
```

The tree corresponding to this XML document is shown in Fig. 1. There are 15 nodes, denoted x1, . . . , x15. The document root node is x1.

SAX parsers. In XML stream processing we avoid reading and representing the entire XML document in main memory. Instead we read only one XML lexical token at a time, and process it immediately; this allows a system to scale to large streams without running out of memory. The parser for XML lexical tokens is called a SAX parser (Simple API for XML), and returns the following five types of events:

```
startDocument()
startElement(a)
text(s)
endElement(a)
endDocument()
```

Here `a` is a tag. For example, for the XML document in Fig. 1, the SAX parser returns the following sequence of events:

```
startDocument()
startElement(people)
```

```
startElement(person)
startElement(name)
text("Smith")
endElement(name)
startElement(address)
startElement(city)
text("Seattle")
endElement(city)
startElement(state)
text("WA")
endElement(state)
endElement(address)
endElement(person)
startElement(person)
    . . . similarly for Mary . . .
endElement(person)
endElement(people)
endDocument()
```

An application provides five call-back functions corresponding to the five event types.

XPath is a simple language that allows navigation through XML trees and returning a set of XML nodes [9]. We only consider a small fragment in this paper, described by the grammar below:

```
P ::= /E | //E
E ::= text() | tag | *
      E/E | E//E | E[Q]
Q ::= E | E = string | Q and Q | Q or Q | not Q
```

The language allows navigation through the XML document. In this fragment we only consider two navigation axes: / denotes the child axis, and // denotes the descendant axis. After each navigation step we can test nodes as follows: `text()` matches any string node in the XML tree, `tag` matches an element node with the corresponding tag, and $*$ matches any element node. Finally, `Q` is a predicate, obtained from equality conditions, and combined with boolean connectives.

We illustrate XPath by examples. Table 1 shows XPath expressions followed by the set of nodes return for the XML tree in Fig. 1.

All predicates are existentially quantified. That is, an XPath expression of the form `/a[b/text()=1]` returns the `a` element if it has *at least* one child labeled `b` with value 1.

Table 1

XPath expression	Result
`/`	`x1`
`/people`	`x2`
`/people/person`	`x3, x11`
`/people//address`	`x6`
`/people/address`	∅
`//name`	`x4, x12`
`//name/text()`	`x5, x13`
`//name[text()="Smith"]`	`x4`
`//name[text()="Smith"]` `[address/state/text()= "WA"`	`x4`
`/people/person[name/text()= "Mary" or` `address/state/text() = "WA"]`	x3, x11

While in the standard XPath semantics expressions return sets of nodes, throughout this paper we will treat them as boolean predicates (or boolean filters). More precisely, given an XPath expression P and an XML document D, we say that P *matches* D if the set of nodes returned by P on D is non-empty, and we say that P does not match D if the set of nodes returned is empty. For example, referring to the table above, all expressions match the document in Fig. 1 except for the expression `/people/address`.

Linear XPath expressions. An important subclass of XPath expressions are the linear XPath expressions, which do not have predicates. They are described by the grammar:

$$\texttt{P} ::= \texttt{/E} \mid \texttt{//E}$$

$$\texttt{E} ::= \texttt{text()} \mid \texttt{tag} \mid * \mid \texttt{E/E} \mid \texttt{E//E}$$

The XML stream processing problem. Formally, we are given a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of XPath expressions, called filters, where each filter has an associated oid. We receive an infinite stream of XML documents. The problem is to compute, for each document D in the stream, the set of oid's corresponding to the XPath expressions that match D.

3. Processing linear XPath expressions with automata

The case when all XPath expressions in a workload are linear has been studied more intensively in the literature than the branching case. The natural approach is to use automata theory: either a nondeterministic finite automaton (NFA) or on a deterministic finite automaton (DFA). We review them here.

3.1. Processing with NFAs

In this approach the set $\mathcal{P}$ of XPath expressions is translated into a single Nondeterministic Finite Automaton. The automaton can then be executed directly on the input XML stream, using a stack of sets of NFA states. Fig. 2(a) illustrates the NFA for a single XPath expression $//a/b/a/a/b$. Notice that this automaton is non-deterministic at state 0, since the $*$ transition matches any tag, including the a tag.

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 `startElement(`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 is implemented as a hash table at each state. On an `endElement(`a`)` SAX event we pop one set from the stack, and that becomes the current S. The stack never grow deeper than the depth of the XML document: this is typically small (say 10 or 20).

Let us analyze the time and space complexity of the NFA evaluation method. The space of the NFA is proportional to the size of $\mathcal{P}$. In addition we need a stack, whose depth 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 $\mathcal{P}$. Hence the space complexity is linear in the size of $\mathcal{P}$. Consider now the time complexity. The time to process one SAX events can be as large as the total number of states, hence is linear in the size of $\mathcal{P}$. As a consequence, the more XPath expressions we have, the slower we can process the input XML stream.

Systems that start from and NFA and incorporate various optimizations are XFilter [3], XTrie [7], and YFilter [12].

3.2. Processing with DFAs

Alternatively, we can evaluate the XPath expressions by constructing a DFA from the NFA. This is obtained using a standard powerset construct, see, e.g., [22]. As an illustration, Fig. 2(b) shows the DFA for the XPath expression $//a/b/a/a/b$, whose NFA is in Fig. 2(a).

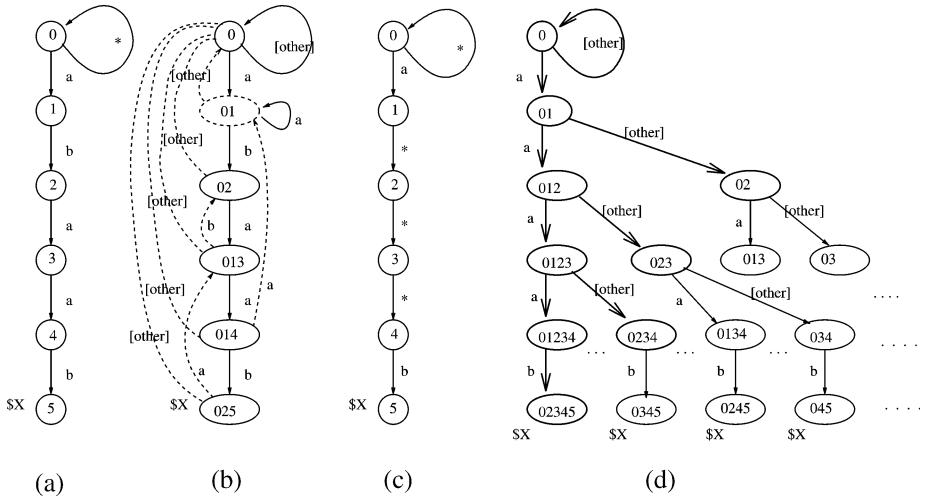

Fig. 2. The NFA (a) and the DFA (b) for $//a/b/a/a/b$. The NFA (c) and the DFA (with back edges removed) (d) for $//a/*/*/*/b$: here the eager DFA has $2^5 = 32$ states, while the lazy DFA, assuming the DTD `<!ELEMENT a (a*|b)>`, has at most 9 states.

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 `startElement(a)` and a `endElement(a)` 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. But in order to use this approach we need to consider first the size of the DFA, which, in the worst case, can be exponential in the workload. We review in the sequel the main results about the size of the DFAs for XPath expressions, from [16].

Let Σ denote the set of all tags, attributes, and text constants occurring in the set of XPath expressions, plus a special symbol ω representing any other symbol that could be matched by $*$ or $//$. For example, for the XPath expression $//a/b/a/a/b$ we have $\Sigma = \{a, b, \omega\}$.

Single XPath expression. We start by studying the number of states in the DFA for a single linear XPath expression. This is related to traditional text processing. For example, consider $P = //a/b/a/a/b$ the NFA and DFA are shown in Fig. 2(a) and (b), and they have the same number of states. The DFA corresponds to Knuth–Morris–Pratt's string matching algorithm [11] 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, as illustrated by Fig. 2(c) and (d).

Given a linear XPath expression P, let $|P|$ be the total number of node tests in P (i.e., tags or $*$), k be the number of $//$'s, and m be the largest number of $*$'s after a $//$, and before the next $//$ or before the end of the expression. For example if $P = //a/*//a/*/b/a/*/a/b$, then $|P| = 9$, $k = 2$, and $m = 2$ (we have 2 $*$'s after the second $//$). As another example, if $P = /*/*/*/*//a/*//b/c$ then $|P| = 8$, $k = 2$, $m = 1$ (the leading 4 $*$'s come before the first $//$ and are not considered in the definition of m). The following characterizes the size of the DFA for linear XPath expressions:

Theorem 1 [16]. *Given a linear XPath expression P, then the following holds. If $k = 0$ then there are exactly $1 + |P|$ states*; *otherwise there are at most $k + k \times |P| \times |\Sigma|^m$ states.*

When there are no wild-cards ($m = 0$), then the number of states is polynomial ($k + k \times |P|$). If, in addition, there is a single $//$, then number of states is linear ($1 + |P|$). For example, the XPath expression $P = //a/b/a/a/b$ has no $*$'s and a single $//$, hence the number of states is $1 + |P| = 5$ (see Fig. 2(b)). However, wild-cards introduce an exponential growth, $|\Sigma|^m$. This is unavoidable, as illustrated in Fig. 2(c), (d) where the DFA for $p = //a/*/*/*/b$, has 2^5 states. More generally, the DFA for $//a/*/\ldots/*/b$ has 2^{m+2} states, where m is the number of $*$'s.

Thus the size of the DFA for a single XPath expression is small, with the only exponent given by the number of wild cards between two consecutive occurrences of $//$. 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 [23].

Multiple XPath expressions. When the workload $\mathcal{P} = \{P_1, \ldots, P_n\}$ consists of several XPath expressions we can still construct the NFA (as their union), and attempt to de-

terminize it, but its size is in general much larger. Still, there is one special case which has a correspondence in text processing, more precisely to Aho and Corasick's dictionary matching problem [2,28]. In this problem 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 $Q = \{//w_1, \ldots, //w_p\}$. The main result in dictionary matching is that the number of DFA states is linear in the total size of Q. However, the dictionary problem captures only a toy XPath workload. Once we introduce multiple occurrences of //, the number of DFA states grows exponentially in the size of $\mathcal{P}$. For example, consider the four XPath expressions:

```
//book//figure
//table//figure
//chapter//figure
//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. Clearly, this argument can be extended to construct a workload of p XPath expressions for which the DFA requires at least 2^p states [16].

This prevents us from constructing the DFA for any workloads of XPath expressions, except perhaps for very small ones. We can still use a deterministic automaton, however, if we construct it lazily, as we discuss next.

3.3. Processing with lazy DFAs

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. To make this possible we need to maintain the internal structure of each DFA state, i.e., the set of NFA states.

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). We justify next that, in the case of XML stream processing, the lazy DFA has a small number of states, even when the set of XPath expressions is very large. For that, we describe two results in [16], which given upper bound guarantees on the number of states in the lazy DFA. The two results apply to different classes of XML documents: data-oriented XML documents, and text-oriented XML documents.

Data-oriented XML streams. DTDs [10] and XML Schemas [32] are both schema formalisms for XML that impose certain constraints on how the elements may be nested in side the XML document. For example they may say that a `<person>` element is allowed to contain a `<name>` and an `<address>` subelement, and not other types of elements. They can be naturally represented as a graph, with nodes corresponding to element names, and edges corresponding to inclusion relationships [1]: for example, the `<person>` node would have two edges, to `<name>` and `<address>`. Virtually, all XML documents occurring in practice have a schema. However, an XML stream processing systems may not know in advance the schemas of all XML documents that it needs to process.

Call a DTD or an XML Schema *simple* if any two distinct cycles in the associated graph have disjoint sets of elements. For example non-recursive DTDs or Schemas are simple, because they have no cycles at all. XML documents used to represent business *data*, called here data-oriented XML, have a schema that is either non-recursive, or has a limited form of recursion, corresponding to a hierarchy (e.g., a `<part>` element may contain other `<part>` elements). These are simple schemas, because each cycle consists of a single element. The theorem below says that, if the documents XML stream have simple schemas (which do not need to be available to the stream processor) then the size of the lazy DFA is bounded by an amount which is independent on the number of XPath expressions.

Theorem 2 [16]. *Let D be the number of simple paths in a simple DTD graph, and let d be the maximum number cycles on any simple path. Let $\mathcal{P}$ be a set of XPath expressions, and $n = \max\{|P| \mid P \in \mathcal{P}\}$. Then the lazy DFA has at most $1 + D \times (1 + n)^d$ states.*

Text-oriented XML instances encoding structured text, however, have mostly non-simple schemas: for example a `table` may contain a `table` or a `footnote`, and a `footnote` may also contain a `table` or a `footnote`. Here, both (`table`) and (`table`, `footnote`) are cycles, and they are not disjoint; hence the schema is not simple. In such cases one can give an upper bound on the size of the lazy DFA in terms of *Data Guides* [13]. For a given XML instance, the data guide G is defined to be the trie of all root-to-leaf element sequences. An alternative definition is that the data guide is the deterministic automaton that accepts all sequences of elements in the XML document, starting at the root.

In practice, users create structured text documents that obey certain conventions, which are not captured by any schema. For example they may place a `footnote` inside a `table`, but never a `table` inside a `footnote`; or, more subtly, they may never place a `table` inside *both* a `footnote` and a `figure` (this constraint cannot be expressed by a schema). This tends to limit the number of distinct sequences of elements that can occur in the XML document.

Indeed, it has been observed empirically that real XML data instances tends to have small data guides, regardless of its DTD. All XML data instances described in [24] have very small data guides (under 100 nodes), except for Treebank [25], where the data guide has $G = 340{,}000$ nodes. The following shows that the lazy DFA for an XML stream with a small data guide has a number of states that is independent on the number of XPath expressions:

Corollary 1 [16]. *Let G be the number of nodes in the data guide of an XML stream. Then, for any set $\mathcal{P}$ of XPath expressions the lazy DFA for $\mathcal{P}$ on that XML stream has at most $1 + G$ states.*

Experiments reported in [16] show that, when run on large sets of XPath expressions (up to 1,000,000) and on several real instance so of XML data, the number of states in the lazy DFA was under 100. The only exception was for the Treebank data instance: here the number of states was 44,000.

The lazy DFA at runtime. The graph in Fig. 3, taken from [16], illustrates the throughput of the lazy DFA. The throughput is shown as a function of the amount of data consumed,

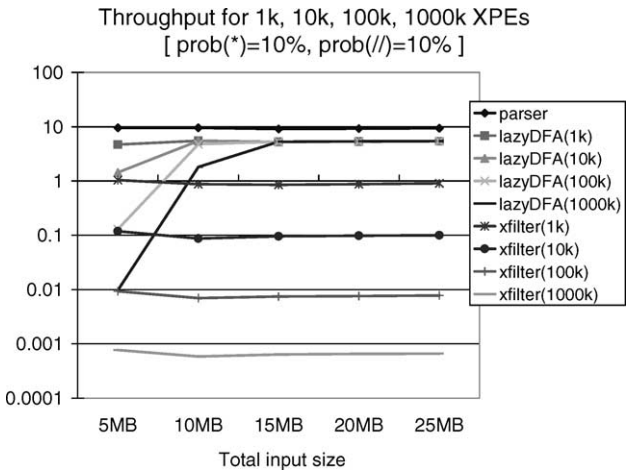

Fig. 3. Experiments illustrating the throughput of the DFA vs. XFilter [3], as a function of the amount of XML data consumed. The number of XPath expressions varies from 1k (1000), to 1000k ($= 10^6$).

both for the lazy DFA, for XFilter [3] (without list balancing), and for the parser only. Recall that XFilter uses an approach based on a NFAs. The data is the NASA XML dataset [26], and is about 25 MB. The four sets of XPath expressions were synthetically generated, and contain 1k ($= 1000$), 10k, 100k, and 1000k XPath expressions, respectively. 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 5.4 MB/s. This is about $1/2$ the parser's throughput. Importantly, the stable throughput is independent on the number of XPath expressions in the query set. By contrast, XFilter does not need a warm-up phase, but the throughput degrades as we increase the number of XPath expressions, until it is about 50,000 smaller than that of the lazy DFA. This is consequence of the fact that the time taken to process on SAX event by an NFA increases linearly with the number of XPath expressions.

Atomic predicates. One can add atomic predicates to linear XPath expressions and still process them with automata. For example the XPath expression:

```
/people/person[name/text() = "Smith"]
```

can be processed by an automaton with four transitions: `people`, `person`, `name`, `"Smith"`. There is a difference, however. The tags in a set of XPath expressions form a rather small set, since they are constrained by the schema. By contrast, the atomic values may form a large set, and one wonders how this affects the number of states in the lazy DFA. The observation here is that all transitions corresponding to atomic predicates end in terminal states (i.e., without outgoing transitions). There may be many more such states than in the rest of the lazy DFA, however, there are not more than XPath expressions in the workload. In other words, the lazy DFA for a set of linear XPath expressions with atomic predicates consists of a core set of states that contain only transitions labeled with tags, and for which the upper bounds in Theorem 2 and Corollary 1 apply, and a large set of terminal states, in one-to-one correspondence with the atomic predicates in the workload.

4. Processing branching XPath expressions with pushdown automata

The case when XPath expressions have branches is more difficult. A top-down pass through the XML tree may not suffice, but instead needs to be followed by, or replaced by a bottom-up pass. This case is more related to the *tree pattern matching* problem, for which exponential-space algorithms are known since the seminal paper by Hoffman and O'Donnell [21].

To illustrate, consider the XPath expression:

```
P = /root/a[b/text()=1 and c/text()=2]
```

Equivalently: `/root/a[b/text()=1][c/text()=2]`. One attempt may be to linearize it, that is compute separately the expressions:

```
P1 = /root/a/b[text()=1]
P2 = /root/a/c[text()=2]
```

The two expressions can be translated into NFAs, as explained at the end of the previous section. However, if a document matches both `P1` and `P2`, it doesn't necessarily matches `P`, as illustrated by:

```
<root> <a> <c> 2 </c> </a> <a> <b> 1 </b> </a> </root>
```

which matches `P1` and `P2` but not `P`.

Clearly, we need a method that processes the original XPath expression `P` directly, without separating the branches. This can be done only bottom up in the XML tree, in other words while listening to the end tags. To see this process consider the following XML document:

```
<root> <a> <c> 2 </c> <d> 9 </d> <b> 1 </b> </a> </root>
```

After reading `<c> 2 </c>` the systems knows that the predicate `[c/text()=2]` matched. Then it needs to wait for `<b> 1 </b>` before knowing that the predicate `[b/text()=1]` matched too. Only when it finds the end tag `</a>` can it conclude that `a[b/text()=1][c/text()=2]` matches. This corresponds to a bottom-up evaluation. Moreover, notice that the system needs to remember which predicates have matched, after the end of the element that has matched them: that is, it remembers that `[c/text()=2]` matched, while scanning over the element `<d> 9 </d>`.

These observations lead to a modified pushdown automaton for processing workloads of XPath expressions in [19]. The automaton, called an XPush machine, is deterministic, and uses a stack to keep track of which predicates have been matched at each level in the XML document. The symbols on the stack correspond to sets of predicates that have matched. When reading a begin tag, `<tag>`, the XPush machine pushes on the stack a symbol corresponding to the empty set of predicates. When reading an end tag, `</tag>`, the automaton inspects the last two symbols on the stack, and computes a new symbol corresponding to a larger set of matched predicates.

Like the DFA, the XPush machine cannot be precomputed. The analogy here is with linear tree pattern matching, for which Hoffman and O'Donnell have described an exponential sized data structure [21]: clearly such an approach does not work in our setting,

due to the large number of XPath expressions. Instead, the states in the XPush machine are computed lazily, as we explained for the DFA. There are two major differences between the lazy XPush machine and the lazy DFA. First, the number of states in the XPush machine tends to be much larger: tens of thousands vs. less than one hundred. The reason is that the states in the XPush machine correspond to sets of predicates that may have matched, and there is no intrinsic upper bound here, like the data guide. In fact, the number of states never reaches an upper bound at runtime, and main memory is eventually exhausted. This is addressed by resetting the XPush machine and restarting it from scratch. The states in the deterministic XPush machine act like a cache: the first time a state is encountered there is a high computation cost, which is later amortized when the state is reused. When memory runs out, the cache is cleared. Experiments in [19] report a hit ratio of about 95%, i.e., 95% of the state accesses can reuse an existing state, while 5% need to construct a new one.

Second, the inner structure of the states of the XPush machine is much smaller than that of the lazy DFA: this makes the XPush machine practical, despite its large number of states. In the experiments reported in [19] the average number of predicates in a state of an XPush machine was less than 1000, sometimes less than 100; for comparable workloads and data sets, the average number of NFA states per DFA state reported in [16] was tens of thousands and higher. The reason is that the XPush machine starts the computation at the leaves of the XML tree, by matching atomic predicates of the form `[text()= "value"]`. Since such predicates have usually low selectivities, only few predicates in the workload match any given atomic value in the XML document. This property propagates upward in the XML tree: only few predates match any given node in the tree. By contrast, the lazy DFA proceeds top-down: tags and wildcards have much higher selectivity, and hence many more XPath expressions will continue to match, before a leaf node is encountered that reduces their number significantly.

5. Discussion and open problems

The XML Stream Processing creates new challenges for algorithm design. Standard techniques from text processing and pattern matching cannot be applied directly, because they do not scale to the number of XPath expressions required here. Instead, the algorithm designer needs to exploit the hidden structure of the data, and/or queries, like data guides were used to justify the lazy DFA. One expects such structure to be found in real XML documents or in XPath query workloads, but it is difficult to formalize. We enumerate here some possible research problems.

Constant time processing of branching XPath expressions. A lazy DFA processes every SAX event in O(1) amortized time. This is because the warm-up phase ends after processing the first few megabytes of data, and afterward the time needed to process one SAX event is O(1).

The XPush machine fails short of achieving the same goal for branching XPath expressions, because it never gets past the warm-up phase. Eventually the main memory is exhausted, and the machine has to be reset and we re-compute all its states. The question is whether it is possible to compute large workloads of branching XPath at O(1) amortized

time per SAX event. Clearly, this is not possible theoretically, but it may be possible if one makes certain realistic assumptions about the structure of the data and/or the workload.

Optimize the lazy machines. Both the lazy DFA [16] and the lazy XPush machine [19] construct their states at runtime; they represent the inner structure of a state as a sorted array. These arrays use a lot of space and are relatively expensive in operations like union or transition-computation, which need to be performed at runtime. Several opportunities exists here. One is to delete the inner structure entirely when one can prove that all possible transitions from that state have been exhausted (e.g., by inspecting the XPath workload, or a DTD, when one is available). Another is to delete the inner structure, even is that state is not yet exhausted: when we need to look up its inner structure, we can re-compute it dynamically, from other states. This trades off space for time. Finally, alternative representations to sorted arrays could be considered: for example, if one uses trees, then one could share common subtrees between states, resulting in memory savings.

Sublinear computations. The Boyer–Moore algorithm matches a string in a text in sublinear time. Is it possible to match a set of XPath expressions on an XML document in sublinear time? An approach that uses an additional data structure is described in [4]. There a *stream index* (SIX) is attached to each XML document. For example in an XML routing application, the source of the XML document would compute the SIX, attach it to the XML document, then all routers that receive that document can use it while evaluating their XPath expressions. The SIX acts like an index for the XML packet, allowing direct access to its content. The possibilities here are endless: an alternative form of data structure allowing direct access is described in [20].

6. Conclusions

This paper has described the XML stream processing problem. This is related to classic problems like text processing and tree pattern matching, yet requires a new suite of techniques, because the assumptions under which they are deployed differ: the number of XPath expressions is much larger that usual in text processing, while the depth of the XML documents is much smaller than a typical text length. The challenge is to design techniques that scale, and which can be guaranteed to work in practice. The latter is particularly hard, because the assumptions about the hidden structure of the data and/or query workloads (like the data guide) are critical for performance, but are not stated explicitly and need to be carefully defined in a formalism that normal users can understand in order to determine if they apply to their settings.

Acknowledgements

Most of the results mentioned in this paper have been obtained in work done in collaboration with T.J. Green, A. Gupta, A. Halevy, 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] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web: From Relations to Semistructured Data and XML, Morgan Kaufmann, 1999.

[2] A. Aho, M. Corasick, Efficient string matching: an aid to bibliographic search, Comm. ACM 18 (1975) 333–340.

[3] M. Altinel, M. Franklin, Efficient filtering of XML documents for selective dissemination, in: Proceedings of VLDB, Cairo, Egipt, 2000, pp. 53–64.

[4] I. Avila-Campillo, T.J. Green, A. Gupta, M. Onizuka, D. Raven, D. Suciu, XMLTK: An XML toolkit for scalable XML stream processing, in: Proceedings of PLANX, 2002.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the ACM SIGART/SIGMOD Symposium on Principles of Database Systems, 2002, pp. 1–16.

[6] J. Bosak, XML, java, and the future of the web, World Wide Web J. 2 (4) (1997) 219–227.

[7] C. Chan, P. Felber, M. Garofalakis, R. Rastogi, Efficient filtering of XML documents with XPath expressions, in: Proceedings of the International Conference on Data Engineering, 2002.

[8] V. Christophides, S. Abiteboul, S. Cluet, M. Scholl, From structured documents to novel query facilities, in: R. Snodgrass, M. Winslett (Eds.), Proceedings of 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, MN, 1994, pp. 313–324.

[9] J. Clark, XML path language (XPath), 1999, http://www.w3.org/TR/xpath.

[10] W.W.W. Consortium, Extensible markup language (xml) 1.0, 1998, http://www.w3.org/TR/REC-xml.

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

[12] Y. Diao, P. Fischer, M. Franklin, R. To, Yfilter: efficient and scalable filtering of xml documents, in: Proceedings of the International Conference on Data Engineering, San Jose, CA, 2002.

[13] R. Goldman, J. Widom, DataGuides: enabling query formulation and optimization in semistructured databases, in: Proceedings of Very Large Data Bases, 1997, pp. 436–445.

[14] G. Gonnet, R. Baeza-Yates, T. Snider, Lexicographical indices for text: inverted files vs. PAT trees, in: W.B. Frakes, R.A. Baeza-Yates (Eds.), Information Retrieval: Data Structures & Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1992, pp. 66–82.

[15] G. Gonnet, F. Tompa, Mind your grammar: A new approach to modelling text, in: Proceedings of 13th International Conference on Very Large Databases, 1987, pp. 339–346.

[16] T.J. Green, G. Miklau, M. Onizuka, D. Suciu, Processing XML streams with deterministic automata, in: Proceedings of ICDT, 2003.

[17] M. Gudgin, M. Hadley, J. Moreau, H. Nielsen, SOAP version 1.2 part 1: Messaging framework, 2001, Available from the W3C, http://www.w3.org/2000/xp/Group/.

[18] M. Gudgin, M. Hadley, J. Moreau, H. Nielsen, SOAP version 1.2 part 2: adjuncts, 2001, Available from the W3C, http://www.w3.org/2000/xp/Group/.

[19] A. Gupta, D. Suciu, Stream processing of XPath queries with predicates, in: Proceedings of ACM SIGMOD Conference on Management of Data, 2003.

[20] A. Gupta, D. Suciu, A. Halevy, The view selection problem for xml content based routing, in: Proceedings of PODS, 2003.

[21] C.M. Hoffmann, M.J. O'Donnell, Pattern matching in trees, J. ACM 29 (1) (1982) 68–95.

[22] J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Englewood Cliffs, NJ, 1979.

[23] Z. Ives, A. Halevy, D. Weld, An XML query engine for network-bound data, 2001, unpublished.

[24] H. Liefke, D. Suciu, XMill: an efficient compressor for XML data, in: Proceedings of SIGMOD, Dallas, TX, 2000, pp. 153–164.

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

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

[27] G. Navarro, R. Baeza-Yates, Proximal nodes: a model to query document databases by content and structure, ACM Trans. Inform. Syst. 15 (4) (1997) 400–435.

[28] G. Rozenberg, A. Salomaa, Handbook of Formal Languages, Springer-Verlag, Berlin, 1997.

[29] A. Salminen, F.W. Tompa, Pat expressions: an algebra for text search, in: Papers in Computational Lexicography: COMPLEX'92, 1992, pp. 309–332.
[30] A. Salminen, F.W. Tompa, Pat expressions: an algebra for text search, Acta Linguistica Hungarica 41 (1–4) (1994) 277–306.
[31] A. Snoeren, K. Conley, D. Gifford, Mesh-based content routing using XML, in: Proceedings of the 18th Symposium on Operating Systems Principles, 2001.
[32] H. Thompson, D. Beech, M. Maloney, N. Mendelsohn, XML schema part 1: structures, May 2001, http://www.w3.org/TR/xmlschema-1/.