Summary Structures for XML

David Malan

TR-05-04

2004

Computer Science Group
Harvard University
Cambridge, Massachusetts

Summary Structures for XML

David Malan
malan@eecs.harvard.edu

1 Introduction

The second of ten goals pursued by the World Wide Web Consortium (W3C) in designing the eXtensible Markup Language (XML) was "to support a wide variety of applications" [26]. Since its introduction in 1998, XML has been deployed as a format for the delivery and storage of hierarchical data, as a declarative language for stylesheets, as a transport mechanism for remote procedure calls, and more. It is on the first of these applications that this paper focuses. In particular, this work explores weaknesses in the W3C's prescription for the serialization and navigation of XML and offers novel remedies through summary structures.

This paper lays the groundwork for its exploration in Section 2 with an overview of XML. Section 3 exposes inefficiencies in the serialization and navigation of XML, while Sections 4 and 5 propose remedies for the same. Section 6 makes mention of related work. Section 7 presents areas for further study. Section 8 concludes.

2 Background

Despite its name, XML is actually a metalanguage with which tag-based markup languages can be defined. As such, XML enables documents to be self-describing, their content *tagged* hierarchically with semantic metadata. A derivative of SGML not unlike HTML, XML allows data to be annotated with elements, attributes, processing instructions, and comments. Although XML requires that documents be *well-formed* (*i.e.*, syntactically valid), conformance to (*i.e.*, validity with respect to) a document type definition (DTD) [26] or an XML schema [27] is optional. Programmatic access to an XML document is commonly provided to an application by the Simple API for XML Processing (SAX) [14], which hands to the application a series of events (each triggered by the parsing of some portion of the document), or by the Document Object Model (DOM) [23], which returns to the application the root of an ordinal tree whose nodes capture the document's structure. Whereas SAX is appropriate for applications whose needs for a document's content are fleeting, DOM is appropriate for applications that require traversals of or non-serialized access to a document's structure. Navigation of an XML document's structure is facilitated by the XML Path Language (XPath) [24, 28] and the XML Query Language (XQuery) [29].[1]

2.1 XML

XML requires that a single, *root* element (delimited by a *start-tag* and an *end-tag*) surround a document's data, portions of which may be further tagged with elements, so long as those elements

[1] Although XPath 1.0 has been an official recommendation from the W3C since 1999, XPath 2.0 and XQuery 1.0 are not yet finalized.

are properly nested. In addition to zero or more descendants, an element may have zero or more attributes, each of whose values must be flanked by a pair of quotation marks. An XML document may additionally contain comments and processing instructions (which provide direction to humans and applications, respectively), neither of which need be encapsulated by the document's root element. In fact, an XML document may also contain as its first line an *XML declaration*, which specifies the document's encoding and the version of XML with which it complies.

Although the W3C's recommendation specifies additional requirements for XML, `database.xml` (Figure 1) can be said to be a representative document.

```
<?xml version="1.0" encoding="UTF-8"?>
<?application data?>
<!-- my database -->
<database>
  <actors>
    <actor id="00000001">Kaleil Isaza Tuzman</actor>
    <actor id="00000002">Tom Herman</actor>
    ...
  </actors>
  <movies>
    <movie id="12345678">
      <title>Startup.com</title>
      <genre>documentary</genre>
      <rating>R</rating>
      <summary>The rise and fall of the American dream.</summary>
      <details>Friends since high school, 20-somethings Kaleil Isaza Tuzman and
      Tom Herman have an idea: a Web site for people to conduct business with
      municipal governments. This documentary tracks the rise and fall of
      govWorks.com from May of 1999 to December of 2000, and the trials the
      business brings to the relationship of these best friends.</details>
      <year>2001</year>
      <director>Chris Hegedus</director>
      <studio>Artisan</studio>
      <user_rating>4</user_rating>
      <runtime>107</runtime>
      <actor ref="00000001"/>
      <actor ref="00000002"/>
      <vhs>19.99</vhs>
      <vhs_stock>106</vhs_stock>
      <dvd>26.99</dvd>
      <dvd_stock>118</dvd_stock>
    </movie>
    ...
  </movies>
</database>
```

Figure 1: An excerpt from `database.xml`, a representative XML document with 390 elements (40 of which are `actor` elements, 18 of which are `movie` elements, 18 of which are `title` elements) and 779 text nodes. Ellipses denote omitted `actor` and `movie` elements; elements of the form `<foo bar="baz"/>` denote "empty" elements (*i.e.*, elements with no children).

2.2 Modelling XML

Its first three lines aside, `database.xml` would seem to constitute a pre-order traversal of some ordinal tree, the nodes of which are this document's elements.

It is precisely this view that DOM brings to XML; though, in addition to elements, DOM also models a document's attributes, processing instructions, comments, and text as nodes in a tree, all of which ultimately descend from an implicit, nameless root (the so-called *document node*). Figure 2 depicts DOM's view of `database.xml`. The representative document, then, is but a serialization of

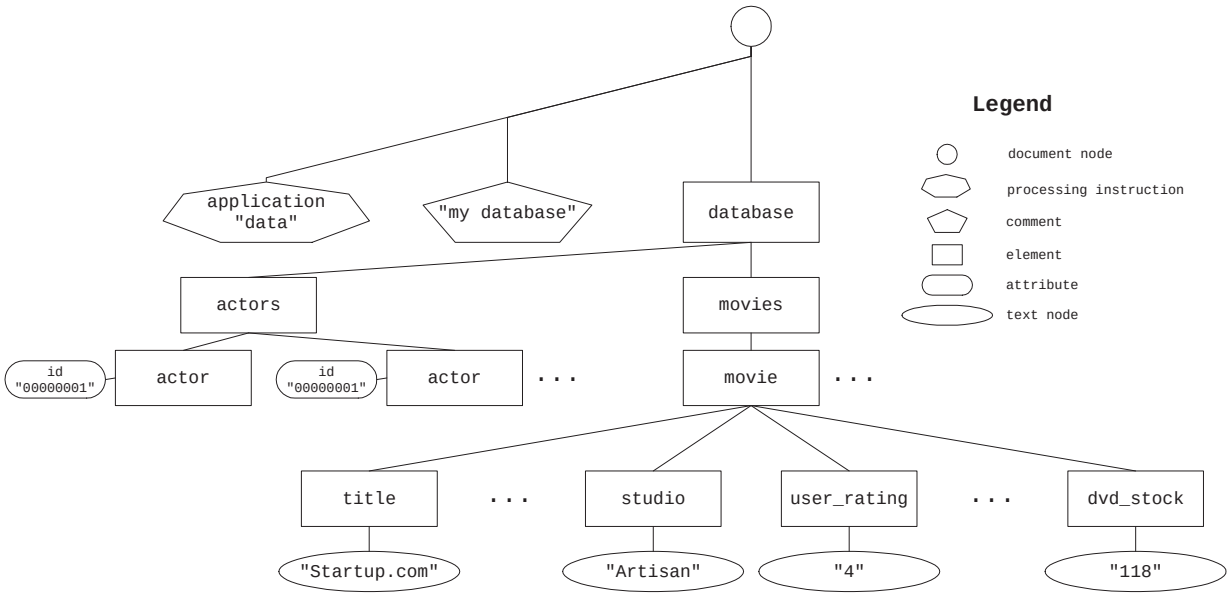


Figure 2: The DOM for Figure 1's `database.xml`. Names appear unquoted; values appear quoted. Ellipses denote omitted elements.

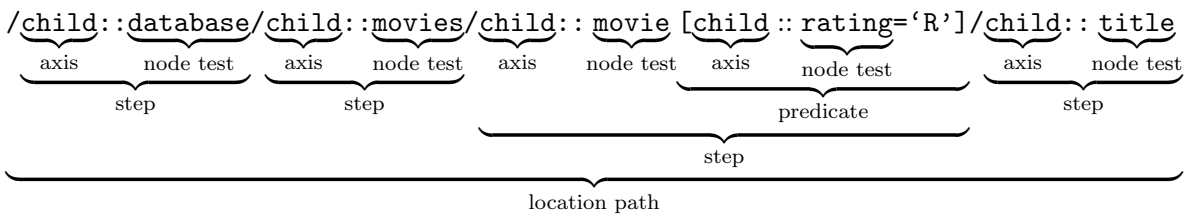
this DOM. In fulfillment of the W3C's first of ten goals ("XML shall be straightforwardly usable over the Internet" [26]), the serialized form has proved a useful format for storage and transmission (over, say, HTTP). But in-memory, pointer- or reference-based DOMs have proved far more useful for applications requiring actual traversals of or non-serialized access to a document's nodes, particularly those which utilize XPath or XQuery for queries.

Unfortunately, the API offers little support for efficient execution of said queries. As DOM's definition of nodes (Figure 3) suggests, traversal of DOMs requires navigation of links between parents and their first (*i.e.*, "left-most") children; siblings are accessible only via linked lists.

2.3 Querying XML

XPath and XQuery facilitate the specification of hierarchical queries on XML documents through *location paths* which are but sequences of *steps*, each of which indicates an *axis* and a *node test*. Each step may be refined with zero or more *predicates*, the effect of which is to filter a query's

results. For instance, the (annotated) XPath query below, applied to `database.xml`, would select all of the titles of movies with a rating of 'R' in our database.



More precisely, the query would select all `title` elements that descend, in turn, from `database`, `movies`, and `movie` elements, the latter of which must have some `rating` child with value `'R'`. It turns out that the `child` axis can be abbreviated with the empty string, so this same *node-set* can be selected more succinctly with the below.

```
/database/movies/movie[rating='R']/title
```

Although XQuery is capable of more complicated queries, the following XQuery query selects the same, additionally wrapping the results in a `titles` element.

```
<titles>
  for $t in document("database.xml")/database/movies/movie/title
  return $t
</titles>
```

The XPath query below, meanwhile, utilizes the `preceding` axis as well some additional syntax to select all of the ratings mentioned in the database, with duplicates removed.

```
/database/movies/movie/rating[not(.=preceding::*)]
```

It suffices to say that XPath and, even more so, XQuery offer powerful querying capabilities. Answering queries efficiently, however, particularly when the document is in serialized form, but even when the document is modelled as a DOM, is non-trivial.

3 The Problems of Serialization and Traversal

Although no information is lost in the serialization of DOMs to XML, retrieval of certain information can be costly. For instance, consider again the retrieval of all titles from `database.xml`: discovery of all `title` elements may require that the entirety of `database.xml` be parsed![2] Despite the need for a minority of `database.xml`'s content, a majority of it must be examined.

[2]The availability of a DTD or schema may allow the parser to terminate before reaching the document's end, but the gains may be negligible. Consider `database.xml`: only once the parser has reached `</movies>` can it be certain that no `movie` (and, in turn, `title`) elements remain.)

```
interface Node {
  // NodeType
  const unsigned short      ELEMENT_NODE       = 1;
  const unsigned short      ATTRIBUTE_NODE     = 2;
  const unsigned short      TEXT_NODE          = 3;
  const unsigned short      CDATA_SECTION_NODE = 4;
  const unsigned short      ENTITY_REFERENCE_NODE = 5;
  const unsigned short      ENTITY_NODE        = 6;
  const unsigned short      PROCESSING_INSTRUCTION_NODE = 7;
  const unsigned short      COMMENT_NODE       = 8;
  const unsigned short      DOCUMENT_NODE      = 9;
  const unsigned short      DOCUMENT_TYPE_NODE = 10;
  const unsigned short      DOCUMENT_FRAGMENT_NODE = 11;
  const unsigned short      NOTATION_NODE      = 12;

  readonly attribute  DOMString            nodeName;
           attribute  DOMString            nodeValue;
                                                 // raises(DOMException) on setting
                                                 // raises(DOMException) on retrieval
  readonly attribute  unsigned short       nodeType;
  readonly attribute  Node                 parentNode;
  readonly attribute  NodeList             childNodes;
  readonly attribute  Node                 firstChild;
  readonly attribute  Node                 lastChild;
  readonly attribute  Node                 previousSibling;
  readonly attribute  Node                 nextSibling;
  readonly attribute  NamedNodeMap         attributes;
  readonly attribute  Document             ownerDocument;
  Node                      insertBefore(in Node newChild,
                                         in Node refChild)
                                         raises(DOMException);
  Node                      replaceChild(in Node newChild,
                                         in Node oldChild)
                                         raises(DOMException);
  Node                      removeChild(in Node oldChild)
                                        raises(DOMException);
  Node                      appendChild(in Node newChild)
                                       raises(DOMException);
  boolean                   hasChildNodes();
  Node                      cloneNode(in boolean deep);
};
```

Figure 3: The DOM (Level 1) API's definition of a `Node`, in Interface Definition Language (IDL) [23]. Documents, elements, attributes, comments, and processing instructions all inherit from this interface.

For large XML documents, the implication may be numerous cache misses or, even worse, disk accesses, even though much of the document's content will prove unwanted. Of course, the document could be retained in main memory as, for instance, a DOM, in order to expedite subsequent queries. But the startup cost of parsing the document just once remains. For queries that require exhaustive traversals of a document's structure (*e.g.*, `/descendant-or-self::*`) or for sequences of queries that require the same, incurrence of said cost may be an unfortunate necessity.

But sparse traversals of XML, particularly in the context of eXtensible Stylesheet Language Transformations (XSLT) [25], are not uncommon. Consider, after all, our simple query for `database.xml`'s `title` elements. Even with a DTD or schema available, a parser may still need to examine most of `database.xml`; it cannot seek from one `title` element to another, since it does not know in advance the exact locations of those elements, particularly when the content of certain elements (*e.g.*, `details`) may be of arbitrary length. Similarly can a parser not seek immediately to the document's first `movie` element; it must first parse the document's `actors` element and all of its descendants.

The serialization of a DOM, then, commits an application to a future reconstruction (or, at least, re-parsing) of that document's structure. In other words, it forces the re-computation of work already performed prior to serialization (*e.g.*, direct association of elements with their children by way of pointers or references).

A solution to this problem is, of course, *lazy parsing*: postponing reads of elements, attributes, processing instructions, or comments until their parsing is absolutely necessary. For Java-based applications, in which memory allocation tends to be expensive [11], avoiding unnecessary instantiation of DOM nodes through lazy parsing is of particular import.

It is toward this end of laziness that Section 4 proposes a summary structure for XML, derived from Munro and Raman's work with balanced parentheses [17]. Unfortunately, this structure fails to improve performance for certain inputs, the result, in part, of DOM's limited navigational capabilities. Section 5, then, proposes an additional summary structure reminiscent of the Lore project's DataGuides [22, 10, 18, 9], along with modifications to DOM itself, to combat those limitations.

4 Summarizing XML with Balanced Parentheses

It is not uncommon for an XML document to be accompanied by a DTD, a grammar for the document's markup. Figure 4, for instance, suggests a DTD for `database.xml`.

XML Schema, a more recent recommendation from the W3C, offers similar capabilities, with additional support for datatypes and more flexible orderings of elements.

Both DTD and XML Schema, then, are designed to specify what the structure of an XML document *may* be. But neither is well-suited for summarizing what the structure of an XML document actually *is*. To be sure, DTD is capable of capturing the exact structure of a document by foregoing use of regular expression operators. For instance, if `database.xml` happened to contain 5 `movie` elements in total, the document's DTD could define the `movies` element as the below.

```
<!ELEMENT movies (movie, movie, movie, movie, movie)>
```

Unfortunately, the DTD now requires that each child of `movies` be identical in structure: every `movie` element must be defined by the same production, such as the below.

```
<!ELEMENT movie (title, genre, rating, summary, details, year, director, studio,
 user_rating, runtime, actor_ref, actor_ref, vhs, vhs_stock, dvd, dvd_stock)>
```

Formerly optional elements are now required (in particular quantities, no less), despite XML's foundation in extensibility. Moreover, explicit (and, really, redundant) enumeration of a document's elements is hardly space-efficient. And the cost of parsing the summary remains.

Fortunately, a summarization mechanism more space- and time-efficient exists.

4.1 Balanced Parentheses

In 1999, Munro and Raman [17] proposed a succinct representation for rooted, orginal trees of n nodes that supports discovery of any node's parent, next sibling, and previous sibling in constant time as well as discovery of any node's ith child in $O(i)$ time, using only $2n + o(n)$ bits, a near information-theoretic optimum. (The representation assumes a RAM model with lg n-bit words.) The representation derives from a depth-first traversal of a tree, such that, for each node, a left parenthesis (*e.g.*, a 0 bit) is produced "on the way down," a right parenthesis (*e.g.*, a 1 bit) "on the way up."

With respect only to elements, Munro and Raman's view of `database.xml` is depicted in Figure 5.

```
<!ELEMENT database (actors,movies)>
<!ELEMENT actors (actor)*>
<!ELEMENT actor (#PCDATA)>
 <!ATTLIST actor id CDATA #REQUIRED>
<!ELEMENT movies (movie)*>
<!ELEMENT movie (title, genre?, rating, summary, details, year, director+, studio?,
 user_rating?, runtime?, actor_ref+, vhs?, vhs_stock?, dvd?, dvd_stock?)>
 <!ATTLIST movie id CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT rating (#PCDATA)>
<!ELEMENT summary (#PCDATA)>
<!ELEMENT details (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT director (#PCDATA)>
<!ELEMENT studio (#PCDATA)>
<!ELEMENT user_rating (#PCDATA)>
<!ELEMENT runtime (#PCDATA)>
<!ELEMENT actorRef (#PCDATA)>
<!ELEMENT vhs (#PCDATA)>
<!ELEMENT vhs_stock (#PCDATA)>
<!ELEMENT dvd (#PCDATA)>
<!ELEMENT dvd_stock (#PCDATA)>
```

Figure 4: A DTD for Figure 1's `database.xml`.

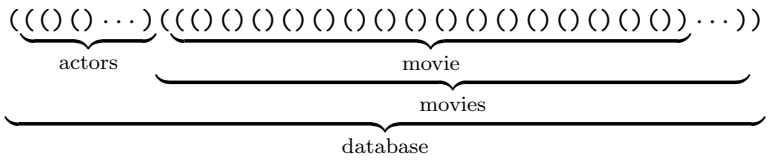


Figure 5: Balanced parentheses for `database.xml`'s elements. Unlabelled pairs represent children of `actors` and `movie` elements. Ellipses denote omitted `actor` and `movie` elements.

Balanced parentheses alone, however, provide insufficient support for lazy parsing, which requires linkage between those parentheses and the sequence of bytes they represent. However, assuming instead a RAM model with $\Omega(\lg N)$-bit words, where N is a document's size in bytes, each single-bit parenthesis can be replaced with a word, whose highest-order bit signifies a start-tag or end-tag and whose lower-ordered bits capture the tag's byte-wise address in the document.

Armed with such a summary, an XML parser can seek in constant time from the start-tag of one element to that of a sibling, without parsing (and possibly instantiating objects for) the former's descendants, as would otherwise be required without any summary. Consider the savings for `database.xml` alone: balanced parentheses with addressing enable a parser to seek from `<actors>` directly to `<movies>`, without even traversing the former's (possibly many) `actor` children.

Inasmuch as XPath and XQuery processors tend to access a node's children in turn, constant-time access to a node's ith child is unnecessary. Therefore, although Benoit, Demaine, Munro, and Raman offer exactly that with their depth-first unary degree sequence representation (DFUDS) for ordinal trees [3], Munro and Raman's balanced parentheses suffice for XML (and, in fact, can be more efficiently outputted by a SAX parser).

Of course, this optimization does come at a price: a document of size N must contain an additional $O(\lg N)$ bits for each of its elements. But the W3C's tenth goal for XML provides some reassurance of the probable negligibility of this overhead: "Terseness in XML markup is of minimal importance" [26]. For small documents that can be parsed quickly, summaries may be of little benefit anyway and can certainly be omitted.

4.2 Generating Balanced Parentheses

To demonstrate the potential value of this paper's proposal of balanced parentheses with addressing, I decided to implement a proof of concept. Lest this work challenge the definition of XML itself, I decided it best to implement the summary structure through processing instructions, which are, by definition, ignored by applications not expecting them.

Although XML does support inclusion of binary data, I opted, in the interests of simplicity and human-readability, to summarize documents using UTF-8 characters in a processing instruction called `summary`. A summarized document's `summary` includes two lines of interest, the first of which contains balanced parentheses for the document's elements and text nodes, the second of which contains the addresses of those parentheses' corresponding elements and text nodes. Albeit small, Figure 6's `foo.xml` is a representative, summarized document.

```
<?summary
 ( ( ) ( (   )  ) (   )  )
 0 5 6 7 22 25 31 32 32 38
?>
<foo>
  <bar baz="qux">quux</bar>
</foo>
```

Figure 6: A representative, summarized document, `foo.xml`. Additional spaces have been added between parentheses for readability. The address at which the the document's first start-tag, `<foo>`, begins, is considered byte 0. The `foo` element's first child, a text node (*i.e.*, "`\n\t`"), begins at byte 5 and ends at byte 6. The `baz` element's start-tag begins at byte 7; the baz element's end-tag ends at byte 31. And so on.

Producing a summary is fairly trivial, inasmuch as it, like XML, represents a pre-order traversal of a DOM's nodes, a traversal which is already necessary for the serialization of a DOM into XML.[3] Of course, prefixing the resulting XML with this processing instruction may require that the former be buffered until the latter is fully computed, especially if the XML is to be streamed to some remote parser. For large documents, such a buffer might be overrun, in which case it would not be unreasonable to append the summary to the XML, provided the recipient needs to write each byte of the document to memory or disk anyway (and can later seek directly to the summary). The purpose of the summary, after all, is to provide a parser with constant-time access to any node's next sibling, once the node's document is already in main memory or on disk. Alternatively, the summary could be stored (or transmitted) as a separate document, as is often done with DTDs and XML schemas.

4.3 Processing Balanced Parentheses

To test the utility of balanced parentheses for XML, I proceeded to implement in Java a *DOM parser* (*i.e.*, a parser which, given an XML document as input, returns the root of a DOM as output). However, whereas a straightforward implementation of the API might process its input serially, constructing the entirety of a DOM before returning its root, my parser's product was a "lazy DOM." Rather than instantiate all of a DOM's nodes, my parser instead parses only the document's `summary`, effectively storing the processing instruction's parentheses and addresses in separate arrays, thereafter returning the root of a DOM not yet constructed. Only when an application requests that document node's first child does my parser's DOM proceed to instantiate a node for its root element; only when that root element's first child is requested does my parser seek directly to and parse that child's start-tag, thereafter instantiating its node; only when that child's next sibling is requested does my parser seek directly to (via a lookup not unlike Munro and Raman's *findclose(i)* operation) and parse its start-tag, thereafter instantiating its node. In short, nodes are instantiated on demand. Boolean flags signify whether nodes have already been instantiated.

Although languages like C and C++ might have offered me greater control over memory man-

[3] In fact, generation of the summary only requires slight modifications to a typical SAX handler (for, *e.g.*, `startElement`, `endElement`, and `characters` events).

agement, Java is, daresay, the world's language of choice for XML-based applications. It seemed prudent, then, for the sake of comparison with other implementations of DOM, to develop my lazy parser in Java.

Rather than implement the W3C's API from scratch, I opted to enhance and instrument Hillion and Kormann's open-source software, DOMJuan 0.1 [12], a partial implementation of DOM (Level-2) which nonetheless provides full support for XPath queries.

4.4 Benchmarks

To evaluate performance, I proceeded to query my parser's DOM with three different XPath processors, all of which accept as input the root of a DOM: Apache's Xalan-J 2.1.0 [6], SourceForge's Jaxen 1.0-FCS [21], and the Jakarta Project's JXPath 1.1 [5]. Although newer versions of Xalan-J exist, all of them convert their input to an alternative, internal representation, a document table model (DTM), which assigns a number to each node and models parent-child and next-sibling relationships entirely with arrays; DOM-like nodes are not actually instantiated.

The first input to my parser was `database.xml`. In its complete, original form, this document contains 390 elements (40 of which are `actor` elements, 18 of which are `movie` elements, 18 of which are `title` elements) and 779 text nodes. In its original form, the file is 19.2 KB; in its summarized form (*i.e.*, with the addition of a `summary` processing instruction), the file is 33 KB. In the interests of inducing a non-exhaustive search of my parser's DOM, my choice of queries for this document was `/database/movies/movie/title`.

For comparison, I performed this query on a DOM constructed by DOMJuan 0.1 in its original form. I also performed the same on two DOMs constructed by Apache's Xerces 1.4.0, one the result of Xerces's "deferred" mode (in which a document is fully parsed, its content stored in arrays, but nodes are only instantiated upon demand), the other the result of "non-deferred" mode (in which all nodes are instantiated prior to queries). Out of curiosity, I also evaluated the performance of Xalan-J 2.5.D1's non-DOM-compliant DTM, using a deferred DOM built by Xerces 2.3.0.

The results appear in Figure 7. The superiority of a `summary`-enabled DOMJuan (at least for queries inducing instantiation of, roughly, 62% of a DOM's nodes) over DOMJuan in its original form seems clear: overall performance (*i.e.*, the sum of build time and query time) improved by, roughly, a factor of 2.

Unfortunately, the results also reveal weaknesses in my parser. Clearly, as Xerces-J 1.4.0's runs suggests, better performance is possible even without lazy parsing. Yet better performance is possible if one foregoes DOM altogether and leverages arrays![4] Although I opted to modify DOMJuan instead of Xerces-J 1.4.0 (which is also open-source) in light of the former's relatively simpler source code, it seems clear that Apache's product would have offered a better baseline.

Figure 9 offers some additional results, for executions of `/DBI/RESULTSET/ROW/TITLE` on `dce_courses.xml`, an SQL dump to XML of the Harvard Extension School's course catalogue, structured as Figure 8 suggests. In its complete, original form, this file contains 8,834 elements (552 of

[4] I suspect DTM's superior performance stems from its avoidance of instantiation of node-like objects altogether as well as from its use of compact, randomly accessible arrays, which prove better fits for cache lines.

DOMJuan	My Lazy Parser	Xerces (non-deferred)	Xerces (deferred)
808	117	255	221

(a)

	DOMJuan	My Lazy Parser	Xerces-J (non-deferred)	Xerces-J (deferred)
Xalan-J	58 (866)	309 (423)	72 (327)	118 (339)
Jaxen	95 (903)	335 (449)	92 (347)	119 (340)
JXPath	171 (979)	411 (525)	173 (428)	201 (422)

(b)

	DOMJuan	My Lazy Parser	Xerces-J (non-deferred)	Xerces-J (deferred)
Xalan-J	1,169	719	1,169	719
Jaxen	1,169	719	1,169	719
JXPath	1,169	737	1,169	737

(c)

	Xerces-J (deferred)
Xalan-J (DTM)	134 (283)

(d)

Figure 7: Results of requesting `/database/movies/movie/title` of `database.xml`, using various XPath engines (Xalan-J 2.1.0, Jaxen 1.0-FCS, and JXPath 1.1) and DOMs (DOMJuan 0.1, my lazy DOM, and Xerces-J 1.4.0 in non-deferred and deferred modes) on a 1.6 GHz Pentium 4 with 768 MB of RAM. (a) Build times, in milliseconds, averaged over 10 runs. (b) Query times, in milliseconds, averaged over 10 runs. Sums of build and query times appear parenthetically. (c) Numbers of element and text nodes instantiated. JXPath appears to instantiate a number of nodes unnecessarily. (d) Query time (and sum of build and query times), in milliseconds, averaged over 10 runs, using Xalan-J 2.5.D1's DOM-incompliant DTM and Xerces-J 2.3.0's deferred mode; build time was 149 ms.

```
<DBI driver="fasdv">
  <RESULTSET statement="select acad_year, term_id, crn, term_name,
  course_group, course_num, title, course_head_name, course_head_title,
  meeting_days, meeting_begin, meeting_end, location, description,
  course_group_name from dcero.course where meeting_days is NOT NULL">
    ...
    <ROW>
      <ACAD_YEAR>2002</ACAD_YEAR>
      <TERM_ID>2</TERM_ID>
      <CRN>21701</CRN>
      <TERM_NAME>Spring Term 2003</TERM_NAME>
      <COURSE_GROUP>CSCI</COURSE_GROUP>
      <COURSE_NUM>E-259</COURSE_NUM>
      <TITLE>Developing e-Business Applications Using XML</TITLE>
      <COURSE_HEAD_NAME>David Malan</COURSE_HEAD_NAME>
      <COURSE_HEAD_TITLE>AB, Harvard University</COURSE_HEAD_TITLE>
      <MEETING_DAYS>M</MEETING_DAYS>
      <MEETING_BEGIN>1935</MEETING_BEGIN>
      <MEETING_END>2135</MEETING_END>
      <LOCATION>Sever Hall 103</LOCATION>
      <DESCRIPTION>Intended for students with previous Java programming
      and web development experience, this course introduces XML as a
      key enabling technology in today's e-business applications. Students
      will learn the fundamentals of XML: schemas, XSL stylesheets, and
      programmatic access using standard APIs. Building on these foundations,
      the course will explore in detail a number of case studies that utilize
      XML in e-business: e-commerce, web personalization, portals, and web
      services. Data modeling techniques in XML will be introduced in the
      context of the case studies. Prerequisites: CSCI E-12, CSCI E-50b,
      and CSCI E-119 or equivalents.</DESCRIPTION>
      <COURSE_GROUP_NAME>Computer Science</COURSE_GROUP_NAME>
    </ROW>
    ...
  </RESULTSET>
</DBI>
```

Figure 8: An excerpt from `dce_courses.xml`, an XML document containing 8,834 elements (552 of which are `ROW` elements, 552 of which are `TITLE` elements) and 17,665 text nodes. Ellipses denote additional `ROW` elements.

which are `ROW` elements, 552 of which are `TITLE` elements) and 17,665 text nodes. In its original form, the file is 669 KB; in its summarized form the file is 1.05 MB.[5]

Although selection of this catalogue's `TITLE` elements does induce instantiation of every element in `dce_courses.xml`, my lazy parser nonetheless avoids parsing and instantiating thousands of text nodes (*i.e.*, the children of every `TITLE` element's siblings). Again, as Figure 9(d) reveals, my lazy parser again offers twice the performance of the original DOMJuan but fails to match Apache's performance.

I make no claim that my queries are representative of XPath and XQuery in general. In fact, truly representative queries may not exist, given the languages' flexibility and XML's extensibility. But these queries are certainly not of uncommon form. And, inasmuch as they both induce only

[5]Clearly, the overhead of storing parentheses and addresses as UTF-8 characters becomes significant for large files. But, again, said representation was employed only for the sake of readability; more efficient, binary encodings are possible.

DOMJuan	My Lazy Parser	Xerces-J (non-deferred)	Xerces-J (deferred)
5887	400	722	633

(a)

	DOMJuan	My Lazy Parser	Xerces-J (non-deferred)	Xerces-J (deferred)
Xalan-J	128 (6,015)	2434 (2,834)	139 (861)	277 (910)
Jaxen	183 (6,070)	2,506 (2,906)	200 (922)	246 (879)
JXPath	816 (6,703)	3,730 (4,130)	855 (1,577)	1,062 (1,695)

(b)

	DOMJuan	My Lazy Parser	Xerces-J (non-deferred)	Xerces-J (deferred)
Xalan-J	26,499	18,221	26,499	18,221
Jaxen	26,499	18,221	26,499	18,221
JXPath	26,499	18,773	26,499	18,773

(c)

	Xerces-J (deferred)
Xalan-J (DTM)	473 (828)

(d)

Figure 9: Results of requesting `/DBI/RESULTSET/ROW/TITLE` of `dce_courses.xml`, using various XPath engines (Xalan-J 2.1.0, Jaxen 1.0-FCS, and JXPath 1.1) and DOMs (DOMJuan 0.1, my lazy DOM, and Xerces-J 1.4.0 in non-deferred and deferred modes) on a 1.6 GHz Pentium 4 with 768 MB of RAM. (a) Build times, in milliseconds, averaged over 10 runs. (b) Query times, in milliseconds, averaged over 10 runs. Sums of build and query times appear parenthetically. JXPath postpones actual selection of nodes until serialization of the query's results, the effect of which is to inflate JXPath's "query times" because of I/O stalls. (c) Numbers of element and text nodes instantiated. Again, JXPath appears to instantiate a number of nodes unnecessarily. (d) Query time (and sum of build and query times), in milliseconds, using Xalan-J 2.5D1's DOM-incompliant DTM and Xerces-J 2.3.0's deferred mode; build time was 355 ms.

partial traversal of a document's structure, they do confirm gains possible from lazy parsing.

5 Summarizing XML with Representative Objects

Unfortunately, lazy parsing, albeit aided by balanced parentheses, is inadequate to combat the costs of sparse queries on DOMs. Consider again the number of nodes visited (not to mention instantiated) in search of `database.xml`'s titles. Not only were the `title` elements' ancestors visited, so were the siblings of every `title` element! Similarly was the node of each element in `dce_courses.xml` visited. Even with lazy parsing, more than 700 nodes were visited for the sake of 18 elements in `database.xml`; for the sake of 552 elements in `dce_courses.xml`, more than 18,000 nodes were visited.

This inefficiency results from DOM's limited navigational support.

5.1 Twins and Cousins

Although DOM captures parent-child and next-sibling relationships, it fails to capture relationships among *twins* and *cousins*, where two nodes shall be said to be twins if they are of siblings of like name and two nodes shall be said to be cousins if they are of like name and are children of two siblings or of two cousins.

Were DOM to offer direct links between twins and cousins, XPath and XQuery expressions of the form `/foo/.../baz` (where the ellipsis denotes one or more steps along the `child` axis) could be evaluated far more efficiently in cases where two or more `baz` elements descend from distinct (like-named) nodes. For instance, selection of `/database/movies/movie/title` would only require discovery of one such `title` element; the rest could be found through a traversal of twin and cousin edges.

Predicates do threaten the efficiency of this approach, insofar as twin- and cousin-based retrieval might select nodes with one or more ancestors that fail to satisfy some predicate. However, this work proposes to eliminate said inefficiency through differentiation of cousins into *first cousins*, *second cousins*, *third cousins*, and so on, where two cousins shall be said to be first cousins if their parents are siblings, second cousins if their parents are first cousins, third cousins if their parents are second cousins, and so on.

5.2 Representative Objects

Unfortunately, edges among twins and cousins are insufficient for efficient discovery of like-named nodes, inasmuch as discovery of the first in a series of twins and cousins may require a nearly exhaustive search of a document's structure. However, efficient discovery is possible with *representative objects*, summary structures that provide concise descriptions of hierarchical data [18].

The implementation of such structures for XML is straightforward.

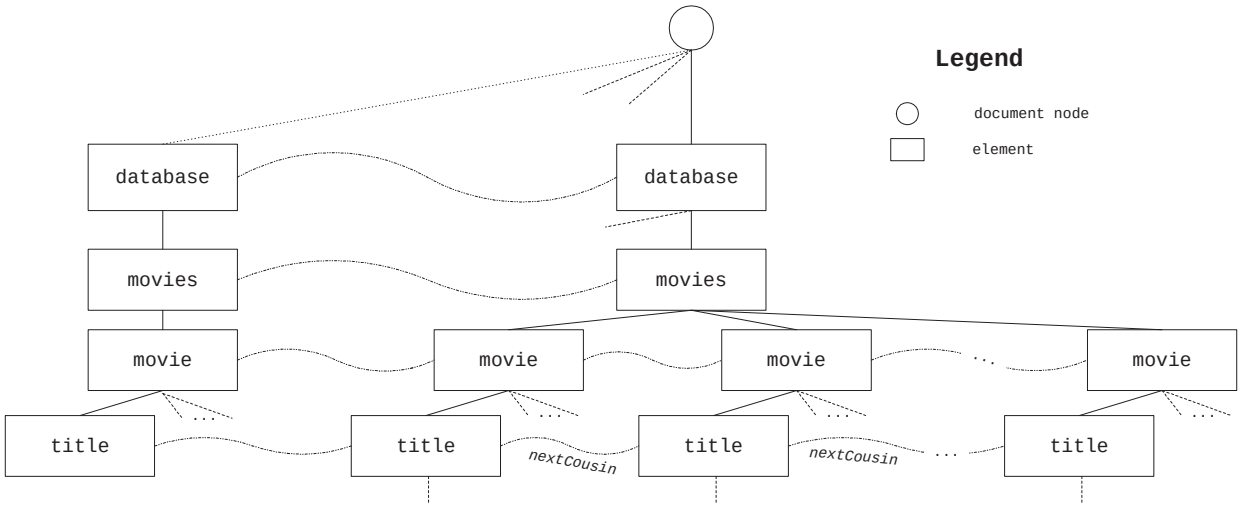


Figure 10: A SUMDOM for `database.xml`; the representative object appears at left. Ellipses and dashed edges imply omitted nodes.

5.3 Implementing Representative Objects

With minimization of node visits (rather than time, per se) now my aim, I decided DOMJuan an acceptable codebase for an alternative to DOM. Putting aside my lazy parser, I proceeded to implement SUMDOM, a superset of DOM with constant-time support for *getPreviousTwin()*, *getNextTwin()*, *getPreviousCousin()*, and *getNextCousin()* operations. SUMDOM additionally maintains a summary structure (not unlike the Lore project's DataGuides [22, 10, 18, 9]) which is essentially a DOM in which twins and cousins are collapsed into a single node. Each node in the summary maintains a link to the first of the nodes it represents, thereby enabling queries for cousins in $O(k + l)$ time, where k is the number of steps in a query and l is the number of cousins to be retrieved. Each node in the summary also maintains a link to the last of the nodes it represents, thereby enabling construction of a SUMDOM in time proportional to that required by a (non-lazy) DOM. Figure 10 depicts a SUMDOM for `database.xml`.

5.4 Benchmarks

In order to evaluate the performance of SUMDOM, I implemented a simple engine for evaluating (predicate-less) XPath queries. Given some query of the form `/foo/.../baz` along with the root of a SUMDOM as input, the engine traverses the SUMDOM's summary structure in accordance with the query's steps, thereafter following any links to twins and cousins. As was expected, SUMDOM expedites navigation significantly (Figure 11). Needless to say, for documents devoid of twins and cousins, the API doubles the number of nodes instantiated and fails to accelerate retrieval. But for applications whose principal concern is minimization of (predicate-less) queries' execution times, SUMDOM reduces edge traversals to a near minimum.

Document	Query	DOM	SUMDOM
`database.xml`	`/database/movies/movie/title`	698	21
`dce_courses.xml`	`/DBI/RESULTSET/ROW/TITLE`	17,666	555

Figure 11: Numbers of edges traversed by DOM and SUMDOM.

5.5 Differentiating Cousins

Section 5.4 took care to qualify SUMDOM as presently optimized only for predicate-less queries. Unfortunately, predicates, particularly ones among a location path's early steps, can degrade SUMDOM's performance significantly. Consider, for instance, the document portrayed in Figure 12, whose elements are numbered (in XML's so-called *document order*) for the sake of discussion.

Selecting Figure 12's `quux` elements via SUMDOM would involve 4 steps along SUMDOM's summary structure and 16 traversals of links between pairs of twins and cousins. Suppose, however, that node 2 failed to satisfy some predicate in a k-step query (*e.g.*, `/foo/bar[@id='2']/baz/qux/quux`, where $k = 5$ and `@` denotes the `attribute` axis), the implication of which is that nodes 5, 6, 8, 9, 12, 13, 15, and 16 should be omitted from the query's results. Ideal would be a SUMDOM that enables a query processor to reach node 20 from node 5 in constant time (or at least $O(k)$ time, where, again, k is the query's length), as it might via a third-cousin edge. Efficient construction of ordinal trees in which each node maintains an arbitrary number of links to cousins remains an area for further study. However, XML's sixth design goal does mitigate this challenge: "XML documents should be human-legible and reasonably clear" [26]. Inasmuch as XML documents, in the interests of readability, are unlikely to boast more than n levels of nesting, where n is small, it may suffice, for real-world applications, to store no more than n links to cousins for each node or, alternatively, to maintain undifferentiated relationships among cousins with skip lists [20].

6 Related Work

Although this work is the first to propose an embedded summary structure for the sake of lazy parsing, it is not the first to propose lazy parsing (for XML) itself. Xerces has long supported a deferred DOM; but, again, even that model requires that its input be parsed fully and stored in various arrays. Noga, Schott, and Lowe, meanwhile, have proposed a parser that scans its input for tag delimiters (*i.e.*, `<` and `>`) in order to construct arrays not unlike Xerces'; however, actual lexing of the input is performed only on demand [19]. In effect, Noga, Schott, and Lowe's parser recomputes the structure of a document, without assistance from any summary.

Just as this work proposes an alternative to DOM, so have others proposed alternatives.

Apache's Xalan-J, recall, maintains a document's structure with DTM [7]. But DTM owes its performance more to issues of memory management than it does to any rethinking of ordinal trees; in fact, the API it exposes to Xalan's XPath processor is essentially the same as that offered by DOM.

JDOM [13] offers an alternative to DOM, but only to the extent that it tailors the API to Java

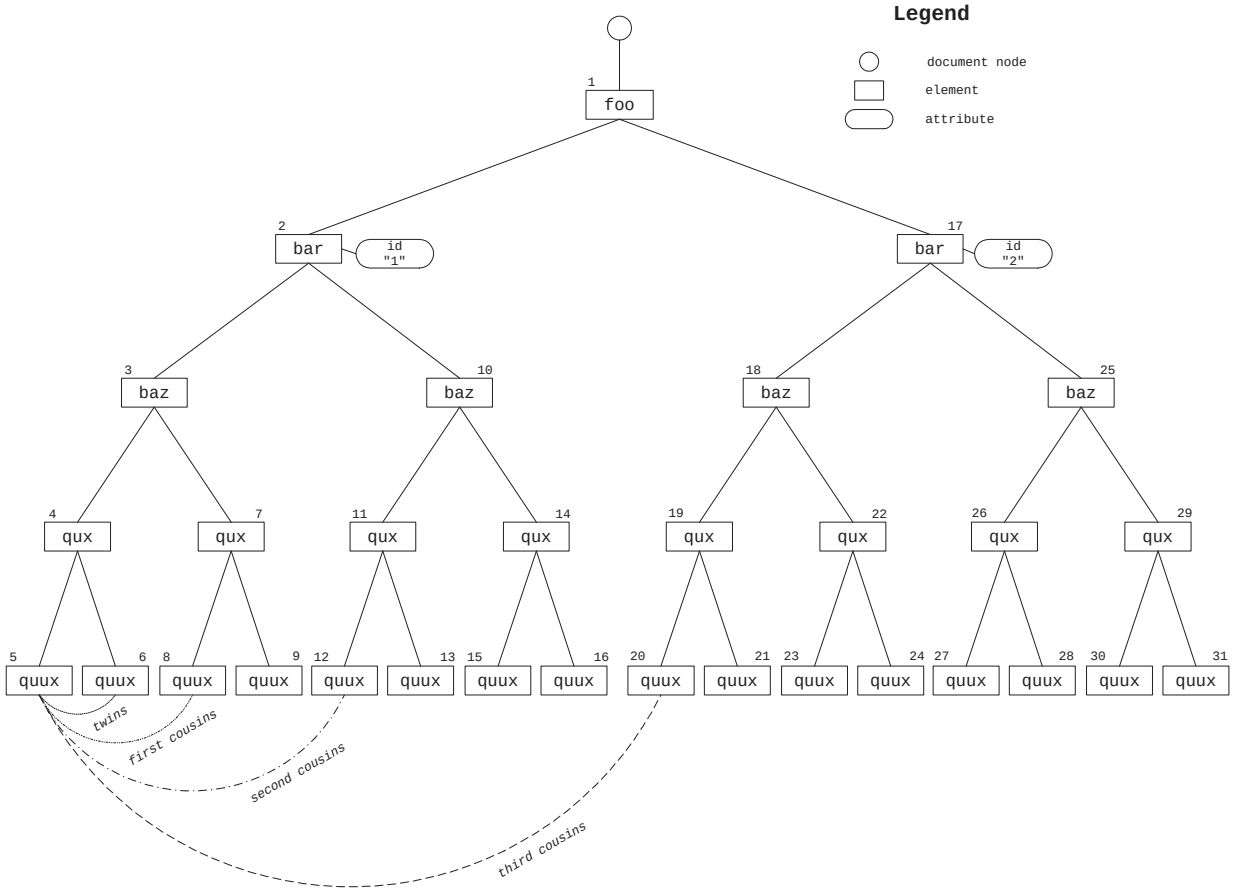


Figure 12: A document for which edges among twins and cousins, such as those pictured, could prove of great benefit, particularly for queries like `/foo/bar[@id='2']/baz/qux/quux`. Names appear unquoted; values appear quoted. Elements are numbered in document order. For readability, only one set of twins and cousins is pictured; a representative object, which would facilitate access to node 5 in four steps, is not shown.

conventions. The tree it maintains is essentially that maintained by DOM.

In contrast, eXist [15], an open-source XML database, maintains indices (reminiscent of this work's representative objects) in which the names of elements and attributes are mapped to arrays of node identifiers, each of which represents a node with that name. However, eXist stops short of evaluating queries "from the bottom-up" (*e.g.*, locating all `quux` elements, thereafter filtering efficiently based on the truth of previous steps' predicates) [16]. Rather, "eXist uses a kind of mixed approach to resolve ancestor-descendant steps in XPath subexpressions, because for two sets of nodes, it conceptually starts at the descendant node and goes up the tree to find the corresponding ancestor."[6]

Although Tamino [1], a commercial product, is rumored to take some sort of bottom-up approach, details are not public.[7]

To my knowledge, efficient evaluation of XPath and XQuery queries through representative objects, supported by twins and cousins, has yet to be implemented fully in any XML processor or database.

7 Further Study

Efficient discovery a node's ith cousin in ordinal trees is an interesting problem, one that would seem to transcend this work's foundation in XML.

Also of interest is the question of efficient layout for DOM, or variants thereof, in multi-level memory hierarchies. Chilimbi, Davidson, and Larus, for instance, have provided evidence of the effect of data's organization on cache performance [4]. Gil and Itai have explored efficient packing of trees in contexts in which block size is known [8]. Building upon that work, Alstrup, Bender, Demaine, Farach-Colton, Munro, Rauhe, and Thorup have extended approximately optimal algorithms to the *cache-oblivious* setting, in which said size is unknown [2]. But the latter both assume a known probability distribution on a tree's nodes. To the extent that queries are not known in advance, DOMs (or variants thereof) which dynamically adjust their nodes' locations in memory might prove an interesting area of study. Subjectivity to insertions and deletions would seem to render the problem all the more interesting.

Incorporation of this work's two proposed summary structures for XML, balanced parentheses and representative objects, into a single implementation might also prove a valuable endeavor, particularly since efficient construction of the latter requires modifications to the former.

8 Conclusion

This work has proposed two summary structures for XML in order to mitigate weaknesses in the W3C's prescription for the serialization and traversal of of the same.

[6] Personal correspondence with Wolfgang Meier, eXist's author.

[7] Personal correspondence with Michael Kay, author of SAXON, an XSLT processor.

The first structure, balanced parentheses, facilitates lazy parsing of an XML document by capturing the document's structure prior to serialization. Support for this structure is already available through processing instructions; incorporation of the structure into documents, then, is straightforward.

The second structure, representative objects, enables efficient execution of location path-based queries through the introduction of twins and cousins to DOM. Yet greater efficiency might be possible through differentiation of cousins.

Of course, a larger question of XML's fundamental appropriateness for hierarchically structured data and queries thereof remains. But this work does provide evidence that more efficient serialization and traversal of XML is possible, despite this metalanguage's disrespect for terseness.

References

[1] Software AG. Tamino XML Server. `http://www.softwareag.com/tamino/`.

[2] Stephen Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, J. Ian Munro, Theis Rauhe, and Mikkel Thorup. Efficient tree layout in a multilevel memory hierarchy. Manuscript, November 11 2002.

[3] David Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing trees of higher degree. In Frank Dehne, Arvind Gupta, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Proceedings of the 6th International Workshop on Algorithms and Data Structures (WADS'99)*, volume 1663 of *Lecture Notes in Computer Science*, pages 169–180, Vancouver, British Columbia, Canada, August 11–14 1999.

[4] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 1999.

[5] The Apache Software Foundation. JXPath. `http://jakarta.apache.org/commons/jxpath/`.

[6] The Apache Software Foundation. Xalan-Java. `http://xml.apache.org/xalan-j/`.

[7] The Apache Software Foundation. Xalan-Java DTM. `http://xml.apache.org/xalan-j/dtm.html`.

[8] Joseph Gil and Alon Itai. How to pack trees. *J. Algorithms*, 32(2):108–132, 1999.

[9] R. Goldman and J. Widom. Summarizing and Searching Sequential Semistructured Sources. Technical report, Stanford University, March 2000.

[10] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445. Morgan Kaufmann, 1997.

[11] Jonathan Hardwick. Java Microbenchmarks. http://www.cs.cmu.edu/~jch/java/benchmarks.html.

[12] Stéphane Hillion and Thierry Kormann. DOMJuan - A DOM Level 2 implementation. http://koala.ilog.fr/domjuan/.

[13] Jason Hunter and Brett McLaughlin. JDOM. http://www.jdom.org/.

[14] David Megginson. http://www.saxproject.org/.

[15] Wolfgang Meier. Open Source XML Database. http://exist-db.org/.

[16] Wolfgang Meier. eXist: An Open Source Native XML Database. In Erhard Rahm B. Chaudri, Mario Jeckle and Rainer Unland, editors, *Web, Web-Services, and Database Systems*, 2593, Erfurt, Germany, 2002. Springer LNCS Series.

[17] J. Ian Munro and Venkatesh Raman. Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[18] Svetlozar Nestorov, Jeffrey D. Ullman, Janet L. Wiener, and Sudarshan S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *ICDE*, pages 79–90, 1997.

[19] Markus L. Noga, Steffen Schott, and Welf Lwe. Lazy xml processing. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 88–94. ACM Press, 2002.

[20] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[21] James Strachan. Project: jaxen. http://sourceforge.net/projects/jaxen/.

[22] Jennifer Widom. Lore. http://www-db.stanford.edu/lore/.

[23] World Wide Web Consortium. Document Object Model (DOM) Technical Reports. http://www.w3.org/DOM/DOMTR.

[24] World Wide Web Consortium. XML Path Language (XPath), Version 1.0, W3C Recommendation. http://www.w3.org/TR/xpath, November 1999.

[25] World Wide Web Consortium. XSL Transformations (XSLT), Version 1.0, W3C Recommendation. http://www.w3.org/TR/xslt, November 1999.

[26] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation. http://www.w3.org/TR/REC-xml, October 2000.

[27] World Wide Web Consortium. XML Schema Part 0: Primer, W3C Recommendation. http://www.w3.org/TR/xmlschema-0/, May 2001.

[28] World Wide Web Consortium. XML Path Language (XPath), Version 2.0, W3C Working Draft. http://www.w3.org/TR/xpath20/, November 2002.

[29] World Wide Web Consortium. XQuery 1.0: An XML Query Language, W3C Working Draft. http://www.w3.org/TR/xquery/, November 2002.