XMLTK: An XML Toolkit for Scalable XML Stream Processing

Iliana Avila-Campillo* Todd J. Green† Ashish Gupta* Makoto Onizuka‡
Demian Raven* Dan Suciu*

Abstract

We describe a toolkit for highly scalable XML data processing, consisting of two components. The first is a collection of stand-alone XML tools, s.a. sorting, aggregation, nesting, and unnesting, that can be chained to express more complex restructurings. The second is a highly scalable XPath processor for XML streams that can be used to develop scalable solutions for XML stream applications. In this paper we discuss the tools, and some of the techniques we used to achieve high scalability. The toolkit is freely available as an open-source project.

1 Introduction

We describe a toolkit for highly scalable XML data processing. The toolkit has two components. The first is a collection of stand-alone tools that perform simple XML transformations (sorting, aggregation, nesting, unnesting, etc) and that can be chained to express more complex restructurings. The second is a highly scalable XPath processor for XML streams that can be used to develop scalable solutions for XML stream applications. The toolkit is an open-source project at `http://xmltk.sourceforge.net`.

Our project has two goals. The first is to provide in the public domain a collection of stand-alone XML tools, in analogy with Unix commands for text files. Each tool performs one single kind of transformation, but can scale to arbitrarily large XML documents in, essentially, linear time, and using only a moderate amount of main memory. There is a need for such tools in user communities that have traditionally processed data formatted in line-oriented text files, such as network traffic logs, web server logs, telephone call records, and biological data. Today, many of these applications are done by combinations of Unix commands, such as `grep`, `sed`, `sort`, and `awk`. All these data formats can and should be translated into XML, but then all the line-oriented Unix commands become useless. Our goal is to provide tools that can process the data after it has been migrated to XML.

Our second goal is to study highly efficient XML stream processing techniques. The problem in XML stream processing is the following: we are given a large number of boolean XPath expressions and a continuous stream of XML documents and have to decide, for each document, which of the XPath expressions it satisfies. In stream applications like publish/subscribe [2] or XML packet routing [15] this evaluation needs to be done at a speed comparable with the network throughput, and scale to large numbers of XPath expressions (say $10^4 - 10^6$). DOM-based approaches typically take too long to parse, and in current XPath processors performance decreases linearly with the number of XPath expressions. Our approach, described in detail in [10], is to use a SAX parser and a lazy deterministic automaton, DFA. This results in a constant throughput, independent of the number of XPath expressions. While this may sound counter-intuitive, it essentially trades time for space, since a DFA takes a constant amount of time to process one SAX event, but its number of states may grow very large. The work in [10] performs a theoretical study of the number of states, justifying this approach, and validates it experimentally for up to 10^6 XPath expressions, with an XML data throughput of about 5.4MB/s.

We report here one novel technique for stream XML processing called Stream IndeX, SIX, and describe its usage in conjunction with the stand-alone tools. A SIX for an XML file (or XML stream) consists of a sequence of byte offsets in the XML file that can be used by the XPath processor to skip unneeded portions. When used in applications like XML packet routing, the SIX needs to be computed only once for each packet, which can be done when the XML packet is first generated, then routed together with the packet. It is important here to keep the size of the SIX small, otherwise it would consume additional bandwidth: in our experiments the SIX is about 2% of the data. When used in conjunction with the stand-alone tools, one computes once a SIX file for each XML file, then all the tools running on that XML file will automatically run faster.

*University of Washington
†Xyleme (work done at UW).
‡NTT Cyber Space Labs, NTT Corp (work done at UW).

```
<dblp>
 <book key="books/oreilly/HaroldM01">
  <author>Elliotte Rusty Harold</author>
  <author>W. Scott Means</author>
  <title>XML in a Nutshell</title>
  <publisher>O'Reilly</publisher>
  <year>2001</year>
  <isbn>0-596-00058-8</isbn>
 </book>
 <inproceedings key="conf/www/Devillers01">
  <author>Sylvain Devillers</author>
  <title>XML and XSLT Modeling for Multimedia
         Bitstream Manipulation.</title>
  <year>2001</year>
  <booktitle>WWW Posters</booktitle>
  <ee>http://www10.org/cdrom/posters/1112.pdf</ee>
  <url>db/conf/www/www2001p.html#Devillers01</url>
 </inproceedings>
 <inproceedings key="conf/webdb/HosoyaP00">
  <author>Haruo Hosoya</author>
  <author>Benjamin C. Pierce</author>
  <title>XDuce: A Typed XML Processing Language
          (Preliminary Report).</title>
  <pages>111-116</pages>
  <year>2000</year>
  <booktitle>WebDB (Informal Proceedings)</booktitle>
  <ee>www.research.att.com/conf/webdb2000/PAPERS/7c.ps</ee>
  <url>db/conf/webdb/webdb2000.html#HosoyaP00</url>
 </inproceedings>
 <article key="journals/cn/GirardotS00">
  <author>Marc Girardot</author>
  <author>Neel Sundaresan</author>
  <title>Millau: an encoding format for efficient representation
        and exchange of XML over the Web.</title>
  <pages>747-765</pages>
  <year>2000</year>
  <volume>33</volume>
  <journal>WWW9 / Computer Networks</journal>
  <number>1-6</number>
  <url>db/journals/cn/cn33.html#GirardotS00</url>
 </article>
</dblp>
```

Figure 1: Sample XML data from the DBLP database

We describe the stand-alone tools in Sec. 2 then describe the XPath processor in Sec. 3. Related work is discussed in Sec. 4, then we conclude in Sec. 5.

2 The Tools

2.1 Overview

The stand-alone tools currently in the XML toolkit are summarized in Fig. 2. Every tool's inputs/outputs XML stream via standard i/o, except `file2xml` which takes a directory as an input and outputs XML to the standard output.

`xsort` is by far the most complex one and we describe it in more detail. The others we only illustrate briefly, for lack of space, but note that most can be used in quite versatile ways. We shall illustrate the tools on the DBLP database [13]; a fragment is shown in Fig. 1. There are 256599 bibliographic entries in the version used in our experiments.

2.2 Sorting

The command below sorts the entries in the bib file in ascending order of their year of publication[1]:

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

The first argument, `-c`, defines the *context*: this is the collection under which we are sorting. The second argument, `-e`, specifies the *items* to be sorted under the context: for the example in Fig. 1, this matches the `book`, `inproceedings`, `inproceedings`, and `article` items. Finally, the last argument, `-k`, defines the *key* on which we sort the items. The result of this command is the file `sorted-dblp.xml` which lists the four publications in increasing order of the `year`. In the four publications in Fig. 1 `year` has values `2001`, `2001`, `2000`, and `2000`, hence the items will be listed in the output in the order 3, 4, 1, 2, since the sorting algorithm we use is stable. The input file, when omitted, defaults to the standard input.

The command arguments for `xsort` are shown in Fig. 2, with some details omitted. There can be several context arguments (`-c`), each followed by several item arguments (`-e`), and each followed by several key arguments (`-k`). The semantics is illustrated in Fig. 3. First, all context nodes in the tree are identified (denoted `c` in the figure): all nodes that are not below some context node are simply copied to the output in unchanged order. Next, for each context node, all nodes that match that context's item expressions are identified (denoted `e1, e2, ...` in the figure), and a key value is computed for each of them, by evaluating the corresponding key expressions. These item nodes are then sorted according to the key values, and output in increasing order of the keys. Notice that the nodes that are below a context, but not below an item are deleted from the output.

We show below several examples of `xsort`.

Simple sorting We start with a simple example:

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

which returns all authors, sorted by their text value. Other elements under `dblp` that are not `author` elements are erased. The result is shown in Fig. 4 (a).

Sorting with multiple key expressions The following example illustrates the use of two keys. Assuming that `author` elements have a `firstname` and a `lastname` subelement, it returns a list of all authors, sorted by `lastname` first, then by `firstname`:

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

Command	Arguments (fragment) P = XPath expr, N = number	Brief description
xsort	(−c P (−e P (−k P)*)*)*	sorts an XML stream
xagg	(−c P (−a aggFun valP)*)*	computes the aggregate function aggFun (see Fig. 6)
xnest	(−e P ((−k P)*) | −n N)*	groups elements based on key equality or number
xflatten	(−r)? −e P	flattens collections (deletes tags, but not content)
xdelete	−e P	removes elements or attributes
xpair	(−e P −g P)*	replicates an element multiple times, pairing it with each element in a collection
xhead	(−c P (−e P (−n N)?)*)*	retains only a prefix of a collection
xtail	(−c P (−e P (−n N)?)*)*	retains only a suffix of a collection
file2xml	−s dir	generates an XML stream for the dir file directory hierarchy

Figure 2: Current tools in the XML toolkit.

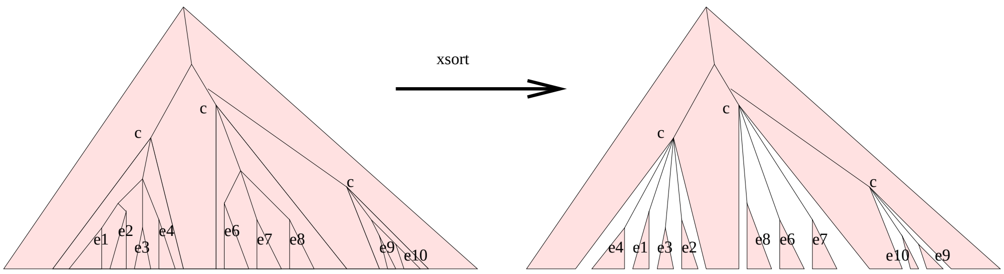


Figure 3: Semantics of `xsort`. Under each *context* node the *item* nodes are sorted based on their *key*. Any nodes that are "between" context nodes and item nodes are not copied in the output.

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

    <dblp>
      <author>Benjamin C. Pierce</author>
      <author>Elliotte Rusty Harold</author>
      <author>Haruo Hosoya</author>
      <author>Marc Girardot</author>
      <author>Neel Sundaresan</author>
      <author>Sylvain Devillers</author>
      <author>W. Scott Means</author>
    </dblp>
```

(a)

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

    <dblp>
       <article> . . . . </article>
       <article> . . . . </article>
       . . .
       <inproceedings> . . . </inproceedings>
       <inproceedings> . . . </inproceedings>
       . . .
       <book> . . . </book>
       <book> . . . </book>
       . . .
       <manuscript> . . . </manuscript>
       <incollection> . . . </incollection>
       . . .
    </dblp>
```

(b)

Figure 4: Results of various `xsort` commands.

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

Sorting with multiple item expressions When multiple `-e` arguments are present, items are included in the result in the order of the command line. For example the following command:

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

lists all `articles` first, then all `inproceedings`, then all `books`, then everything else. Within each type of publication the input document order is preserved. The output will look like in Fig. 4 (b).

Sorting at deeper contexts By choosing contexts other than the root element we can sort at different depths in the XML document. A common use is to normalize the elements by listing their subelements in a standard order. For example, consider:

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

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

    <dblp>
      <book>
        <title>XML in a Nutshell</title>
        <author>Elliotte Rusty Harold</author>
        <author>W. Scott Means</author>
        <publisher>O'Reilly</publisher>
        <year>2001</year>
        <isbn>0-596-00058-8</isbn>
      </book>
      <inproceedings>
        <title>XML and XSLT Modeling . . . </title>
        <author>Sylvain Devillers</author>
        <url>db/conf/www/www2001p.html#Devillers01</url>
        <year>2001</year>
        <booktitle>WWW Posters</booktitle>
        <ee>http://www10.org/cdrom/posters/1112.pdf</ee>
      </inproceedings>
      . . . .
     </dblp>
```

(a)

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

    <dblp>
      <book>
        <title>XML in a Nutshell</title>
        <author>Elliotte Rusty Harold</author>
        <author>W. Scott Means</author>
      </book>
      <inproceedings>
        <title>XML and XSLT Modeling . . . </title>
        <author>Sylvain Devillers</author>
      </inproceedings>
      <inproceedings>
        <title>XDuce: A Typed XML Processing . . . </title>
        <author>Haruo Hosoya</author>
        <author>Benjamin C. Pierce</author>
      </inproceedings>
      <article>
        . . .
      </article>
    </dblp>
```

(b)

Figure 5: Normalizing element order with `xsort`: with a catch-all (a), and without a catch-all (b).

This outputs, for each publication, its elements in the following order: first all `title` elements, then all `author` elements, then all `year` elements, and then everything else. The output looks like in Fig. 5(a) (attributes are omitted).

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

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

returns a result like in Fig. 5 (b).

In the last two examples the `author` order is preserved, since no key has been specified for `author`. If we want to sort authors alphabetically inside each publication, then we issue the following command:

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

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

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

(a)

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

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

(b)

Table 1: Experiments with `xsort`: a global sort (a), and multiple local sorts (b). Numbers are running times in seconds. A "-" indicates ran out of memory

Sorting with multiple context expressions Finally, multiple context arguments can be specified to sort according to different criteria. For example:

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

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

DTDs xsort and the other tools use a non-validating xml parser, and do not generate a DTD for the output data. Inferring a DTD for the transformed output data is a complex that we don't address.

Implementation We have optimized `xsort` to scale up efficiently to large XML streams. We sort one context at a time, copying the other elements to the output file in unchanged order. When sorting one context, we create a *global key* for each item to be sorted, consisting of the item identification number on the command line, the concatenation of all its keys, and its order number under the current context (to make `xsort` stable). We use multiway merge-join, with as much main memory as available, and stop after at most two steps. The first step produces the initial runs, using STL's priority queue [3], and applying replacement selection [9]. This results in initial runs that may be larger than main memory: in particular, a single run is produced if the input is already sorted. If more than one run is generated then a second step

valP (from Fig. 2)	type	meaning
int	number	text() interpreted as integer
float	number	text() interpreted as float
text	text	text() interpreted as string
depth	number	the depth of the current element

aggFun (from Fig. 2)	type	meaning
count	any	counts the elements
sum	number	sum value
	text	concatenates the values
max	number	maximum value
min	number	minimum value
avg	number	average value
first	any	returns the first data value found
last	any	returns the last data value found
choice#342	any	returns the 342nd data value, or 0 if out-of-bound

Figure 6: Details of the `xagg` command.

is executed, which merges all runs to produce the final output. With today's main memories, practically any XML file can be sorted in only two steps. For example, with 128MB of main memory and disk pages of 4KB, we can sort XML streams of up to 4TB [7], and the file size increases quadratically with the memory size. More practical considerations, such as a hard limit of 2GB on file sizes on most systems, or limits on the number of file descriptors, are more likely to limit the size of the largest file we can sort.

Experiments Two sets of experiments[2] are shown in Table 1, where we compare `xsort` with `xalan`, a publicly available XSL processor. For `xsort` we limit the main memory window to 32MB. The first represents a global sort which reorders all bibliographic entries: `xsort`'s running time increases linearly, with the exception of an extra factor of two, when the data size exceeds the memory size. The second table represents local sorts, with small contexts. Here a single pass over the data is always sufficient, and the sorting time increases linearly. `xalan`'s processing model is DOM-based, and supports a more general class of transformations (including joins).

2.3 Other Tools

All the other tools are designed to do a single pass over the XML data; we illustrate them here only briefly. Some are straightforward, like `xdelete`; others are quite versatile, like `xagg`, but we omit more interesting examples for lack of space.

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

Aggregation The `xagg` command line is given in Fig 2, while some details of the `-a` argument are given in Fig. 6. We illustrate it here with three examples:

```
xagg -c /dblp -a count text *
xagg -c /dblp -a count text *
  -a count text */author -a avg float */price
xagg -c /dblp/* -a first text title
  -a count text author -a count text url
```

The first example counts the total number of publications under `dblp`. Its result is:

```
<xagg>
  <context path="/dblp">
    <agg type="count" path="*">256599</agg>
  </context>
</xagg>
```

That is, there are 256599 bibliographical entries in the dblp data. The tags `xagg`, `context`, and `agg` are chosen by default and can be overridden in the command line.

The second computes two aggregate functions: the total number of elements, and the average value of `price` (assuming some publications have a numeric `price` subelement). Its result will look like in Fig. 7 (a): this is a hypothetical result, in reality the dblp data does not contain prices.

The third computes two aggregate functions for each publication: the *first* `title` element and the number of authors. The result will have the form shown in Fig. 7 (b). There will be as many `context` elements in the result as publications in the input data.

Collection-oriented operations The toolkit contains a few collection-oriented tools, inspired from [4]: `xnest`, `xflatten`, `xpair`, and `xdelete`. The `xdelete` command simply deletes elements matching one or several XPath expressions. `xflatten` flattens a nested collection; equivalently, it deletes only the tags, but not the content. For example:

```
xflatten -e //b
```

transforms the input XML document as follows:

```
from:                                      to:
<a> <b> <c> </c>                           <a> <c> </c>
        <d> </d>                               <d> </d>
        <b> <e> </e> </b>                      <b> <e> </e> </b>
    </b>
    <c> <d> </d> </c>                          <c> <d> </d> </c>
    <c> <b> <e> </e> </b> </c>                 <c> <e> </e> </c>
</a>                                       </a>
```

Only the two top-most `b` tags are deleted: the flag `-r` specifies recursive flattening. `xnest` groups multiple adjacent elements under a new collection: in other words, it inserts new tags in the XML document, without erasing anything. For example:

```
xagg -c /dblp -a count text * -a count text */author
                              -a avg float */price

<xagg>
  <context path="/dblp">
    <agg type="count" path="*">256599</agg>
    <agg type="count" path="*/author">548856</agg>
    <agg type="avg" path="*/price">44.4503945</agg>
  </context>
</xagg>
```

(a)

```
xagg -c /dblp/* -a first text title
                -a count text author
                -a count text url
<xagg>
  <context path="/dblp/*">
    <agg type="first" path="title">XML in a Nutshell</agg>
    <agg type="count" path="author">2</agg>
    <agg type="count" path="url">0</agg>
  </context>
  . . .
</xagg>
```

(b)

Figure 7: Results of various `xagg` commands.

```
xnest -e /dblp/* -k year/text()
```

groups publications based on their `year` subelement. The output is illustrated in Fig. 8 (a). Here one group is created for every set of adjacent publications that have the same `year` value. Notice that there may be multiple groups with the same key value, like 2001 above: to have unique groups, one needs to sort first. Multiple keys can be specified, like in `xsort`. If no key is specified then all adjacent elements are placed under the same group. There is a second variant of `xnest` that creates groups by their number of elements, see Fig. 2.

Finally, `xpair`, called `pair-with` in [4], pairs an element with each item of a collection. It corresponds to `pairwith` in [4]. For example:

```
xpair -e /a/b/c -g /a/b/d
```

replaces each occurrence of `/a/b/d` with an element `<pair> <c> </c> <d> </d> </pair>`, where the `c` element is the last it has seen before. Its effect is:

```
from
<a> <b> <c> 1 </c>
        <d> 2 </d>
        <d> 3 </d>
    </b>
    <b> <d> 4 </d> </b>
    <b> <c> 5 </c>
        <d> 6 </d>
    </b>
</a>
```

```
to
<a> <b> <c> 1 </c>
        <pair> <c> 1 </c>
               <d> 2 </d>
        </pair>
        <pair> <c> 1 </c>
               <d> 3 </d>
        </pair>
    </b>
    <b> <pair> <c> 1 </c>
               <d> 4 </d>
        </pair>
    </b>
    <b> <c> 5 </c>
        <pair> <c> 5 </c>
               <d> 6 </d>
        </pair>
    </b>
</a>
```

```
xnest -e /dblp/* -k year/text()

<dblp>
   <group> <key> 2001 </key>
           <book> . . . </book>
           <inproceeding> . . . </inproceedings>
           <inproceeding> . . . </inproceedings>
           . . .
   </group>
   <group> <key> 2000 </key>
           <inproceedings> . . . </inproceedings>
           <article> . . . </article>
           <article> . . . </article>
           <book> . . . </book>
           . . .
   </group>
   <group> <key> 2001 </key>
           . . .
   </group>
   . . .
</dblp>
```

(a)

```
file2xml -s data > output.xml

<directory>
  <name>data</name>
  <file>
    <name>file1</name>
    <filelink xlink:type="simple"
              xlink:href="file:/homes/june/suciu/data/file1">
    </filelink>
    <path>/homes/june/suciu/data/file1</path>
    <size>33</size>
    <permissions>-rw------</permissions>
    <type>regular file</type>
    <userid>13750</userid>
    <groupid>330</groupid>
    <lastAccess>Wed Nov 21 11:22:33 2001</lastAccess>
    <lastModification>Wed Nov 21 11:22:23 2001</lastModification>
  </file>
  ...
</directory>
```

(b)

Figure 8: Illustration of `xnest` and `file2xml`.

Heads or Tails? `xhead` and `xtail` select and output the head or tail of a sequence of elements matching one or several XPath expressions. For example:

```
xhead -c /dblp -e book -n 20 -e article
```

outputs only the first 20 `book` elements and the first 10 (default value) `article` elements under `dblp`.

File Directories to XML The `file2xml` generates an XML stream that describes a file directory hierarchy. For example:

```
file2xml -s data > output.xml
```

traverses the `data` directory and all its subdirectories and creates the `output.xml` document which has an isomorphic structure to the directory hierarchy. The output is shown in Fig. 8 (b).

As another example, the command below lists the top ten largest files in a directory hierarchy:

```
file2xml -s . | xsort -b -c /directory
     -e //file -k size/text():%i |
     xhead -c /directory -e file
```

The `%i` option in `xsort` indicates that `size` is an integer field.

2.4 Putting Them Together ...

The power of the toolkit comes from pipelining several simple tools, to do complex transformations. Since each individual tool was designed to scale up to very large XML documents, this programming style allows programmers to do complex transformations on very large XML streams, that go beyond the capabilities of today's XML engines.

Consider the following classical query: re-group publications by author, rather than title. That is, we want one element for each distinct author in the database, followed by all titles she published. This is achieved with:

```
xsort -c /dblp/* -e title -e author |
xpair -k /dblp/*/title -g /dblp/*/author |
xflatten -e /dblp/* |
xpair -c /dblp -e title -e author |
xflatten -c /dblp/* |
xsort -c /dblp -e pair -k author/text() |
xnest -e /dblp/pair -k author/text()
```

This is rather standard processing of nested collections. First, normalize all entries by listing the title first, then the author(s). Next, pair each title with all the authors, then flatten the collection: now we have a flat list of (`title`, `author`) pairs. Next sort on `author`, and finally nest on the `author`.

Viewed as a "language", this is closer to a physical algebra than to a declarative language like XQuery [5] or a functional language like XDuce [11]. Our purpose is not to supersede high-level languages, but rather to allow sophisticated users to combine the tools in order to process large XML streams.

2.5 ... and Making it Run Even Faster

We have provided two mechanisms for further speeding up the toolkit: a binary format for XML, and a Stream IndeX (SIX).

The Binary Format Our binary XML format (1) replaces tags and attributes with integers called *tokens*, and (2) recognizes some atomic data types like integers, reals, and represents them in binary. While other binary formats exist already [14, 8], ours was designed specifically for XML *data* applications[3]. We also define the *tokenized SAX (TSAX)* events for XML parser whose parameter is tokenized as above. For example, `startElement('book')` becomes in TSAX `startElement(5)` with the value '5' corresponding to 'book'.

[3]We do not compress texts, and we have specialized binary datatypes like integers.

The TSAX offers a uniform interface to both standard XML and the binary XML. Each tool accepts either standard XML or binary XML as input, and can produce standard XML or binary XML as output. The input is automatically recognized; for the output, the user needs to specify a `-b` command argument, if she wants to emit binary XML as output. The binary format reduces the size of the data by roughly a factor of two, and this usually translates into a speedup factor of two, less so for long pipelines. For example, on a 98 MB input file, the following:

```
xcat dblp.xml | xcat | xcat | xcat >/dev/null
```

took 59 seconds to execute, while the same pipeline using binary throughout

```
xcat -b dblp.bin | xcat -b | xcat -b |
                   xcat -b >/dev/null
```

took 37 seconds to execute. Introducing an additional stage of `xcat` in the two pipelines increased the execution times by 14.39 and 9.59 seconds respectively, or 1.5 times better for the binary format. Here `xcat` is a tool used mainly for testing which simply parses the XML input then outputs it.

The Stream IndeX (SIX) Given an XML stream, a SIX is a binary stream consisting of pairs of the form (`beginOffset`, `endOffset`). There is one pair for each XML element. Here `beginOffset` is the byte offset of the begin tag, and `endOffset` of the end tag. The SIX is sorted by `beginOffset`, allowing it to synchronized with the XML stream. The XPath processor matches SIX entries with the tags in the input XML stream and, if it decides that the current XML element is not needed then it uses `endOffset` to skip characters in the XML stream without ever parsing the content. The larger the portion in the XML document that it skips, the greater the performance it gains. It follows that SIX entries corresponding to small XML elements offer little benefits, and can be deleted: this reduces the size of the SIX, further increasing the performance.

To illustrate, a SIX is created as in the following example:

```
createSindex -t 100 dblp.xml > dblp.six
```

This creates a binary file `dblp.six` that is the SIX for the XML file `dblp.xml` and whose SIX entry is only for element larger than 100bytes. Consider now a simple command, like:

```
xagg -c /dblp -a count text book   dblp.xml
```

that counts the total number of `book` elements. If the system finds the corresponding SIX file, called `dblp.six`, then it uses it to skip portions of the XML file. In this particular example it can skip the content of *all* bibliographic entries, hence only the first two levels of the XML tree need to be parsed. On the entire 98MB dblp database this command ran in 14.7 seconds without a SIX, in 2.4 seconds with a full SIX (a factor of 6.125), and in 2.0 seconds with a 100byte element deleted SIX (a factor of 7.35). This is because the size of the full SIX was about 20% that of the data, while the reduced SIX was only 2% that of the data. Note that the SIX is only useful in the first stage of the pipeline. In principle, a SIX could be produced incrementally at each stage and interleaved with the output XML, but we did not implement such a scheme.

3 The XPath Processor

We describe now the second component of the XML Toolkit: the XPath processor for stream-based XML applications. The processor is designed to evaluate large sets of XPath expressions on an input XML stream, and has a C-based API. The architecture is shown in Fig. 10. All tools described in Sec. 2 use this API to evaluate the XPath expressions in their command line.

The stream API defines a simple event-based XML processing model that extends the tokenized SAX parsing model. A "query" is given by a tree, called the *query tree*, with nodes labeled with *variables* and edges labeled with XPath expressions. We illustrate with an example using `xsort`:

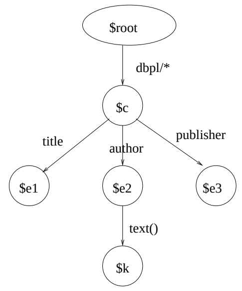
```
xsort -c /dblp/* -e title
    -e author -k text()  -e publisher
```

The query tree that will be registered with the API is shown in Fig. 9 (a) and (b).

The XPath processor's role is to identify when a match of the variable with the input XML stream occurs. TSAX events, plus the new *variable match* events are then forwarded to the application. For illustration, a possible sequence of TSAX and context events sent to the application is shown in Fig. 9 (c).

```
$c in $root/dblp/*
$e1 in $c/title
$e2 in $c/author
$k  in $e2/text()
$e3 in $c/publisher
```

(a)



(b)

```
startVariable($root)
startDocument()
  startVariable($c)
  startElement('book')
    startVariable($e2)
    startElement('author')
      startVariable($k)
      characters('Elliotte Rusty Harold')
      endVariable($k)
    endElement('author')
    endVariable($e2)
    startVariable($e2)
    startElement('author')
    ...
    endElement('author')
    endVariable($e2)
    startVariable($e1)
    startElement('title')
      characters('XML in a Nutshell')
    endElement('title')
    endVariable($e1)
    startVariable($e3)
    startElement('publisher')
      characters('O'Reilly')
    endElement('publisher')
    endVariable($e1)
  endElement('book')
  endVariable($c)
endDocument()
endVariable($root)
```

(c)

Figure 9: A query tree in XPath notation (a) and in graphical representation (b), and a sequence of SAX and variable-match events for this tree (c).

3.1 The Tokenized SAX

We have modified the SAX interface in a few ways. First, all tag and attributes are translated into integers, as explained in Sec. 2.5. This is consistent with our binary XML format, and results in slight performance improvements for applications that need to perform many comparisons between tags. For the XPath processor only, however, it results in a small performance penalty when the input is a plain XML file, since the additional tokenization step involves one extra hash-table lookup.

Second, we have defined in TSAX an event for every individual attribute. This is a change from the standard SAX specification in which the `startElement` event includes its all attributes.

Finally, the TSAX API recognizes certain atomic data types. Currently we support the `extendedint` data type, which means an integer possibly preceded and/or followed by a known string, in notation `PREFIX %iSUFFIX`. For example the application can register the extended integer `USD%i` with the TSAX parser, and TSAX will translate text values like `USD 99`, `USD 1045` into 99, 1045 respectively.

3.2 The XPath Processor

The XPath processor takes the query tree and a stream of TSAX events generated by the parser and identifies the new variable events. While the tools described in Sec. 2 rarely use more than a dozen or so XPath expressions, other applications, such as publish/subscribe systems or XML packet routing often need to evaluate tens or hundreds of thousands of XPath expressions on the XML stream. Our goal was to design the processor to scale to very large numbers of XPath expressions. We only sketch here our approach, and refer the reader to [10] for details.

The processor converts the entire query tree into one single nondeterministic finite automaton (NFA), then computes the corresponding deterministic finite automaton (DFA). Assuming the DFA has already been constructed, the processor simply keeps a pointer to the current state. On a `startElement` event, the processor looks up the next current state in the DFA, and pushes the old state on a stack. On a `endElement` event, the processor pops a state from the stack and set the popped state as the current state. Terminal DFA states have an associated set of variables, and whenever such state is reached, one variable match event is generated for each variable in the set. The stack gets only as deep as the maximum depth of the XML document[4]. We preallocate a stack of depth 1024, and grow it automatically (by doubling the size) if needed. In practice, the initial depth of 1024 is easily deep enough for typical documents, and no additional memory management is necessary. As a consequence, the XPath processor achieves constant throughput, independent of the number of XPath expressions. The experiments in [10] show that the XML input stream can be processed at constant throughput of about 5.4MB/s, independent of the number of XPath expressions (we stopped our experiments at 10^6 XPath expressions).

The main obstacle in using a DFA is constructing it, since, in general, its number of states is exponential in the size of the NFA. Our solution is to construct the DFA lazily. Real XML data tends to nest elements in a predictable fashion, for example as imposed by a DTD or an XML Schema, and the consequence is that the number of states that ever need to be expanded in the lazy DFA is very small. This statement is made precise theoretically, then validated experimentally in [10]. Hence, we construct the DFA lazily. Initially, there is a warm-up phase, when most of the lazy DFA states are expanded, during which the throughput is significantly lower: the length of this phase depends both on the number of the XPath expressions and the complexity of the input XML data. After the warm-up, the throughput reaches its maximum speed.

Currently we support some limited XPath filters: position predicate and any predicate expressions only with an attribute location step. For example, we support:

```
//article[@year>1998]
         [contains(@type,'proceedings')]/title
```

but not support

```
//article[booktitle/text()='ACM SIGMOD']
```

We don't implemented some output buffering scheme in the lazy DFA, so the tail location step can not have a predicate.

The XPath processor has two additional features, that we discuss next.

Echo control The application can indicate that it doesn't need the TSAX events in a certain part of the query tree. This is called *echo control*. Of course, applications could filter the unwanted TSAX events by themselves, but it is important to let the XPath processor know this in order to use a SIX: if *all* TSAX events need to be forwarded to the application, then no portion of the XML file can ever be skipped and the SIX is useless.

Precedence If several variables match the same XML element, then the XPath generates all corresponding events. Some applications, however, require a different semantics, in which variables are evaluated "in order". As a feature, the XPath processor accepts an optional precedence parameter for each variable in the query tree, and only generates events corresponding to the highest matched variable(s). Most tools described in Sec. 2 uses this feature. For example the following `xsort` command:

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

matches subelements of `book` with the `-e author` expression first, and only if there is no match tries to match them with `-e *`. When the xsort module registers these two expressions with the XPath processor it will specify that the former has a higher precedence than the latter.

3.3 The SIX Manager

If a SIX is present, then portions of the XML stream can be skipped using the offsets in the SIX. This is handled by the SIX manager, see Fig. 10, which

[4] The depth is a sum of the number of element and its attributes

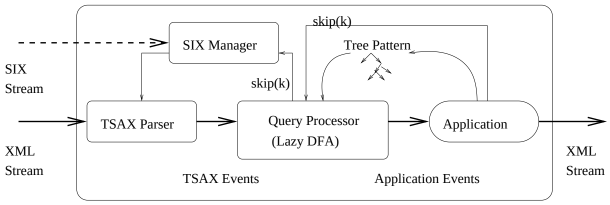


Figure 10: The System Architecture

exposes a single function in its interface: `skip(k)`, meaning "skip the input stream to the end of the k's open tag". For example, `skip(0)` means skip to the end of the current open tag, `skip(1)` means skip to the end of the parent tag, etc. The XPath query processor uses the SIX as follows. When a `startElement` is received from the SAX parser for which there is no transition from the current DFA state, then it issues a `skip(0)` command. Applications can also issue `skip(k)` commands, if they can determine that k of the currently open elements are no longer needed. For example, if it looks for books published after 1977, then, after seeing a `book` element, then a `year` element whose value is 1950 it may issue a `skip(1)` command if it "knows" that the `book` has at most one `year` subelement. Such information is readily available from a DTD, for example.

4 Related Work

The work closest to our toolkit is LT XML, from `http://www.ltg.ed.ac.uk/software/xml/`. It defines a C-based API for processing XML files, and builds a large number of tools using this API. Their emphasis is on completeness, rather than scalability: there is a rich set of tools for searching and transforming XML files, including a small query processor. There exists a sort utility but with much more restricted functionality than our `xsort`. No details about the processing techniques are given.

Two XML stream processing techniques have been proposed: XFilter [2] and XTrie [6]. Both are highly optimized nondeterministic finite automata, and their throughput decreases with the number of XPath expressions. Our lazy-DFA technique achieves throughputs that are between 100 times and 10,000 times faster than XFilter for large number of XPath expressions [10].

Binary XML formats are considered in [14, 8, 1].

To our best knowledge the SIX is the first attempt to index streaming data. Related in spirit, but different in means is "indexing on the air" [12], where the issue is to allow receivers to save power when downloading data from a broadcast channel.

5 Conclusions

We have described a highly scalable toolkit for processing XML data, which is now freely available software in the public domain. Our main emphasis was on techniques that achieve scalability: processing large numbers of XPath expressions on XML streams, indexing XML streams, and efficient sorting.

Acknowledgment This project was partially supported by Suciu's NSF CAREER Grant 0092955, a gift from Microsoft, and an Alfred P. Sloan Research Fellowship.

References

[1] R. Agrawal, R. J. B. Jr., D. Gruhl, and S. Papadimitriou. Vinci: a service-oriented architecture for rapid development of web applications. In *Proceedings of Word Wide Web Conference*, pages 385–365, 2001.

[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] ANDIS/ISO. *C++ Standard*, 1998.

[4] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[5] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: a query language for XML, 2001. available from the W3C, `http://www.w3.org/TR/query`.

[6] 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.

[7] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, Upper Saddle River, New Jersey 07458, 2000.

[8] M. Girardot and N. Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the WWW. In *International World Wide Web Conference*, May 2000.

[9] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[10] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes, 2002. manuscript. http://www.cs.washington.edu/homes/suciu/files/_F2066943700.ps.

[11] H. Hosoya and B. C. Pierce. XDuce: An XML processing language (preliminary report). In *WebDB'2000*, pages 226–244, 2000. http://www.research.att.com/conf/webdb2000/.

[12] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Energy efficient indexing on air. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 25–36. ACM Press, 1994.

[13] M. Ley. Computer science bibliography (dblp). `http://dblp.uni-trier.de`.

[14] B. Martin and B. Jano. WAP binary XML content format, 1999. available from the W3C, `http://www.w3.org/TR/wbxml`.

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