Light-weight XPath Processing of XML Stream with Deterministic Automata

Makoto Onizuka
NTT CyberSpace Laboratories, NTT Corporation
1-1 Hikari-no-oka, Yokosuka, Kanagawa, 239-0847 Japan
onizuka.makoto@lab.ntt.co.jp

ABSTRACT

Several applications based on XML stream processing have recently emerged, such as those for air traffic control and the selective dissemination of information (SDI). Their common need is to process a large number of XPath expressions in continuous XML streams at high throughput.

This paper proposes four techniques for XPath expression processing based on Deterministic Finite Automata (DFA) for two purposes: to improve the memory usage efficiency of the automata and to support the processing of branching XPath expressions. The first technique, called *n-DFA*, clusters the given XPath expressions into n clusters to reduce the number of DFA states. The second, called *shared NFA state table*, lets the Non-Deterministic Finite Automata (NFA) state set be shared among the DFA states. Our experiments show that memory usage in an 8-DFA can, with the *shared NFA state table*, be reduced to 1/40th that of the original 1-DFA. The *optimized NFA conversion* and *general XPath expression processing algorithm* techniques contribute to the processing of branching XPath expressions efficiently; overall performance is better than is possible with earlier approaches.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

General Terms

Performance

Keywords

Streaming XML, XPath processing, Automata, Selective dissemination of information

1. INTRODUCTION

Several applications of XML stream processing have emerged recently in the following areas: air traffic control [19], the selective dissemination of information (SDI) [2, 18], and continuous queries [5]. Their common need is to process a large number of XPath expressions (XPEs) over continuous XML streams at high throughput.

For example, consider an XML router network that processes an XML stream [19] for an air traffic control system. The root routers (data producers) produce air traffic data in XML format (an XML stream is a sequence of XML packets) and forward the XML packets to their children routers (internal routers). These internal routers receive the XML stream from their parent routers and forward the XML packets to their children routers or clients. Clients (data consumer) connect to the routers and provide an XPath query that describes the portions of the XML stream that they would like to receive. To improve system performance by reducing the number of XML packets, each XML router replicates a set of XPath queries from its descendant clients, filters incoming XML packets against those XPath queries, and forwards the filtered XML packets requested by its descendant clients. Regarding this process, Snoeren's paper [19] claimed that the performance of public XML tools should be improved.

LazyDFA [9], and other Non-Deterministic Finite Automata (NFA) based algorithms (XFilter [2], YFilter [7], and XTrie [4]) have been proposed as efficient algorithms for processing a large number of XPEs in XML streams. LazyDFA, which "lazily" constructs Deterministic Finite Automata (DFA), is superior to the others in terms of processing performance, because it insures a constant high throughput (as fast as an XML parser) for a collection of single-path XPEs, while the others' performance is linearly degraded against the number of XPEs. However, there are two problematic issues regarding lazyDFA; 1) when it processes complex XML documents, it requires a large number of DFA states, and thus can run out of memory, and 2) it doesn't handle branching XPEs; it leaves them to the applications.

1.1 Our Contributions

To address the above two issues, this paper proposes four techniques based on Finite Automata. The first, a variant of Finite Automata, is composed of n DFAs (*n-DFA*) to reduce both the total number of DFA states and the size of the NFA state table stored in each DFA state. The second, the *shared NFA state table*, reduces the memory requirements of the NFA state tables. These two techniques resolve the first issue. The *optimized NFA conversion* and *general XPE processing algorithm* techniques resolve the second issue and allow branching XPEs to be efficiently processed.

n-DFA and shared NFA state table

Converting XPEs into a DFA is the most efficient approach to processing XML streams, as long as the generated DFA fits in memory. The lazyDFA paper [9] shows that lazyDFA is applicable to data-oriented real XML, because the upper bound of the number of

DFA states is not large. However, when the XML stream is complex such as document-oriented XML, both the number of DFA states and the size of its NFA state table become excessive; in such a case, lazyDFA could run out of memory.

We consider TreeBank XML (XML-ized TreeBank linguistic database [14]) to be typical of complex real XML documents. The lazyDFA paper [9] also proves that lazyDFA can have, at most, the same number of DFA states as it has DataGuide nodes [8], and that the average size of the NFA state table is about $p/10$ (p is the number of XPEs) when both * (wildcard) node test and // (descendant) axis have probabilities of 5%. In addition, Liefke's paper [13] shows that the number of DataGuide nodes for TreeBank is 340,000. Suppose we convert 100,000 XPEs that have 5% probabilities for both * node test and // axis into lazyDFA, and further that an NFA state table is implemented as an NFA state pointer array. The total NFA state table size is, in the worst case, given by

```
(number of DFA state)*(average NFA state table size)*
  (size of NFA state pointer)
  = 340,000 * 100,000/10 * 4bytes
  = 13.6Gbytes
```

[1] Thus, lazyDFA is not applicable to highly irregular XML if the computer doesn't have a large memory.

To reduce both the number of DFA states and NFA state table size, we propose *n-DFA*, which, given a set of XPEs, groups the XPEs into n clusters, and constructs a DFA for each cluster. Our experiments on TreeBank XML show that this approach reduced the memory usage to 1/8th for 4-DFA and to 1/16th for 8-DFA; the XPEs were clustered according to axis type (/, //). In addition, we reduce the NFA state table memory usage of the DFA states by applying the *shared NFA state table* technique, which allows the DFA states to share a common NFA state set. This reduced the NFA state table size in a DFA state by a factor of 20 in the TreeBank XML experiments described in Section 6. By combining these two techniques, the memory usage in 8-DFA with a *shared NFA state table* is 40 times less than that of the original 1-DFA for TreeBank XML. This means that lazyDFA can support complex XML streams.

Branching XPE Processing

In general, an XPE contains several predicates, each of which is converted into a branch, but lazyDFA leaves branch processing to the applications. An XPE can be divided into a set of related single-path XPEs [9]. For example, the following XPE,

```
/bib/book[@year=1999]
         [contains(title/text(),'XML')]/author
```

Figure 1:

can be divided into four interrelated single-path XPEs in Figure2.

```
Q:     $Y IN /bib/book
       $Z IN $Y[@year=1999]
       $U IN $Y[contains(title/text(),'XML']
       $X IN $Y/author
```

Figure 2:

We propose two techniques for processing branching XPEs. The first, *optimized NFA conversion*, converts the NFA of a branching XPE into a branch-free NFA by using ordered constraints on XML. This technique is very useful because as is usual in data-oriented XML processing; XML schema is defined on an XML document that has ordered element constraints. The lazyDFA is applicable for such branch-free NFAs and ensures constant high throughput (the cost is $O(1)$ for each SAX event). The second is *general XPE processing algorithm* for single-path XPE processors (e.g. *n-DFA*). A single-path XPE processor evaluates the interrelated single-path XPEs and passes application events (filtered SAX events plus Variable events that indicate which XPE becomes matched or un-matched) to the branching XPE processing algorithm. The branching XPE processing algorithm receives the application events and evaluates branching XPEs in a bottom-up manner. The algorithm cost for each SAX event is $O(|ST|)$ (where ST is the set of related single-path XPEs). This is smaller than the XTrie's branching processing cost of $O(|ST_2|) * O(L_{max})$, where ST_2 represents the substrings for which each single-path XPE in ST is divided with * and //; so it is obvious that $|ST| \leq |ST_2|$. L_{max} is the maximum number of levels in the incoming XML stream, and $O(L_{max})$ corresponds to the cost of connecting the divided substrings with * and //.

2. BACKGROUND

2.1 XPE Processing Problem in XML streams

One of the key problems to the XPE processing of XML streams can be expressed as follows: "Given a large set Q of XPEs and an incoming XML stream D, locate the subset of XPath expressions Q' in Q that match D."

XPath models an XML document as a tree of nodes, and forms a path expression that selects a set of nodes in an incoming XML. The path expression, called *location path*, is composed of a sequence of *location steps*. Each location step has three parts: axis (/, //), node test (can be wildcard), and optional predicate list. We define such an XPE, a *single-path XPath expression*, i.e. it doesn't have any predicate or one only at the tail location step. The other XPEs are called *general XPath expressions*. For example, //book[contains(title/text(),"XML")] is a single-path XPE, because the predicate is specified only at the tail location step. The XPE in Figure 1 is a general XPE since it can be divided into the interrelated *single-path XPEs* shown in Figure 2.

2.2 LazyDFA

LazyDFA [9] processes interrelated single-path XPEs and passes application events (filtered SAX events plus Variable events that indicate which XPE has become matched or un-matched) to the applications. The lazy construction technique is influenced by the lazy transition evaluation in automata [1].

LazyDFA processing proceeds as follows. First, the lazyDFA processor inputs a query tree (interrelated single-path XPEs) and each XPE is converted into an NFA (XPath NFA). All XPath NFAs are combined into a single large NFA (combined NFA) by adding a new start state that has epsilon edges next to every start state of the XPath NFA (an interrelation between XPEs is also implemented as an epsilon edge between XPath NFAs). Second, the incoming XML stream is parsed by a SAX parser that generates a stream of *SAX events*; this is input to the lazyDFA processor,which then evaluates the XPEs and generates a stream of *application events*. The applications events are filtered SAX events plus Variable events. During the evaluation, a DFA is lazily constructed from the combined NFA; transitions are evaluated lazily, and only the needed edges and DFA states are constructed.

There are several important features of lazyDFA. To enhance performance, the DFA transition table is implemented as a hash al-

[1]In a realistic case, the number of DFA states is about 43,000 with the above settings, but it will approach the upper bound, 340,000 if we increase the number of XPEs or the probabilities of * or //.

gorithm; thus element/attribute transition lookup is efficiently processed. Its ideal cost is $O(1)$[2]. A value (character() in SAX event) transition can be implemented in several ways, according to the predicate types in the query tree; an exact match can be done with a hash algorithm, as described in [9], arithmetical comparisons can be done with a variant interval tree [6], and a substring match can be done with fast string searching algorithms (e.g. automata based algorithm or Knuth-Morris-Pratt algorithm [6]). These algorithms can also be applied to other NFA-based algorithms.

The NFA state table size in a DFA state is proportional to the number of matching XPEs, because the table contains NFA states for all candidate XPEs that match the incoming XML stream. The number of DFA states means the number of possible cases during XPE evaluation. Generally speaking, when the number of DFA states is small, many XPEs tend to become candidates at the same DFA state, so the NFA state table size tends to be large. Otherwise, when the number is large, the NFA state table size tends to be small. In addition, it is obvious that the number of candidate XPEs increases with the number of XPEs, // probability, and * probability. The // probability, in particular, enlarges the number of DFA states for the following reason. In irregular and deeply nested XML processing, the number of lazyDFA states approaches that of eagerly constructed DFA states. A theorem in Reference [9] shows that the number of eagerDFA states is exponential against the number of XPEs with //.

As mentioned, the processing of branching XPEs is left to the application, and this is another important issue, because in most practical cases, the XPE is a general XPE with branches.

3. SYSTEM ARCHITECTURE

Figure 3 depicts a system architecture that includes an application, a branching XPE processor, a single-path XPE processor, and an XML parser.

The application registers general XPEs Q to a branching XPE processor. Theses are decomposed into single-path XPEs Q', which are then registered in the single-path XPE processor. The XML parser parses XML stream and invokes SAX events. The SAX events are passed to the single-path XPE processor that processes the single-path XPEs Q' and invokes application events. The application events are passed to the branching XPE processor that processes the given queries Q and sends a subset of Q to the application.

4. SINGLE-PATH XPE PROCESSING

4.1 n-DFA

The *n-DFA* key technique groups XPEs into several clusters according to axis types (/, //) at each XPE depth level. This approach reduces both the number of DFA states and NFA state table size for the following reasons.

- In practical use, // axis probability is low (0% to 20%), thus // axis probability is lower than / axis probability;
- The number of DFA states grows exponentially with the number of // (proved by Reference [9]). For example, if we add a new XPE starting with // axis to lazyDFA, in the worst case, the original number of DFA states is multiplied by the number of DFA states of the newly inserted XPE to form the current number of DFA states.

[2]LazyDFA is implemented with an STL library, and its performance is linear to the number of entries

```
Algorithm XPEs-division(Q, N)
Input:  Q is a set of single-path XPEs.
        N is the number of clusters.
Output: R is the set of XPEs set,
 1. lmax = log2(N);
 2. R = {Q};
 3. for i = 1 to lmax do //for each XPE depth
 4.     R2 = {};
 5.     foreach q in R
 6.         R2 = R2 ∪ XPEs-division-sub(q,i);
 7.     R = R2;
 8.     i = i + 1;
 9. return R;
```

```
Algorithm XPEs-division-sub(q, i)
Input:  q is a set of single-path XPEs.
        i is XPE depth.
Output: {C1,C2} is set of XPEs set,
 1. C1 = C2 = {};
 2. foreach xpe in q
 3.     if is-n-Axis(xpe,i,"//")
 4.         C1 = C1 ∪ xpe;
 5.     else
 6.         C2 = C2 ∪ xpe;
 7. return {C1,C2};
```

Figure 4: Algorithm to divide XPEs into n clusters

- The average NFA state table size is proportional to the number of //. For the previous example, if we add a new XPE starting with // axis to lazyDFA, the average NFA state table size should be increased.

Here, we give a heuristic on how to divide XPEs. If we group XPEs into a // cluster and a / cluster at each XPE depth (from 1 to m) in a nested manner, we get 2^m-DFA and can avoid both exponential DFA state growth and NFA state table size growth in the / cluster.

Figure 4 shows the division algorithm for *n-DFA*. The algorithm "XPEs-division" is the main function and its subroutine is "XPEs-division-sub function". The XPEs-division-sub function groups XPEs into two clusters *C1* (// cluster) and *C2* (/ cluster) by checking the axis at depth i for every XPE. The is-n-Axis(XPE, n, $axis$) subroutine checks whether the XPE's axis at depth n equals $axis$. For example, if we divide Q = {//a, //b, /a/b, /a//b, //a/b, /a/b/c} into four clusters, the above algorithm returns four clusters ({//a,//b}, {//a/b}, {/a//b}, {/a/b,/a/b/c}).

However, the performance of *n-DFA* falls as the number of clusters increases, because it must deal with n number of DFAs. Since DFA performance is constant, the *n-DFA* performance is n times slower than that of *1-DFA*. Thus, it is important to set n according to the available memory.

4.2 Shared NFA state table

As we saw in Section 1, the size of the NFA state table is another source of memory exhaustion in lazyDFA. To solve this problem, we introduce a *shared NFA state table*. It holds those NFA states (self-loop-states) that are transitive via a wildcard self-loop edge [3], and shares them among the NFA state tables of DFA states. Once

[3]//n is converted into a wildcard self-loop edge and an *n* labeled edge.

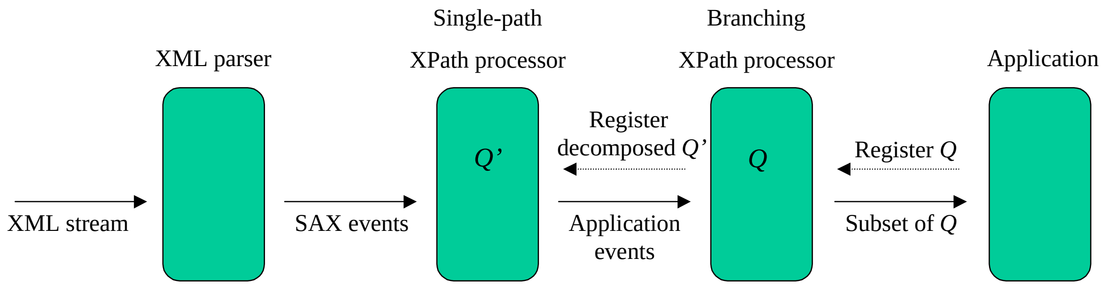


Figure 3: System Architecture

the self-loop-states have been put into an NFA state table in DFA states, they exist in all NFA state tables of the descendent DFA states, because these NFA state are transitive via wildcard self-loops. For example, if the NFA state table in a DFA state D is $\{S_1, S_2, S_3\}$ and S_1 and S_2 have wildcard self-loops, the NFA state table of the descendent DFA states also contain S_1 and S_2. This is a simple example, but it implies that when we have a large number of XPEs, sharing the NFA states can greatly reduce the DFA's memory usage.

In a canonical DFA state construction, we gather all NFA states (will be stored in an NFA state table in the constructing DFA state) that can be transitive from the NFA states stored in the current DFA state according to input labels [11]. In the *shared NFA state table* technique, we gather two types of NFA states: transitive with only wildcard self-loop edges (shared NFA state table), and transitive with other edges (exclusive NFA state table). If a DFA state already has a shared NFA state table, then the next DFA state has not only the same shared NFA state table, but also another shared NFA state table because of the transition. Consequently, the NFA state table structure is composed of a set of shared NFA state tables and an exclusive NFA state table as follows

```
NFA state table = (set<shared NFA state table>,
                   exclusive NFA state table)
```

For example, if Q = {//a, //b, //c}, the number of DFA states can be five (equal to eager DFA construction) and every DFA state shares the shared NFA state table *t0* that contains the root NFA states of all XPEs S1,S2,S3 (Figure 5).

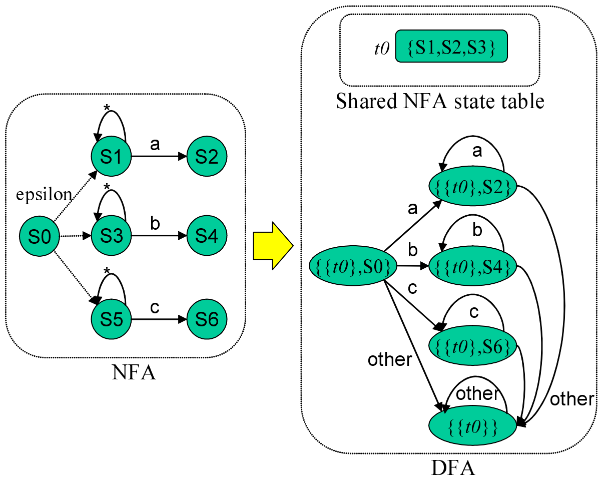


Figure 5: An example of the shared NFA state table

However, such an NFA state implementation makes it expensive to check for DFA state equality when constructing a new DFA state. Determining DFA state equality necessitates a comparison of NFA state tables, and an experiment (not described here) showed that this is very expensive since we must temporarily unpack all shared NFA state tables to build the original NFA state table. Instead, we choose a low cost equality check; we compare each shared NFA state table and exclusive NFA state table of one DFA state with those of another DFA state. If all of these shared and exclusive NFA state tables are equal, the two DFA states are evaluated as being equal. There is a small penalty in that we can have duplicated DFA states. For example, this equality check returns false when we compare two NFA state tables $\{\{\{S_0, S_1\}, \{S_1, S_3\}\}, S_2\}$ and $\{\{\{S_0, S_1\}, \{S_3\}\}, S_2\}$, even though they both are equivalent to $\{S_0, S_1, S_3, S_2\}$. In spite of this penalty, our experiments in Section 6 show that the *shared NFA state table* technique efficiently reduces memory usage.

5. BRANCHING XPE PROCESSING

5.1 Optimized NFA conversion

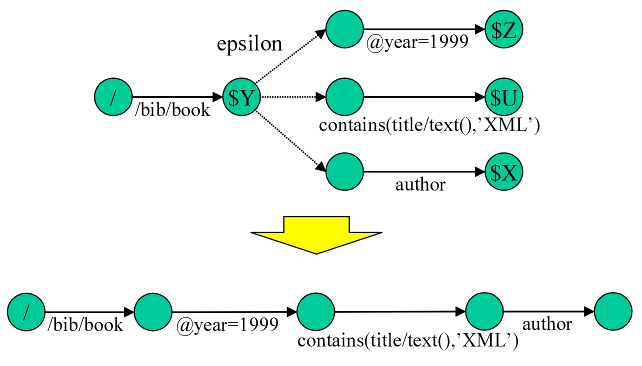


Figure 6: An example of optimized NFA conversion

In the usual case of data-oriented XML processing, XML schema is defined on the XML document and has ordered element constraints. The ordered element constraint, which is defined by a sequence of elements, indicates that the child elements must follow the given order of elements. For example, consider the DTD;

```
<!ELEMENT bib    (book)*>
<!ELEMENT book   (title, author+)>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT title  (#PCDATA)> ...
```

The declaration for *book* element defines such an ordered element constraint; the child elements must follow the order of *title*, *author*.

When there are ordered element constraints and there are XPEs on whose branches the constraint is placed, the branches can be processed sequentially. Consequently, such an XPE can be converted into a branch-free NFA. Figure 6 depicts an optimized NFA for the XPE in Figure 1 given the above ordered constraints. Since there is an ordered constraint ($title$, $author$), the branches can be sequentially processed in the same order.

5.2 Branching XPE Processing Algorithm

As we saw in Figure 3, we first decompose the general XPEs into single-path XPEs and register these single-path XPEs in a single-path XPE processor. The decomposed single-path XPE is either leaf or non-leaf in the query tree. The branching XPE processing algorithm accepts application events from the single-path XPE processor; the application events indicate which XPE becomes matched (startContext event) or un-matched (endContext event). Branch evaluation is done in a bottom-up manner. A leaf single-path XPE is evaluated as true, when the single-path XPE processor invokes an endContext event of the XPE. A non-leaf XPE is evaluated as follows. If several branches are conjunctive, their parent XPE is evaluated as true when all branches are evaluated as true. If several branches are disjunctive, their parent XPE is evaluated as true when at least one branch is evaluated as true. If there is no parent XPE, the original branching XPE is evaluated as true. If a branch has an ordered XPE predicate that can be expressed using a forward-axis (e.g. some branch (b_1) has to be matched before other branch (b_2)), it is evaluated by checking if b_1 has already been evaluated as true when we receive endContext event of $b2$. Some branches that have an ordered XPE predicate expressed using a reverse-axis (parent, preceding, and so forth) can be re-written into equivalent reverse-axis-free ones by rewriting the XPE [17].

For example, consider the query expressed in Figure 1, 'search authors of such books that are published in 1999 and whose title contains XML'. We decompose the XPE into four single-path XPEs (Figure 2) and register them in the single-path XPE processor. When we receive endContext($Z), the [@year=1999] branch is evaluated as true. When we receive the set of, endContext($Z), endContext($U), and endContext($X), their parent XPE ($Y) is evaluated as true. At the same time, since the XPE ($Y) is a root, the original branching XPE is evaluated as true.

In addition, the branching XPE algorithm needs stack control; each stack indicates a context. When we receive a startContext event of a non-leaf XPE, we push a new context to check its branch XPEs. When we receive an endContext event of a non-leaf XPE, we pop the current context. Thus, each non-leaf XPE has a stack. Figure 7 shows all of the above algorithms.

One optimized implementation technique for the conjunctive evaluation restricts the number of child branches; it can be implemented as bit mask operations. We used this approach in our experiments.

```
Algorithm startContext(var)
Input: var is a single-path XPE handle, and
       this function is called-back from
       the single-path XPE processor.
Output: nothing.
 1. if (var->isNonLeafXPE())
 2.     var->pushNewContext();
```

```
Algorithm endContext(var)
Input: var is a single-path XPE handle, and
   this function is called-back from
   the single-path XPE processor.
Output: nothing.
   complete is initialized at startDocument()
   and is returned by endDocument(). It
   contains a subset of Q matches to the
   input D.
 1. if (var->isNonLeafXPE())
 2.    if (var->isBranchesMatched())
 3.       if (var->isRootXPE())
 4.          complete = complete ∪ var;
 5.       else // var is not a root XPE
 6.          var->getParentXPE()->insert(var);
 7.    var->popCurrentContext();
 8. else         // var is a leaf XPE
 9.    if (var->isRootXPE())
10.       complete = complete  ∪  var;
11.    else     // var is not a root XPE
12.       var->getParentXPE()->insert(var);
```

Figure 7: Algorithm for Branching XPE Processing

Table 1: XML stream data

	XML max depth	Number of nestings		Number of elements	
		DTD	data	DTD	data
NASA	7	1	18	113	476645
NAA	31	7	7671	365	372691
TreeBank	36	29	386614	250	2437666

6. EXPERIMENTS

Our execution environment consisted of a dual Intel(R) XEON(TM) PC (CPU 2.40GHz) with 6GB main memory, running RedHat Linux 7.2. Our compiler was gcc version 2.96 with -O2 optimization option, and the XML parser was a non-validating parser [3], which is one of the fastest XML parsers (about ten times faster than the Xerces version 1.4 C++ Parser).

6.1 XML Data

We used three data sets: astronomical NASA XML [16] (data-oriented simple XML), NAA classified advertising XML [15] (data-oriented XML of moderate complexity), and linguistic TreeBank XML (document-oriented complex XML).

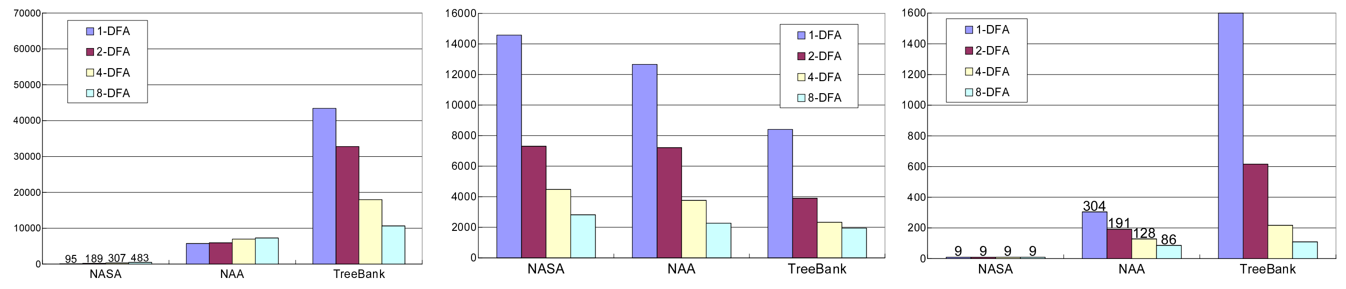
The NASA and TreeBank XMLs are real data while NAA is synthetic data created by the IBM XML data generator [4]. Table 1 shows the features of these data. The first column shows the maximum element level (depth) of XML. The NASA XML is rather shallow, whereas NAA and TreeBank XMLs are very deep. The second column shows the number of nestings in DTD and XML. The nestings of DTD indicate how many elements are recursively defined in DTD. The nestings of XML indicate how many recursive patterns were present in XML. We found that NASA data has only one recursive element in its DTD and 18 patterns in its XML. The NAA XML has a moderate number of recursive patterns. The TreeBank XML has a huge number of recursive patterns. The last column shows the number of element declarations in DTD and start tags in XML.

6.2 n-DFA and shared NFA state table results

n-DFA result

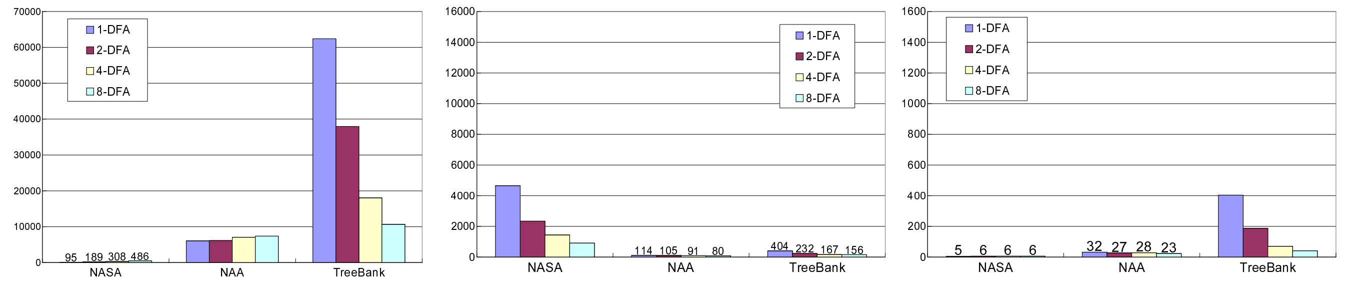
We conducted automata experiments on *n-DFA* (n=1,2,4,8). We used all three data sets and generated 100,000 (100K) XPEs using

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



(a) Number of DFA states (b) Average NFA state table size (c) Memory usage

Figure 8: Evaluations of n-DFA; XPEs 100K, prob(//) 5%, prob(*) 5%



(a) Number of DFA states (b) Average NFA state table size (c) Memory usage

Figure 9: Evaluations of n-DFA with Shared NFA state table technique: same settings as in Figure 8

the XPE generator from yfilter [7] with the probability of $*$ node test and $//$ axis each set to 5%. Figure 8 depicts (a) the total number of DFA states in the n DFAs, (b) average exclusive NFA state table size in DFA states, and (c) memory usage of all n DFAs. In (a), the number increases proportionally for NASA XML, and there is no improvement for NAA XML; This is because the XMLs are not very complex. For TreeBank XML, the number decreases linearly, because there are fewer DFA states of lazyDFA than those of eagerDFA (In the case of eagerDFA, we expect the number would decrease exponentially). In (b), the average NFA state table size decreases linearly for all data sets. In (c), we see the excellent result that the memory usage has been reduced 8 fold in 4-DFA and 16 fold in 8-DFA for TreeBank XML (compared with 1-DFA). For NASA XML, the memory usage is 9MB for all *n-DFA*s, because the NFA state table is simply split. We also found a linear improvement for NAA XML.

Figure 10 depicts TreeBank XML throughput experiments on *n-DFA* for the warm-up and stable phase. The warm-up phase is when the DFA states in lazyDFA are constructed, and the stable phase is when there is no need for DFA state construction (all needed DFA states have been already constructed). As n becomes larger, all DFAs become smaller in TreeBank, so the warm-up throughput increases. In the stable phase, the throughput linearly decreases with n, because it must deal with n DFAs.

n-DFA and shared NFA state table combined result

Next, we conducted the same experiment on *n-DFA* (n=1,2,4,8), but this time using the *shared NFA state table* technique. Figure 9 shows the results. The difference between Figure 9 and Figure 8 confirms the efficiency of the *shared NFA state table* technique. In terms of the number of DFA states (a), only TreeBank XML shows an increase in the number of DFA states, because of the penalty of the low cost DFA state equality check. In terms of average NFA state table size (b), we found a large improvement in all XML data sets which indicates many *shared NFA state table*s were shared among many DFA states. In terms of memory usage (c), we also found a large improvement in NAA and TreeBank XML (9MB becomes 6MB in NASA XML). Especially in TreeBank XML, compared with Figure 8, memory usage is only 1/4, 1/3, 1/3 and 2/5 for 1-DFA, 2-DFA, 4-DFA and 8-DFA, respectively. As a result, by combining *n-DFA* with *shared NFA state table*, 8-DFA with *shared NFA state table* sees a 40 fold reduction in memory usage.

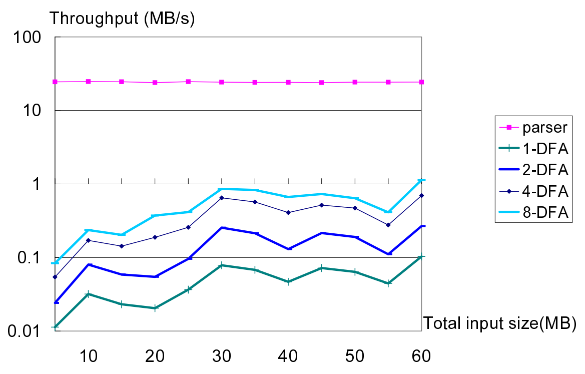
We conducted the same throughput experiments on *n-DFA* using the *shared NFA state table*. The *n-DFA* with the *shared NFA state table* was found to be slightly faster than the without case during the warm-up phase. The throughput result for the stable phase was the same as the *shared NFA state table* result.

Figure 11 depicts the results of TreeBank XML experiments on 8-DFA using the *shared NFA state table*, for which we varied the number of XPEs and * and // probability, and kept constant the other parameters, the number of XPEs (100K), and * and // probability (5%). (a) and (c) show that the large // probability for complex XML like TreeBank requires lazyDFA to have a large number of DFA states and large memory. However, in practical cases, we believe that the // probability is low (0% to 20%), so the *n-DFA* with a *Shared NFA state table* technique should be applicable to complex XML. In addition, we can control the n of *n-DFA*, to reduce memory usage.

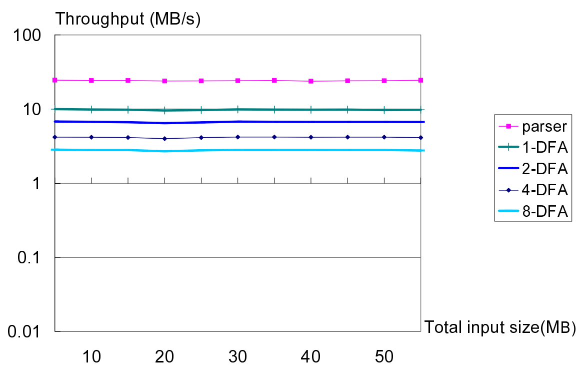
6.3 Branching XPE Processing

We conducted several experiments on branching XPE processing using NASA XML, and found that the number of branches is the only factor that affects processing performance.

Figure 12 depicts the results of throughput experiments on 100K branching XPEs with 5% $*$ node test and $//$ axis probabilities. We varied the nesting XPE probability (0%, 30% and 60%). The number of decomposed single-path XPEs were 100,000, 265,654, and 430,256, respectively. Figure 12 (a) shows the throughput of

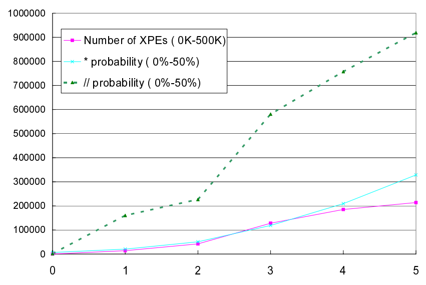


(a) Warm-up phase

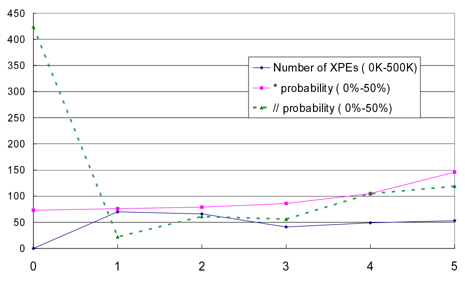


(b) Stable phase

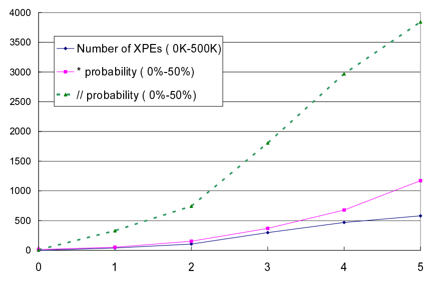
Figure 10: Throughput Experiments for n-DFA (n = 1,2,4,8)



(a) Number of DFA states



(b) Average NFA state table size



(c) Memory usage

Figure 11: Evaluations of 8-DFA with Shared NFA state table technique: TreeBank XML

lazyDFA processing plus that when invoking the startContext and endContext events. (b) shows the throughput of branching XPE processing including (a)'s processing. (c) shows the throughput of branching XPE processing excluding (a)'s processing.

LazyDFA throughput without event invocations was around 25MB/s. Figure 12 (a) shows that event invocation dominates the performance and that the throughput is dependent on the number of decomposed single-path XPEs. (a) and (c) show that branching XPE processing is as fast as event invocation and that the branching XPE processing is also dependent on the number of decomposed single-path XPEs.

7. DISCUSSION

7.1 Branch Processing with DFA

We did not extend lazyDFA to support Branching XPE (see XPush machine description in Section 8.2), because we expected the number of DFA states would grow exponentially with the number of branches. For example, if we were to add a new XPE with a branch (e.g. /bib[@domain="database"]/book) to a lazily constructed DFA, the number of DFA states would almost double because each DFA state (except a DFA state transitive by the root bib element) must remember whether the predicate was evaluated as true or false.

8. RELATED WORK

8.1 Single-path XPE Processing

NFA-based algorithms (XFilter [2], YFilter [7], and XTrie [4]) have been proposed as efficient algorithms to process a large number of XPEs in XML streams. They have a memory space guarantee that is proportional to the size of all XPEs; however their performance linearly falls with the number of XPEs. An optimization in XFilter, called list balancing, can increase throughput by factors of two to four. XTrie identifies common strings in the XPEs and organizes them in a Trie. At run-time, an additional data structure is maintained in order to keep track of the interaction between substrings. The throughput of XTrie is about two to four times higher than that of XFilter with list balancing.

Ives et al. [12] describes a general-purpose XML query processor based on eagerDFA that is efficient for a small number of XPEs (e.g. an XQuery expression can have around twenty XPEs at most). The lazyDFA paper of Green et al. [9] proves that eagerDFA is not applicable to a large number of XPEs. The throughput of lazyDFA is constant because the element/attribute transition table is implemented as a hash algorithm. Thus, lazyDFA outperforms other NFA-based algorithms by factors up to 10,000 for NASA XML.

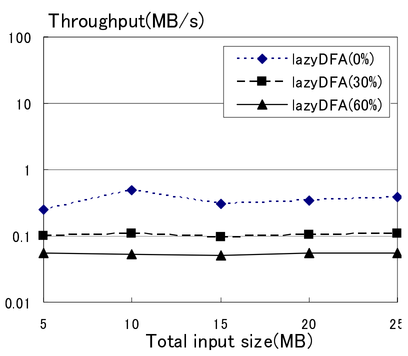
Olteanu et al. [17] use rewriting rules to transform location paths with reverse axes, such as **ancestor** and **preceding** into equivalent reverse-axis-free ones. This enables efficient SAX-based streaming processing of XPEs.

8.2 Branching XPE Processing

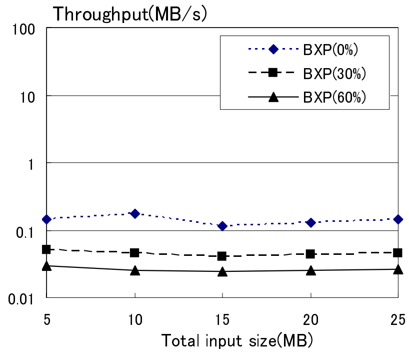
XTrie [4] processes branching XPEs and yields optimization by skipping redundant matching. The cost of our algorithm for each SAX event from the XML parser is

```
(number of matching single-path XPEs)*{
   O(Alg. startContext)|O(Alg. endtContext)}
 = |ST|*{O(pushNewContext)|O(isBranchsMatches)}
 = |ST|*{O(1)|O(1)}
 = O(|ST|)
```

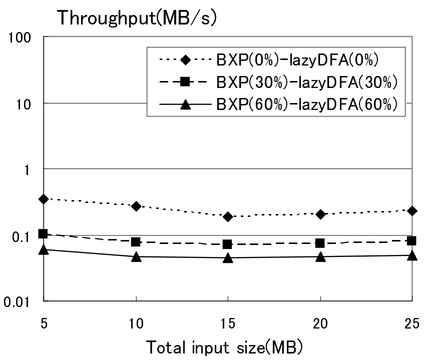
where ST is the set of related single-path XPEs. XTrie's branching processing cost is



(a) lazyDFA Processing + event invocations



(b) Branching XPE Processing



(c) Difference; (b) - (a)

Figure 12: Throughput Experiments for branching XPE Processing; XPEs 100K, prob(//) 5%, prob(*) 5%

```
O(MATCH-SUBSTRING for non-leaf substring)*
    O(FIND-PARENT-LEVEL)
  = O(|ST2|)*O(Lmax)
```

where $ST2$ represents the substrings for which each single-path XPE in ST is divided with * and //; It is obvious that $|ST| \leq |ST2|$. Lmax is the maximum number of levels in the incoming XML stream, and the O(Lmax) corresponds to the cost for connecting the divided substrings with * and //. Thus, our branching XPE processing algorithm is faster than XTrie's.

XPush machine [10] is a bottom-up DFA-based branching XPE processing algorithm with several optimizations; 1) state pruning by additional top-down processing, and 2)generating the DFA from training data. Their experiment implies that even if the data is simple, like NASA, the number of DFA states in the XPush machine reaches 100,000. It would be difficult to apply the XPush machine to more complicated XML such as NAA and TreeBank XML.

9. CONCLUSION

This paper described techniques for XPath expression processing based on DFA for two purposes; to improve memory usage of the automata and to support the processing of branching XPath expressions. The *n-DFA* and *shared NFA state table* techniques contribute to reducing the memory requirements of DFA, especially when processing complex XML. The *optimized NFA conversion* and *general XPE processing algorithm* techniques allow branching XPEs to be efficiently processed. Our experiments show that memory usage in 8-DFA with a *shared NFA state table* can be reduced 40 fold from the original 1-DFA. We can set the n of *n-DFA* according to the available memory. Although the throughput falls linearly as we increase n, it is still constant with respect to the number of XPEs; it thus outperforms other NFA-based algorithms. The branching XPE processing algorithm is also faster than earlier approaches.

10. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

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

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

[4] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of ICDE*, 2002.

[5] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *Proceedings of SIGMOD*, pages 379–390, 2000.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. The MIT Press, 1991.

[7] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings of ICDE*, 2002.

[8] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of VLDB*, pages 436–445, September 1997.

[9] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proceedings of ICDT*, pages 173–189, 2003.

[10] A. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proceedings of SIGMOD*, pages 419-430, 2003.

[11] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation (Second Edition)*. Addison-Wesley, 2001.

[12] Z. Ives, A. Halevy, and D. Weld. Efficient evaluation of regular path expressions on streaming XML data. Technical Report UW-CSE-2000-05-02, 2000.

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

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

[15] NAA classified advertising standards task force. http://www.naa.org/TECHNOLOGY/CLSSTDTF.

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

[17] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: Looking forward. In *Proceedings of Workshop on XML Data Management (XMLDM)*, LNCS. Springer, 2002.

[18] J. Pereira, F. Fabret, H.-A. Jacobsen, F. Llirbat, and D. Shasha. Webfilter: A high-throughput XML-based publish and subscribe system. In *The VLDB Journal*, pages 723–724, 2001.

[19] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh based content routing using XML. In *Symposium on Operating Systems Principles*, pages 160–173, 2001.