Bulletin of the Technical Committee on

Data Engineering

September 1999 Vol. 22 No. 3

IEEE Computer Society

Letters

Special Issue on XML

Conference and Journal Notices

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering (http: www. is open to all current members of the IEEE Computer Society who are interested in database systems.

The web page for the Data Engineering Bulletin is http://www.research.microsoft.com/research/db/debull. The web page for the TC on Data Engineering is http://www.ccs.neu.edu/groups/IEEE/tcde/index.html.

Letter from the Editor-in-Chief

Repeat of Prior ICDE'2000 Announcement

As many of you perhaps are aware, the deadline for the ICDE'2000 conference, which is the flagship conference of our technical committee, was June 16. I am co-PC chair along with Gerhard Weikum. We had almost 300 submissions, so this promises to be a very selective and high quality conference. Our PC members are now just beginning the reviewing of the submissions, with authors notified of acceptance by October first. ICDE'2000 will be held in San Diego, California from Feb. 28 to March 3. San Diego is a great place (even more so in the winter when the weather is tough in many other places) and the conference will have an excellent program. While we will not know the complete technical program for a while, we do know that there will be two great keynote speakers,

Jim Gray: Senior Researcher in Microsoft Research, and 1999 ACM Turing Award winner;

Dennis Tsichritzis: Chairman of the Executive Board of GMD German National Research Center for Information Technology.

Both are terrific speakers with great insights and much useful information to convey. I would encourage you to plan to attend.

This Issue

SQL has been, in Mike Stonebraker's words, "inter-galactic data-speak" for the relational database world. It looks increasingly like XML will play an analogous role in the world of semi-structured data on the web. Unlike the coming of relational databases, however, and the introduction of SQL, the database research community has not played a large part in this XML revolution. The current issue reflects both the non-research origins of XML and, if researchers act promptly, that it is still early in the revolution and research input can be both useful and effective.

What makes the current situation a close parallel to 25 years ago with the relational model is that the business world is again handing researchers the problem to be solved. Indeed, our industry is racing to provide XML infrastructure for e-commerce, information interchange, effective query of diverse sources, and yet more integration of diverse data. In a sense, the emergence of XML is evidence of a hierarchical data "counter-revolution". But, just as in the political world, things are never quite restored to the way they were before. The expertise that has been painstakingly acquired over years of working with the relational model, together with XML's semi-structured aspects, ensure that history will not repeat. Rather, we are once again setting off into unchartered waters.

The current issue is a very nice blend of tutorial, industry perspective, and research opportunity. Donald Kossmann is the editor for this issue, and has done a very nice job of pulling together papers from both research and industry. The issue clearly captures a technical area in the state of flux. That is, of course, both a problem and an opportunity. Let me thank Donald for his very successful efforts, and invite you all to read an interesting and timely Bulletin issue.

David Lomet
Microsoft Research

Letter from the Special Issue Editor

XML is one of the biggest buzzwords these days. Some researchers and practitioners are even talking about an XML revolution. One expectation is that XML will simplify Web site management and the publishing of documents; for this purpose XML was initially designed. A second vision is that XML will be a lingua franca which makes it possible to exchange data and thus develop truly interoperable applications. Another vision is that XML will turn the Web into a database system, thereby making it possible to pose SQL-like queries and get much better results than from today's Web search engines.

Of course, we are still a far cry from all these visions. The purpose of this issue is to present recent developments and describe what needs to be done. Following the tradition of this Bulletin, the focus is on database and data management issues.

The first paper gives background material. This paper gives an up-to-date overview of XML and XML-related standards and proposals; i.e., RDF, XSL, XPath, and XPointer. The second paper addresses the topic of XML query languages; obviously a good query language is crucial to turn the Web into a database. The paper analyzes the requirements of an XML query language, gives an overview of existing proposals, and describes one proposal, XML-QL, in more detail. The third paper describes a suite of tools developed as part of the Araneus project. The overall goal of this project is to simplify XML Web site management. The paper presents the Araneus data model and discusses some of the challenges of (forward and backward) engineering of Web sites. The fourth paper studies how XML data can be stored in a relational database and contains the results of performance experiments. The overall conclusion is that RDBMSs do extremely well in terms of query performance, but might not be appropriate to store XML for other purposes. The last two papers are visionary papers and give a detailed description of technical issues that need to be addressed before the XML revolution becomes reality. The fifth paper specifically describes the steps that need to be taken before interoperable applications can be built using XML as a data exchange format. The sixth paper describes a research agenda for turning the Web into a database system, the ultimate XML vision.

Donald Kossmann
University of Passau

Describing and Manipulating XML Data

Sudarshan S. Chawathe
Department of Computer Science
University of Maryland
College Park, MD 20904
chaw@cs.umd.edu

Abstract

This paper presents a brief overview of data management using the Extensible Markup Language (XML). It presents the basics of XML and the DTDs used to constrain XML data, and describes metadata management using RDF. It also discusses how XML data is queried, referenced, and transformed using stylesheet language XSLT and referencing mechanisms XPath and XPointer.

1 Describing XML Data

The *Extensible Markup Language (XML)* [BPSM98] models data as a tree of *elements* that contain *character data* and have *attributes* composed of name-value pairs. For example, here is an XML representation of catalog information for a book:

```
<book>
   <title>The spy who came in from the cold</title>
   <author>John <lastname>Le Carre</lastname></author>
   <price currency="USD">5.59</price>
   <review><author>Ben</author>Perhaps one of the finest...</review>
   <review><author>Jerry</author>An intriguing tale of...</review>
   <bestseller authority="NY Times"/>
</book>
```

Text delimited by angle brackets (<...>) is *markup*, while the rest is *character data*. (Here, and in the rest of this paper, we introduce concepts informally as needed for our discussion; for formal specifications, see [W3C99].) Elements may contain a mix of character data and other elements; e.g., the book element contains the text "Here are some..." in addition to elements such as `title` and `price`. The element named `title` contains character data denoting the book title and is contained in the `book` element. Similarly, the element `price` contains character data denoting the book's price. This element also has an attribute named `currency` with value `USD`, represented using the syntax `attribute-name="attribute-value"` within the element's start-tag. In general, element names are not unique; e.g., the book element in our example contains two review elements. However, attribute names are unique within an element; e.g., the price element cannot have another attribute named currency. The syntax permits an empty element `<bestseller></bestseller>` to be represented more concisely as `<bestseller/>`. XML documents are called *well-formed* if they satisfy simple syntactic constraints, such as proper delimiting of element names and attributes and proper nesting of start and end tags.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

1.1 DTD

As described above, XML provides a simple and general markup facility which is useful for data interchange. The simple tag-delimited structure of well-formed XML makes parsing extremely simple. However, applications that operate on XML data often need additional guarantees on the structure and content of such data. For example, a program that calculates the tax on the sale of a book may need to assume that each book element in its XML input includes a price subelement with a currency attribute and a numeric content. Such constraints on document structure can be expressed using a *Document Type Definition (DTD)*. A DTD defines a class of XML documents using a language that is essentially a context-free grammar with several restrictions. For example, one may use the following DTD declaration to constrain XML documents such as those in our `book` example:

```
<!ELEMENT book (title, author+, price, review*, bestseller?)>
<!ELEMENT title  (#PCDATA)>
<!ELEMENT author (#PCDATA|lastname|firstname|fullname)*>
<!ELEMENT price  (#PCDATA)>
<!ATTLIST price currency CDATA "USD"
                source   (list|regular|sale) list
                taxed CDATA #FIXED "yes">
<!ELEMENT bestseller EMPTY>
<!ATTLIST bestseller authority CDATA #REQUIRED>
```

The first line of this declaration is an *element type declaration* that constrains the contents of the `book` element. Following common convention, the declaration syntax uses commas for sequencing, parentheses for grouping, and the operators `?`, `*`, and `+` to denote, respectively, zero or one, zero or more, and one or more occurrences of the preceding construct. Note that the declaration requires every book element to have a price sub-element. The second line declares the type for the `title` element to be *parsed character data* (implying an XML processor will parse the contents looking for markup). Note that the use of some element names (e.g., `review`, `lastname`) without a corresponding declaration is not an error; such elements are simply not constrained by this DTD. The last two lines declare `bestseller` to be an entity that must be empty and that must have an `authority` attribute of type `character data`. The declaration also indicates that the `price` element may have attributes `currency`, of type character data and default value `USD`; `source`, with one of the three values shown (an enumerated type) and default value `list`; and `taxed`, with the fixed value `yes`. The fixed attribute type is a special case of the default attribute type; it mandates that the specified default value not be changed by an XML document conforming to the DTD. Fixed-value attributes are convenient for ensuring that data critical to processing an element type is available with the desired value without requiring it to be explicitly specified for each element of that type. Our example DTD specifies that the book in our XML example must be taxed.

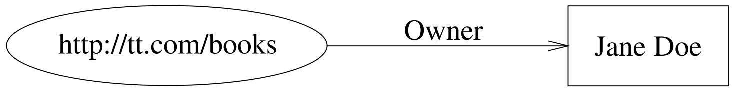
An XML document that satisfies the constraints of a DTD is said to be *valid* with respect to that DTD. The DTD associated with an XML document may be specified using several methods, one of which is the inclusion of a *document type declaration* `<!DOCTYPE BOOKCATALOG SYSTEM "http://tt.com/bookcatalog.dtd">`. in a special section near the beginning of a document, called its *prolog*. This declaration indicates that the XML document claims validity with respect to the `BOOKCATALOG` DTD which may be found at the indicated location.

The data modeling facilities provided by DTDs are insufficient for many applications. For example, we cannot use DTDs to require that the value of the element `price` be a fixed-precision real number in the range zero through 10000 with two digits after the point. Thus our tax-calculation application cannot rely on XML validity with respect to its DTD for such simple error-checking. The XML Schema proposal [BLM$^+$99, BM99] defines facilities that address these needs.

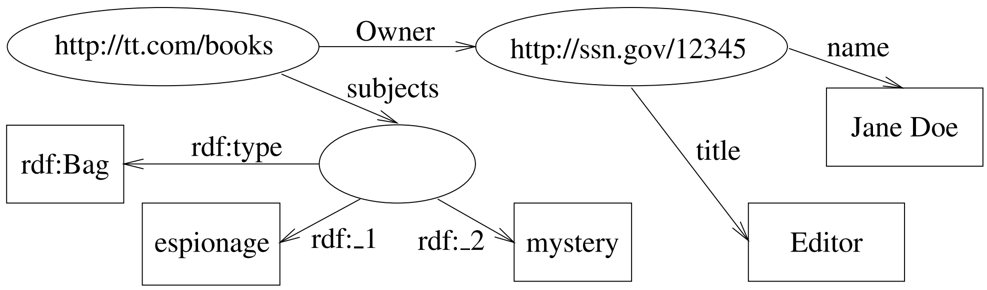
1.2 RDF

The *Resource Description Framework (RDF)* [LS99, BG99] provides a general method to describe metadata for XML documents. More specifically, RDF describes *resources*, which are objects (not necessarily Web-accessible) identified using *Uniform Resource Identifiers (URIs)* [BLFM98]. The attributes that are used to describe resources are called *properties*. RDF *statements* associate a property-value pair with a resource; they are thus triples composed of a *subject* (resource), a *predicate* (property), and an *object* (property value).

For example, suppose we associate the URI `http://tt.com/books` with the XML document in our `book` example above. We may indicate that the XML document is owned by "Jane Doe" using an RDF statement with the following triple: `(Subject: http://tt.com/books; Predicate:Owner; Object: "Jane Doe")` Such RDF statements are graphically represented using ovals for resources, rectangles for literal values, and directed arcs for properties:



The value of a property is not required to be a literal such as the string "Jane Doe" above; it may be another resource. For example, the following RDF graph indicates that the owner of the books data is the resource identified by URI `http://ssn.gov/12345`, which has the name Jane Doe and title Editor.



The above example also illustrates the RDF *container* facility. The subjects property of the books resource has the bag `{espionage, mystery}` as its value. RDF also provides container types sequence and alternative. Note that, like all RDF properties, the subjects property in our example has a single value (the bag). To make a statement about each member of a container, one must use a *distributive referent* attribute that intuitively modifies the meaning of the description element from a single statement to a container of statements (one for each element of the referenced container) [LS99].

RDF also specifies a concrete syntax based on XML for expressing RDF statements. For example, here is a complete XML document representing the above RDF graph:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://w3.org/TR/1999/PR-rdf-syntax#"
         xmlns:bs="http://myschemas.org/books-schema#">
   <rdf:Description about="http://tt.com/books">
      <bs:Owner rdf:resource="http://ssn.gov/12345"/>
   </rdf:Description>
   <rdf:Description about="http://ssn.gov/12345">
      <bs:Name>Jane Doe</bs:Name>
      <bs:Title>Editor</bs:Title>
   </rdf:Description>
   <rdf:Description about="http://tt.com/books">
      <bs:Subjects>
         <rdf:Bag><rdf:li>espionage</rdf:li><rdf:li>mystery</rdf:li></rdf:Bag>
      </bs:Subjects>
   </rdf:Description>
</rdf:RDF>
```

This example also introduces some more XML concepts. Although technically not required, XML documents should begin with an *XML declaration*, similar to the one on the first line of our example, identifying the version of XML used. Element and attribute names appearing in an XML document may be qualified using *XML namespace declarations* [BHL99] such as those on lines 3–4 above. Our example introduces two namespaces. The first is identified using the URI http://w3.org/TR/1999/PR-rdf-syntax# and is assigned a shorthand `rdf`. This namespace contains the elements and attributes defined in [LS99]. The second is an imaginary *books schema* namespace containing properties that describe books (e.g., *Owner*); it is assigned the shorthand `bs`. Namespaces are an important addition to the base XML recommendation because they permit distributed, autonomous development of XML schemas without fear of name clashes. URIs are used in namespaces only for convenience in generating unique names and are not required to identify any Web resource.

RDF permits an intensional definition of bags using URIs. Such a definition is implicit in the use of a *distributive referent* of type `forEachPrefix`. For example, the `forEachPrefix` attribute below intensionally defines a bag containing all resources whose URIs have the specified prefix and establishes the Creator and Publisher of each resource in the bag.

```
<rdf:RDF xmlns:rdf="http://w3.org/TR/1999/PR-rdf-syntax#"
         xmlns:DC="http://purl.org/DC#">
   <rdf:Description aboutEachPrefix="http://cs.umd.edu/~chaw">
      <DC:Creator>Sudarshan S. Chawathe</DC:Creator>
      <DC:Publisher>Dept. Computer Science, Univ. of Maryland</DC:Publisher>
   </rdf:Description>
<rdf:RDF/>
```

The above example uses terms from the *Dublin Core* content description model which is described at the URI shown. This model predates RDF and the RDF Schema recommendation [BG99] includes a schema for it.

One often needs to make statements about other statements (e.g., "Foo believes that the creator of Bar is Baz"). For this purpose, RDF allows a statement to be *reified*: It can be transformed into a resource of type `statement`, with properties `subject`, `predicate`, and `object`, to which additional properties (e.g., `authority` and `PGP-signature` below) may be attached:

```
<rdf:Description>
   <rdf:subject>Bar</rdf:subject>
   <rdf:predicate resource="http://purl.org/DC#Creator"/>
   <rdf:object>Baz</rdf:object>
   <rdf:type resource="http://w3.org/TR/1999/PR-rdf-syntax#Statement"/>
   <authority>Foo</authority>
   <PGP-signature>XmdkA093cDks...</PGP-signature>
</rdf:Description>
```

2 Manipulating XML Data

Given an XML document, one often needs to transform it to better suit the needs of an application. For instance, we may wish to generate a printed catalog containing information about all the books in our running example. In one printed catalog, we may wish to include only the title, authors, and price of each book, skipping other details such as reviews. We may also wish to generate a smaller catalog containing only bestsellers. Further, we may wish to automatically generate a table of contents for these catalogs. Of course, one can implement such applications by writing procedural programs that access the required parts of the source XML document, perhaps using a convenient object interface such as the *Document Object Model (DOM)* [A^+98]. However, XML applications, like database applications, stand to benefit from a declarative languages. Note that the languages described in this section, like RDF in the previous section, operate on the logical tree structure of an XML document (e.g., as supported by DOM), not on its serialization syntax.

2.1 XSL

The *Extensible Stylesheet Language (XSL)* is a language for transforming and formatting XML. Recently, the transformation and formatting parts of XSL were separated. In this paper, we focus on the *XSL transformation language*, called *XSLT* [8], and the related *XPath* [7] and *XPointer* proposals [DJ99].

An XSLT stylesheet is a collection of *transformation rules* that operate (non-destructively) on a source XML document (source tree) to produce a new XML document (result tree). Each rule consists of a *pattern* and a *template*. During rule processing, patterns are matched against the nodes of the source tree, and the template is instantiated (typically using references to the matched nodes) to produce part of the result tree. Templates may contain, in addition to literals and references to matched nodes, explicit instructions for creating result tree fragments. Rule processing starts by instantiating the template of the rule that matches the root element of the source tree. (XSLT uses a conflict resolution mechanism when several rules match a node and default rules when no rules match a node.) Additional elements are processed only when they have been selected for processing by the template of some previously processed element.

Here is an XSL stylesheet for transforming an XML document containing `book` elements (from our running example) to an XHTML [XHT99] document that pretty-prints the title, author, and price of each book, and that includes only the first review for each book. (XHTML is a reformulation of HTML 4.0 in XML.)

```
                    <xsl:stylesheet xmlns:xsl="http://w3.org/XSL/Transform/1.0"
                                    xmlns="http://w3.org/TR/xhtml1"
                                    indent-result="yes">
<!-- Rule 1 --> <xsl:template match="/">
                       <html><head><title>Our New Catalog</title></head>
                          <body>
                             <xsl:apply-templates/>
                          </body>
                       </html>
                    </xsl:template>
<!-- Rule 2 --> <xsl:template match="book/title">
                       <h1><xsl:apply-templates/></h1>
                    </xsl:template>
<!-- Rule 3 --> <xsl:template match="book/author">
                       <b><xsl:apply-templates/></b>
                    </xsl:template>
<!-- Rule 4 --> <xsl:template match="book/price">
                       <xsl:apply-templates/> <xsl:apply-templates select="@*">
                    </xsl:template>
<!-- Rule 5 --> <xsl:template match="book/review[1]" priority="1.0">
                       <xsl:apply-templates/>
                    </xsl:template>
<!-- Rule 6 --> <xsl:template match="book/review" priority="0.5">
                    </xsl:template>
                    </xsl:stylesheet>
```

The first three lines declare the XSL and XHTML namespaces used by the stylesheet. The XHTML namespace is made the default namespace (by skipping the local shorthand in the declaration); thus, unqualified element and attribute names (e.g., `head`) are implicitly in the XHTML namespace. (In XML, text between the *comment delimiters* `<!--` and `-->` is ignored by processors.) Each `template` element describes one transformation rule. The `match` attribute of a template element specifies the rule pattern while its content is the template used to produce the corresponding portion of the result tree. The pattern "`/`" of the first rule denotes the root of the source tree. The template contains some standard XHTML header and trailer constructs. The `apply-templates` element is a rule-processing instruction that denotes recursive processing of the contents of the matched element. (XSLT includes several other instructions which permit templates with constructs such as for-loops, conditional sections, and sorting.) The second rule's pattern, "`book/title`" matches a `title` element if its parent is a `book` element. The template calls for recursive processing of the contents, enclosed in XHTML literals for bold display (`<b>...</b>`). XSL processing includes implicit rules that match elements, attributes, and character data (text) not matched by any explicit rules; these rules simply copy data from source to result tree. In our example, all character data (such as the the text "The spy..." in the title) is copied to the result tree. Rule 4, for processing `price` elements, is similar but includes an additional apply-template instruction to extract the `currency` attribute using the syntax `@*`. Rule 5 matches only the first `review` element in each `book` element due to the "`[1]`" specification. The template simply copies the contents to the result tree (using recursive processing with `apply-templates` combined with the default rules). We ensure that the first review for each book is processed using Rule 5 instead of Rule 6 by assigning Rule 5 a higher `priority`.

2.2 XPath

XSLT rules contain patterns that are matched against nodes (elements, attributes, etc.) in the XML source tree. The language for specifying these patterns is *XML Path Language (XPath)* [7]. Principally, XPath defines the syntax and semantics of *path expressions* such as the following, which matches the last `report` child (in document order) of the `weather` descendants of the node with unique identifier "`favorites`":

```
id("favorites")/descendant::weather/child::report[position()=last()]
```

Path expressions are evaluated in a *context* consisting of a node called the *context node*, a set of nodes called the *context node list*, a set of variable bindings, a function library, and the set of namespaces in scope. Path expressions may be *relative*, selecting nodes by navigating from the current context node, or *absolute*, selecting nodes by navigating from the document root. A path expression consists of a sequence of `/`-separated *steps*, where a step is a *basis* followed by an optional list of *predicates*. Informally, a basis indicates a navigational selection of nodes based on the current context, while the predicate list narrows the list of selected nodes using properties such as position and value. A basis is of the form *AxisName::NodeTest*, where *AxisName* refers to one of several inter-node relationship types and *NodeTest* is a selection condition based on this relationship.

Our path expression example above has three steps: (i)a predefined function that selects the (unique) node that has an `ID` attribute of value "`favorites`"; (ii) a basis `descendant` and node test `weather` that returns a list, in document order, of all `weather` descendants of the context node; and (iii) navigation from these `weather` nodes, giving a list of their `report` children, which are filtered using the predicate in square brackets to yield only the last `report` child for each `weather` node (in document order). The function `position` returns the position of the current context node (at evaluation time) in the context node list, while `last` returns the number of nodes in this list. These functions are from the XPath *core function library*, which includes other functions that return properties of the context node and list as well as common utility functions on numbers, strings, and booleans. Note that the result of performing a basis step is a list of context nodes, not a set. The list order depends on the axis. Intuitively, the basis nodes are in ascending order of distance from the context node.

In addition to `child` and `descendant`, XPath provides the following axes. The `parent` axis contains the parent, if any, of the context node; the parent of an attribute or namespace node is defined to be the element it modifies. The `following-sibling` axis contains siblings of the context node that precede it in the document. The `following` axis contains only element nodes that strictly follow the context node in the document. Descendants of the context node are excluded. The `preceding` axis is analogous; it contains element nodes that strictly precede the context node. The `ancestor` axis contains the proper ancestors of the context node (based on the `parent` axis). The `attribute` (`namespace`) axis contains the attribute (respectively, namespace) nodes attached to the context node if it is an element node, and is empty otherwise. Finally, the `ancestor-or-self` and `descendant-or-self` axes are defined as their names suggest.

Node tests may also use constraints on attribute nodes. For our books example, the following XPath selects book nodes whose price child has attribute currency equal to `USD` and attribute source equal to

```
list: root()/descendant::book/child:price[attribute::currency="USD"
                and attribute::source="list"]/parent::node()
```

This syntax for path expressions is verbose. XPath also defines an *abbreviated syntax* by mapping it to this syntax. For example, `./foo` and `.//foo` select all `foo` children and descendants, respectively, of the context node; `./foo[3]` selects the third `foo` child of the context node. Further, "`.`" and "`..`" are abbreviations for `self::node()` and `parent::node()`, respectively. Thus, we may rewrite our two examples as: `id("favorites")//weather/report[last()]` and `//book/price[@currency="USD" and @source="list"]/..`

2.3 XPointer

Applications may need to address precise portions within XML documents that cannot be modified, e.g., an XML tutorial may wish to annotate specific sections, paragraphs, or sentences of the XML recommendation [BPSM98] without modifying it. (This application is described in [Bra98].) Addressing parts of XML documents is also important when transforming XML using XSLT and XPath as described above. However, the addressing capabilities of XPath are not sufficient. For example, the above application may wish to highlight a region of the XML recommendation that is not a well-formed XML fragment. The *XML Pointer Language (XPointer)* [DJ99] extends XPath to support such applications by adding two new axes to specify basis steps in XPath.

The *range axis* addresses the XML region bounded by the locations addressed by its two arguments. For example, the following XPointer selects the document region between the first and fifth book reviews (inclusive) for our running example:

```
//book/range::review[1],following-sibling::review[4]
```

The range axis is extremely useful for denoting all regions that are marked using a pair of empty elements such as `my-edits-begin/-end` in the following example:

```
XML fragment: ...<Observations>
                    <Temp> 98 99 101 92 <my-edits-begin/> 76 32 99 </Temp>
                    <Pressure> 30 31 33 32 </Pressure>
                 </Observations>
                 <Conclusion>Interestingly, <my-edits-end/>...
XPointer:     //range::descendant::my-edits-begin,following::my-edits-end[1]
```

This XPointer specifies a range beginning at each `my-edits-begin` element and ending at the next `my-edits-begin` element. It illustrates two restrictions on the range axis: (i) although its first argument may reference multiple locations, each such location must be followed by exactly one location referenced by the second argument; (ii) unlike other axes, a range axis may denote regions that cannot be mapped to well-formed context node lists (e.g., the region between the edit markers above). Hence, range axis results cannot be processed further using the XPointer mechanism. (They are intended for use by application programs.)

The *string* axis selects regions using character-based matches (in contrast with the node-based matches used by other axes). The expression "`string::`n,M,p,l," where n, p, and l are integers and M is a string, selects the sequence of l characters starting at the p'th position following the last character of the n'th occurrence of the pattern M. For our running example, the following selects the tenth occurrence of "spy" within a book review element or its descendants, along with 20 characters before and after the word: `/book/review//string::10,"spy",23,43`.

XPointer also adds two functions that specify absolute location paths (similar to the `/` and `id()` expressions used by XPath). The `here()` function locates the element that directly contains (as content or attribute) the XPointer itself (instead of a node in the source tree). The absence of such an element is an error. The `origin()` function is intended for link-traversal and refers to the resource from which the traversal in context began, the absence of such a traversal signaling an error.

References

[A+98] V. Apparao et al. Document Object Model (DOM) level 1 specification version 1.0. W3C Recommendation, October 1998. Available at `http://www.w3.org/TR/REC-DOM-Level-1-19981001`.

[BG99] D. Brickley and R. Guha. Resource Description Framework (RDF) schema specification. W3C Proposed Recommendation, March 1999. Available at `http://www.w3.org/TR/PR-rdf-schema`.

[BHL99] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. World Wide Web Consortium Recommendation. Available at `http://www.w3.org/TR/REC-xml-names`, January 1999.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. IETF (Internet Engineering Task Force) RFC 2396: Uniform Resource Identifiers (URI): Generic syntax, August 1998. Available at `http://www.ietf.org/`.

[BLM+99] D. Beech, S. Lawrence, M. Maloney, N. Mendelsohn, and H. Thompson. XML schema part 1: Structures. W3C Working Draft, May 1999. Available at `http://www.w3.org/TR/1999/xmlschema-1/`.

[BM99] P. Biron and A. Malhotra. XML schema part 2: Datatypes. W3C Working Draft, May 1999. Available at `http://www.w3.org/TR/1999/xmlschema-2/`.

[BPSM98] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0. World Wide Web Consortium Recommendation. Available at `http://www.w3.org/TR/REC-xml`, February 1998.

[Bra98] T. Bray. Using XML to build the annotated XML specification, September 1998. Available at `http://www.xml.com/pub/98/09/exexegesis-0.html`.

[CD99] J. Clark and S. DeRose. XML path language (XPath) version 1.0. W3C Working Draft, July 1999. Available at `http://www.w3.org/TR/WD-xpath-19990709`.

[Cla99] J. Clark. XSL transformations (XSLT) version 1.0. W3C Working Draft, July 1999. Available at `http://www.w3.org/TR/WD-xslt-19990709`.

[DJ99] S. DeRose and R. Janiel Jr. XML pointer language (XPointer). W3C Working Draft, July 1999. Available at `http://www.w3.org/TR/WD-xptr-19990709`.

[LS99] O. Lassila and R. Swick. Resource Description Framework (RDF) model and syntax specification. W3C Proposed Recommendation, January 1999. Available at `http://www.w3.org/TR/PR-rdf-syntax`.

[W3C99] The World-Wide Web Consortium. `http://www.w3.org/`, 1999.

[XHT99] XHTML 1.0: The extensible hypertext markup language. W3C Working Draft, May 1999. Available at `http://www.w3.org/TR/1999/xhtml1-19990505/`.

Querying XML Data

Alin Deutsch
Univ. of Pennsylvania
adeutsch@gradient.cis.upenn.edu

Mary Fernandez
AT&T Labs – Research
mff@research.att.com

Daniela Florescu
INRIA Rocquencourt, France
Daniela.Florescu@inria.fr

Alon Levy
University of Washington, Seattle
alon@cs.washington.edu

David Maier
Oregon Graduate Institute
maier@cse.ogi.edu

Dan Suciu
AT&T Labs – Research
suciu@research.att.com

1 Introduction

XML threatens to expand beyond its document markup origins to become the basis for data interchange on the Internet. One highly anticipated application of XML is the interchange of electronic data (EDI). Unlike existing Web documents, electronic data is primarily intended for computer, not human, consumption. For example, businesses could publish data about their products and services, and potential customers could compare and process this information automatically; business partners could exchange internal operational data between their information systems on secure channels; search robots could integrate automatically information from related sources that publish their data in XML format, like stock quotes from financial sites, sports scores from news sites. New opportunities will arise for third parties to add value by integrating, transforming, cleaning, and aggregating XML data.

Once it becomes pervasive, it's not hard to imagine that many information sources will structure their external view as a repository of XML data, no matter what their internal storage mechanisms. Data exchange between applications will then be in XML format. What is then the role of a query language in this world? One could see it as a local adjunct to a browsing capability, providing a more expressive "find" command over one or more retrieved documents. Or it might serve as a souped-up version of XPointer, allowing richer forms of logical reference to portions of documents. Neither of these modes of use is very "databasey". From the database viewpoint, the enticing role of an XML query language is as a tool for structural and content-based query that allows an application to extract precisely the information it needs from one or several XML data sources.

One salient question is why not adapt SQL or OQL to query XML. The answer is that XML data is fundamentally different from relational and object-oriented data, and therefore, neither SQL nor OQL is appropriate for XML. The key distinction between data in XML and data in traditional models is that XML is not rigidly structured. In the relational and object-oriented models, every data instance has a schema, which is separate from and independent of the data. In XML, the schema exists with the data. Thus, XML data is self-describing and can naturally model irregularities that cannot be modeled by relational or object-oriented data. For example, data items may have missing elements or multiple occurrences of the same element; elements may have atomic values in some data items and structured values in others; and collections of elements can have heterogeneous structure. Even XML data that has an associated DTD is self-describing (the data can still be parsed, even if

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

the DTD is removed) and, except for restrictive forms of DTDs, may have all the irregularities described above. Most importantly, this flexibility is crucial for EDI applications.

Self-describing data has been considered recently in the database research community. Researchers have found this data to be fundamentally different from relational or object-oriented data, and called it semistructured data [1, 3, 18]. Semistructured data is motivated by the problems of integrating heterogeneous data sources and modeling sources such as biological databases, Web data, and structured text documents, such as SGML and XML. Research on semistructured data has addressed data models [16], query-language design [2, 5, 10], query processing and optimization [13], schema languages [15, 4, 11], and schema extraction [14].

In this paper we address the problem of querying XML databases. We start spelling out some requirements for an XML query language in Section 2. Next we describe in some detail XML-QL in Section 3, a query language specially designed for XML, and also illustrate how it satisfies some of the requirements. Section 4 briefly reviews some other languages. We conclude in Section 5

2 Requirements for a query language for XML

In this section we set forth characteristics for an XML query language that derive from its anticipated use as a net query language, along with an explanation of the need for each.

1. **Precise Semantics.** An XML query language should have a formal semantics. The formalization needs to be sufficient to support reasoning about XML queries, such as determining result structure, equivalence and containment. Query equivalence is a prerequisite to query optimization, while query containment is useful for semantic caching, or for determining if a push stream of data can be used to answer a particular query.
2. **Rewritability, Optimizability** XML data will often be generated automatically from other formats: relational, object-oriented, special-purpose formats. Thus, such XML data will be a view over data in these other models. It is desirable that an XML query over that view be translateable into the query language of the native data, rather than having to convert the native data to XML format and then apply a query. Alternatively, when the XML data is native and is processed by a query processor, then XML queries need to be optimizable, like SQL queries over relational data.
3. **Query Operations.** The different operations that have to be supported by an XML query language are: *selection* (choosing a document element based on content, structure or attributes), *extraction* (pulling out particular elements of a document), *reduction* (removing selected sub-elements of an element), *restructuring* (constructing a new set of element instances to hold queried data) and *combination* (merging two or more elements into one). These operations should all be possible in a single XML query. It should not be necessary to resort to another language or multiple XML queries to perform these operations. One reason is that an XML server might not understand the other language, necessitating moving fragments of the desired result back to the sender for final processing. Some of these operations greatly reduce data volumes, so are highly desirable to perform on the server side to reduce network requirements.
4. **XML Output.** An XML query should yield XML output. Such a closure property has many benefits. Derived databases (views) can be defined via a single query. Query composition and decomposition is aided. It is transparent to applications whether they are looking at base data or a query result.
5. **Compositional Semantics.** Expressions in the XML query language should have referential transparency. That is, the meaning of an expression should be the same wherever it appears. Furthermore, expressions with equal result types should be allowed to appear in the same contexts. In particular, wherever an XML term is expected, an expression returning an XML term should be allowed. This latter requirement, interestingly, isn't met by SQL, much to the detriment of those who must implement and use it.
6. **No Schema Required.** An XML query language should be usable on XML data when there is no schema (DTD) known in advance. XML data is structurally self-describing, and it should be possible for an XML

query to rely on such "just-in-time" schema information in its evaluation. This capability means that XML queries can be used against an XML source with limited knowledge of its documents' precise structures.

7. **Exploit Available Schema.** Conversely, when DTDs are available for a data source, it should be possible to judge whether an XML query is correctly formed relative to the DTDs, and to calculate a DTD for the output. This capability can detect errors at compile time rather than run time, and allows a simpler interface for applications to manipulate a query result.
8. **Preserve Order and Association.** XML queries should be able to preserve order and association of elements in XML data, if necessary. The order of elements in an XML document can contain important information — a query shouldn't lose that information. Similarly, the grouping of sub-elements within elements is usually significant. For example, if an XML query extracts `<title>` and `<author>` sub-elements from `<book>` elements in a bibliographic data source, it should preserve the `<title>`-`<author>` associations.
9. **Mutually Embedding with XML.** XML queries should be mutually embedding with XML. That is, an XML query should be able to contain arbitrary XML data, and an XML document should be able to hold arbitrary queries. The latter capability allows XML document to contain both stored and virtual data. The former capability allows an XML query to hold arbitrary constants, and allows for partial evaluation of XML queries. Partial evaluation is useful in a distributed environment where data selected at one source is sent to another source and combined with data there.
10. **Support for New Datatypes.** An XML query language should have an extension mechanism for conditions and operations specific to a particular datatype. The specialized operations for selecting different kinds of multimedia content are an example.
11. **Suitable for Metadata.** XML query language should be useful as a part of metadata descriptions. For example, a metadata interchange format for data warehousing transformations or business rules might have components that are queries. It would good if XML query language could be used in such cases, rather than defining an additional query language. Another possible important metadata use would be in conjunction with XML for expressing data model constraints.
12. **Server-side Processing.** An XML queries should be suitable for server-side processing. Thus, an XML query should be self-contained, and not dependent on resources in its creation context for evaluation. While an XML query might incorporate variables from a local context, there should be a "bound" form of the XML query that can be remotely executed without needing to communicate with its point of origin.
13. **Programmatic Manipulation.** XML queries should be amenable to creation and manipulation by programs. Most queries will not be written directly by users or programmers. Rather, they will be constructed through user-interfaces or tools in application development environments.
14. **XML Representation.** An XML query language should be representable in XML. While there may be more than one syntax for XML query language, one should be as XML data. This property means that there do not need to be special mechanisms to store and transport XML queries, beyond what is required for XML itself.

3 An example: XML-QL

We illustrate here XML-QL, a query language specifically designed for XML [9]. Some other XML query languages are summarized in Section 4. While presenting XML-QL we also illustrate how it satisfies the requirements listed in Section 2.

3.1 Simple XML-QL Query Operations

We start by showing how XML-QL can express the query operations. Throughout this section we use the following running example. The XML input is in the document www.a.b.c/bib.xml, containing bibliography entries decribed by the following DTD:

```
<!ELEMENT bib  ((book|article)*)>
<!ELEMENT book (author+, title, publisher)>
<!ATTLIST book year PCDATA>
<!ELEMENT article (author+, title, year?, (shortversion|longversion))>
<!ATTLIST article type PCDATA>
<!ELEMENT publisher (name, address)>
<!ELEMENT author (firstname?, lastname)>
```

This DTD specifies that a book element contains one or more author elements, one title, and one publisher element and has a year attribute. An article is similar, but its year element is optional, it omits the publisher, and it contains one shortversion or longversion element. An article also contains a type attribute. A publisher contains name and address elements, and an author contains an optional firstname and one required lastname. We assume that the type of all the other entities (e.g., name, address, title) is PCDATA.

Selection and Extraction. *Selection* in XML-QL is done with patterns and conditions. The example below selects all books published by Addison-Wesley after 1991:

```
WHERE <bib> <book year=$y> <publisher><name>Addison-Wesley</name></publisher>
                           <title>  $t </title>
                           <author> $a </author>
      </book> </bib> IN "www.a.b.c/bib.xml", $y > 1991
CONSTRUCT $a
```

XML-QL queries consists of a `WHERE` clause, specifying what to select, and a `CONSTRUCT` clause, specifying what to return. The expression `<bib> ... </bib>` in the `WHERE` clause is called a *pattern*: it is like any XML expression, but can have variables in addition to text data. *Extraction* is done with variables: the query binds the variables `$t`, `$a` and `$y`, and returns only `$a`. The query's answer will have the form:

```
 <firstname> John </firstname>  <lastname> Smith </lastname>
 <firstname> Joe  </firstname>  <lastname> Doe   </lastname>
 <lastname>  Aravind </lastname>
 <firstname> Sue </firstname>  <lastname> Smith </lastname>
 . . .
```

Reduction and Restructuring. The following query retrieves the same data as above but groups the results differently:

```
WHERE <bib> <book year=$y> <publisher> <name>Addison-Wesley </> </>
                           <title>  $t </>
                           <author> $a </>
      </> </> IN "www.a.b.c/bib.xml", $y > 1991
CONSTRUCT <result> <author> $a </>
                   <title>  $t </>
          </>
```

For the simplicity of the syntax we allow `</>` to match any ending tag. *Restructuring* is controlled by the *template* expression in the `CONSTRUCT` clause. Our query's answer will have the following form:

```
<result> <author><firstname> John </firstname> <lastname> Smith </lastname> </author>
       <title/> Tractability </title> </result>
<result> <author><firstname> John </firstname> <lastname> Smith </lastname> </author>
       <title/> Decidability </title> </result>
<result> <author><lastname> Arvind </lastname> </author>
       <title> Efficiency </title> </result>
. . .
```

Reduction is achieved by controlling what elements are returned by the `CONSTRUCT` clause: in this case only `author` and `title`. Usually these elements have to be repeated twice, in the `WHERE` and the `CONSTRUCT` clause. To avoid repetition, XML-QL has a syntactic sugar in which allows us to to give a name to elements in the `WHERE` clause and reuse them in the `CONSTRUCT` clause. Then the query above can be rewritten as:

```
WHERE <bib> <book> <publisher> <name>Addison-Wesley </> </>
                   <title>  $t </> ELEMENT_AS $x
                   <author> $a </> ELEMENT_AS $y
      </> </> IN "www.a.b.c/bib.xml"
CONSTRUCT <result> $x $y </>
```

The query also illustrates preservation of association: authors and titles are grouped as they appear in the input document.

More Complex Restructuring. In the previous answer an author occurs multiple times, once for every title he or she published. The following query groups results by book title. To do so it uses a nested query:

```
WHERE <bib> <book> <title> $t </>
                 <publisher> <name> Addison-Wesley </> </>
    </>  CONTENT_AS $p </>  IN "www.a.b.c/bib.xml"
CONSTRUCT <result> <title> $t </>
                 WHERE  <author> $a </> IN $p
                 CONSTRUCT <author> $a </>
        </>
```

`CONTENT_AS` is like `ELEMENT_AS`, but binds the variable to the element's content. The query's semantics is as follows. The first `WHERE` clause binds the variable `$p` to the content of `<book>...</book>`. For each such binding one `<result>` and one `<title>` element are emitted. Then the inner `WHERE` clause is evaluated, which, in turn, produces one or several authors. Thus the query's answer has the form:

```
<result> <author><firstname> John </> <lastname> Smith </> </> <title/> Tractability </>
                                                              <title/> Decidability </> </>
<result> <author><lastname> Arvind </> </> <title> Efficiency </> </>
 . . .
```

Combination. Recall that *combination* is the operation obtained by merging together two different elements. Assume that we have a second data source at `www.a.b.c/reviews.xml` containing a review, for some of the books, with the following DTD:

```
<!ELEMENT reviews (entry*)>
<!ELEMENT entry   (title, review)>
<!ELEMENT review  (#PCDATA)>
```

The following query combines the the `<book>` elements from the bibliography source with the `<entry>` elements in the reviews:

```
WHERE <bib> <book> <title> $t </> <publisher> $p </> </> </> IN "www.a.b.c/bib.xml"
    <reviews> <entry> <title> $t </> <review> $r </> </> </> IN "www.a.b.c/reviews.xml"
CONSTRUCT <book> <title> $t </> <publisher> $p </> <review> $r </> </>
```

This XML-QL query reads data from two sources, and computes a "join". The join value here is the common title `$t`. The query tries every match in the first data source against every match in the second data source, and checks if they have the same title: if yes, a `book` is output. The query's result will look like:

```
<book> <title> Tractability </title> <publisher>...</publisher> <review>...</review> </book>
<book> <title> Decidability </title> <publisher>...</publisher> <review>...</review> </book>
...
```

Two titles have to be identical to be joined. If we need an approximate match, we have to write some external function and use it as below:

```
WHERE <bib> <book> <title> $t1 </> <publisher> $p </> </> </>  IN "www.a.b.c/bib.xml"
    <reviews> <entry> <title> $t2 </> <review> $r </> </> </> IN "www.a.b.c/reviews.xml"
    similar($t1, $t2)
CONSTRUCT <book> <title> $t1 </> <publisher> $p </> <review> $r </> </>
```

Combination with Skolem Functions. Combination is achieved best in XML-QL with Skolem Functions. The previous query reports only book titles occurring in both sources. The next query inspects the two sources independently, and reports books in either of them. When a book occurs in both, the two elements are combined:

```
{ WHERE <bib> <book> <title> $t </> <publisher> $p </> </> </> IN "www.a.b.c/bib.xml"
  CONSTRUCT <book ID=f($t)>  <title> $t </> <publisher> $p </> </>
}
{ WHERE <reviews> <entry> <title> $t </> <review> $r </> </> </> IN "www.a.b.c/reviews.xml"
  CONSTRUCT <book ID=f($t)> <title> $t </> <review> $r </> </>
}
```

Ignore for the moment the Skolem function part `ID=f($t)`. Without that the query's answer consists of two disjoint sets of books: one with `title` and `publisher`, the other with `title` and `review`: a book occurring in both sources would be reported twice. The Skolem function takes care of the combination. Recall that in XML an attribute of type `ID` assigns a unique key to that element. In XML-QL the attribute name `ID` is predefined to be of type `ID`: hence our query assigns unique id's to the `book` it creates. A Skolem function controls how these id's are created. In our example the function is `f`, and is applied to the argument `$t` meaning that a new id is created for every binding of `$t`. Thus, the first `WHERE-CONSTRUCT` block creates a `book` element for every distinct book with `title` and `publisher` subelements. We assume that distinct books have distinct titles: the Skolem function `f` creates a fresh id for every title. The second block attempts to create new `book` elements, with a `title` and a `review` subelements. However, when a title is found that had been seen before, the Skolem function returns the old id, rather than creating a new one. In that case the `title` and `review` subelements are appended to the existing element. The result is that `book` elements from the first source are combined with `entry` elements from the second source.

No Schema Required. In XML-QL we can query even when the schema is not fully known. This is achieved with tag variables and regular path expressions. The following query finds all publications published in 1995 in which Smith is either an author or an editor:

```
WHERE <bib> <$p> <title> $t </title>
                 <year> 1995 </>
                 <$e> Smith </>   </>
      </> IN "www.a.b.c/bib.xml",   $e IN {author, editor}
CONSTRUCT <$p> <title> $t </title>
               <$e> Smith </>
          </>
```

In this query $\$p$ is a tag variable that can be bound to any tag, e.g., `book`, `article`, etc. Note that the `CONSTRUCT` clause constructs a result with the same tag name. Similarly, `$e` is a tag variable; the query constrains it to be bound to one of `author` or `editor`.

Regular expressions allow us to go one step further. XML data often specifies nested and cyclic structures, such as trees, directed-acyclic graphs, and arbitrary graphs. Querying such structures often requires traversing arbitrary paths through XML elements. For example, consider the following DTD that defines the self-recursive element part:

```
<!ELEMENT part (name, brand, part*)>
<!ELEMENT name  (#PCDATA)>
<!ELEMENT brand (#PCDATA)>
```

Any part element can contain other nested part elements to an arbitrary depth. To query such a structure, XML-QL provides regular-path expressions. The following query retrieves the name of every part element that contains a brand element equal to Ford, regardless of the nesting level at which the part occurs.

```
WHERE <(part)*> <name> $r </> <brand> Ford </> </> IN "www.a.b.c/bib.xml"
CONSTRUCT <result> $r </>
```

Here `(part)*` is a regular-path expression, and matches any sequence of zero or more edges, each of which is labeled part. The wildcard _ matches any tag and can appear wherever a tag is permitted. Other regular-path expressions are alternation (|), concatenation (.), and Kleene-star (*) operators. A regular-path expression is permitted wherever XML permits an element.

Transforming XML data between DTD's. An important use of XML-QL is transforming XML data between DTDs. To illustrate, consider our bibliography data, and assume we want to send it to an application expecting data about people. That is, the application assumes the following DTD:

```
<!ELEMENT person (lastname, firstname, address?, phone?, publicationtitle*)>
```

The following query does the transformation, by mapping `<author>` to `<person>` (of course, the `address` and `phone` elements will be left unfilled):

```
WHERE <bib> <$> <author> <firstname> $fn </>
                         <lastname> $ln </>  </>
                <title> $t </>
      </> </>  IN "www.a.b.c/bib.xml",
CONSTRUCT <person ID=PersonID($fn, $ln)>
              <firstname> $fn </>
              <lastname> $ln </>
              <publicationtitle> $t </>
          </>
```

Note the use of Skolem functions to ensure that a unique `person` element is created for each author given by a firstname and a lastname.

3.2 Data Model and Semantics for XML-QL

In order to define XML-QL's semantics, it is necessary to describe its data models. For XML-QL, there are two data models: an unordered and an ordered model. We describe here only the unordered model, and the associated semantics, and refer the reader to [9] for the ordered model.

The *unordered* data model of an XML document is graph G, in which each node is represented by a unique string called an *object identifier* (OID), with a distinguished node called the *root* and *labeled* as follows: G's edges are labeled with element tags, G's nodes are labeled with sets of attribute-value pairs, and G's leaves are labeled with string values. The graph's edges correspond to XML elements, while the nodes correspond to contents. Attributes are associated to nodes.

The model allows several edges between the same two nodes, but with the following restriction. A node cannot have two outgoing edges with the same labels and the same values. Here "value" means the string value in the case of a leaf node, or the oid in the case of a non-leaf node. Restated, this condition says that (1) between any two nodes there can be at most one edge with a given label, and (2) a node cannot have two leaf children with the same label and the same string value.

Semantics of XML-QL. Consider some XML-QL query `WHERE` P `CONSTRUCT` C. Here P consists of one or several conditions binding the variables $x_1, \ldots, x_k$. Let G be a unordered graph corresponding to an XML document. The semantics consists of two steps. Step 1 deals with the `WHERE` clause. Here we construct a table $R(x_1, \ldots, x_k)$ with one column for each variable. Each row corresponds to a binding of the variables which satisfies all conditions in P. In each row, a variable can be bound either to the oid of an internal node in the graph (i.e. non-leaf), or to a string value (a leaf or an attribute value), or to a tag. Assume R has n rows. Step 2 deals with the `CONSTRUCT` clause, which has a template $C(x_1, \ldots, x_k)$ depending on some of the variables $x_1, \ldots, x_k$. We consider each row i in R, for $i = 1, n$. Let $x_1^i, \ldots, x_k^i$ be the bindings in row i. Then, for that row, we construct the XML fragment $C_i := C(x_1^i, \ldots, x_k^i)$. The query's answer is defined to be $\bigcup_{i=1,n} C_i$.

4 Other XML Query Languages

In this section we will briefly describe other proposals for XML query languages. A more complete list can be found at http://www.w3.org/TandS/QL/QL98/.

Lore (Lightweight Object Repository) is a general-purpose semistructured data management system developed at Stanford [2]. Its query language, Lorel, was obtained by extending OQL to querying semistructured data. Recently [13] Lorel has been adapted to querying XML data. For example, the first XML-QL query is represented in Lorel as:

```
SELECT bib.book.author
WHERE  bib.book.publsher.name="Addison-Wesley" AND
       bib.book.year > 1991
```

	Precise semant.	rewri-tabi-lity	XML output	Ser-ver proc.	All query oprs.	Compq seman-tics	No schema req.	Expl. schema	Pres. order	Prog. manip.	XML repres.	XML embed.	New data types	Meta-data
XML-QL	y	y	y	y	y	y	y	n	y	y	n	y	n	y
LOREL	y	y	n	y	y	n	y	n	y	y	n	y	n	y
XSL	y	n	y	y	n	y	y	n	y	y	y	y	n	y
XQL	y	n	y	y	n	y	y	n	y	y	n	y	n	y
XML-GL	y	n	y	y	y	y	y	y	y	n	n	n	n	y
WEBL	n	n	y	n	y	y	y	n	y	y	n	y	y	y

Table 1: Some XML query languages and how they satisfy requirements 1-13.

XSL (Extensible Stylesheet Language) [8, 7], the stylesheet language proposed by the W3C, can also be viewed as an XML query language. Its expressive power is limited: it doesn't have joins or Skolem Functions. Still, it can serve as a query language of limited scope. Unlike other query languages XSL makes it easy to express recursive processing. As a simple illustration, the following XSL query retrieves all `author` elements, regardless of how deep they occur in the data:

```
<xsl:template>  <xsl:apply-templates/> </xsl:template>
<xsl:template match="author"> <result> <xsl:value-of/> </result> </xsl:template>
```

An XSL program consists of a collection of *template rules*: there are two rules above. Each rule has a match pattern (the value of the `match` attribute), and a template. The match pattern specifies to which element the rule applies: e.g., the second template can only be applied to an `author` element, while the first can be applied to any element (this is the default, when the match pattern is missing). The XSL program proceeds recursively on the XML tree, starting from the root. At each node it tries to apply one of the rules: only the first rule above applies to `bib`, and it's template instructs the processor to apply all rules recursively, on all the children. Eventually, when an `author` element is found, then the second rule applies (it has higher priority than the first rule), which causes the processor to output a `result` element with the `author`'s value.

The XQL [17] language essentially consists of XSL's match patterns extended with some concise syntax for constructing results. Its restructuring expressive power is a strict subset of XSL.

XML-GL [6] is a graphical query language for XML, similar in spirit with the QBE language. Both the `WHERE` and the `CONSTRUCT` clauses are specified with a graphical user interface. Its expressive power is somewhat similar to XML-QL.

Finally, WebL [12] is a scripting language for HTML and XML documents with two important features. First it defines a *markup algebra*, which is similar in spirit to region algebras in text databases. Second, it proposes a *service combinators* for reading a document. Such combinators allow one to express, for instance, that two sources are to be contacted in parallel (e.g., they may have mirror images), but for no more than 20 seconds: if no source has finished downloading, then the data should be fetched from a third source.

Table 1 summarizes how these query languages fulfill the requirements in Section 2.

5 Summary

Given the expectation that XML data will become as prevalent as HTML documents, we anticipate an increased demand for search engines that can both search the content of XML data query its structure. We also expect that XML data will become a primary means for electronic data interchange on the Web, and therefore high-level support for tasks such as integrating data from multiple sources and transforming data between DTDs will be necessary.

In this paper we presented a list of requirements for an XML query language and we presented one possible such proposal: XML-QL. The language supports querying, constructing, transforming, and integrating XML data. XML data is very similar to semistructured data, which has been proposed by the database research community to model irregular data and support integration of multiple data sources. XML-QL is designed based on the previous experience with other query languages for semistructured data. An interpretor for the XML-QL language can be downloaded from http://www.research.att.com/sw/tools/xmlql.

Acknowledgements

The authors thank Serge Abiteboul, Catriel Beeri, Peter Buneman, Stefano Ceri, Ora Lassila, Alberto Mendelzon, Yannis Papakonstantinou.

References

[1] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, Deplhi, Greece, 1997. Springer-Verlag.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[3] P. Buneman. Tutorial: Semistructured data. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 117–121, 1997.

[4] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory*, pages 336–350, Deplhi, Greece, 1997. Springer Verlag.

[5] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[6] S. Ceri, S. Comai, E. Damiani, P. Fraternali, and S. Paraboschi. XML-GL: a graphical language for querying and restructuring xml documents. In *Proceedings of WWW8*, Toronto, Canada, May 1999.

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

[8] J. Clark. XSL transformations (XSLT) specification, 1999. `http://www.w3.org/TR/WD-xslt`.

[9] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *International World Wide Web Conference*, 1999. To appear.

[10] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experience with a web-site management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1998.

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

[12] T. Kistler and H. Marais. Webl — a programming language for the web. In *Seventh WWW Conference*, 1998.

[13] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, Edinburgh, UK, September 1999.

[14] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proceedings of the Workshop on Management of Semi-structured Data*, 1997. Available from `http://www.research.att.com/~suciu/workshop-papers.html`.

[15] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: concise representation of semistructured, hierarchical data. In *International Conference on Data Engineering*, pages 79–90, 1997.

[16] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

[17] J. Robie. The design of XQL, 1999. `http://www.texcel.no/whitepapers/xql-design.html`.

[18] D. Suciu. An overview of semistructured data. *SIGACT News*, 29(4):28–38, December 1998.

Araneus in the Era of XML

Giansalvatore Mecca [1] *Paolo Merialdo* [1,2] *Paolo Atzeni* [2]
mecca@dia.uniroma3.it merialdo@dia.uniroma3.it atzeni@dia.uniroma3.it
[1] D.I.F.A. - Università della Basilicata [2] D.I.A. - Università di Roma Tre

1 Introduction

A large body of research has been recently motivated by the attempt to extend database manipulation techniques to data on the Web (see [13] for a survey). Most of these research efforts – which range from the definition of Web query languages and the related optimizations, to systems for Web site development and management, and to integration techniques – started before XML was introduced, and therefore have strived for a long time to handle the highly heterogeneous nature of HTML pages. In the meanwhile, Web data sources have evolved from small, home-made collections of HTML pages into complex platforms for distributed data access and application development, and XML promises to impose itself as a more appropriate format for this new breed of Web sites. XML brings data on the Web closer to databases, since, differently from HTML, it is based on a clean distinction between the way the data, its logical structure (the DTD), and the chosen presentation (the stylesheet) are specified. By virtue of this, most of the early research proposals for data management on the Web are now being reconsidered in this new perspective (see, for a collection of references, [4]).

In this paper, we discuss the impact of XML on the research work conducted in the last few years by our group in the framework of the ARANEUS project. ARANEUS started as an attempt to investigate the chances of re-applying traditional database concepts and abstractions, such as the ones of data-model and query language, to data on the Web. In this spirit, we have developed several tools and techniques to handle both structured and semistructured data, in the Web style, as follows: (i) a data model called ADM for modeling Web documents and hypertexts [8]; (ii) languages for wrapping [12, 16, 14] and querying [8, 17] Web sites; (iii) tools and techniques for Web site design [9] and implementation [18].

An interesting question is how these tools and applications will work in the era of XML. In the following sections, we will try to answer to this question, by emphasizing how XML fits in this framework, how it is influencing our ideas and our way of thinking about Web data sources, and the design choices needed to adapt our tools – originally conceived for HTML – to the management of XML data. However, a word of caution is needed here: although very popular, XML is still a new proposal, and there are very little (if any) real XML-based applications on the Web. It is therefore quite difficult to reason about the data management problems that will come with XML, since we still haven't experienced them. Because of this, our development will be mainly informal; we will basically try to discuss some of the choices and tradeoffs related to modeling (in Section 2), querying (in Section 3), and developing (in Section 4) XML data sources.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 ADM as a Model for XML

The ARANEUS Data Model (ADM) [8] was introduced as a tool to give an intensional description of Web sites, in the spirit of databases. The model allows to describe the organization of data in Web pages by abstracting the logical features with respect to the physical HTML code. The fundamental modeling primitives of the model have a natural counterpart in the ones that are typically offered by object-database systems, the main differences being the absence of hierarchies and inheritance, and the presence of union types. Thus, ADM modeling primitives might somehow be considered as a subset of ODMG and SQL3 data models, enriched with union types.

In ADM each page is seen as a complex object, with an identifier, the URL, and a set of attributes. Pages sharing the same structure are grouped in *page-schemes* or *page-types*. Attributes in page-schemes have a type, which can be either simple, i.e. mono-valued, or multi-valued. Simple types are `TEXT`, `IMAGE`, and `LINK` (plus other popular MIME types). Links may be typed or untyped; in the latter case, the destination of a link need not to belong to a specified page-type; this is very important when modeling Web sites, since it is often the case that homogeneous links reach pages of different structure and functions. Complex attributes are built using the following type constructors: (i) *tuples*; (ii) *union types*, i.e., disjunctions of attributes; (iii) *lists*, i.e., ordered collections of tuples (possibly nested).[1] A site scheme is a collection of page-types, one for each class of pages in the site, connected by links.

In essence, ADM is neither a conventional, heavy-weight object-model, nor a light-weight semistructured model; it can rather be considered as a *middle-weight* data model, in the sense that it allows to capture structure when this is present, but at the same time offers flexibility in modeling heterogeneities and irregularities using union types and untyped links. In our experiences, the model has proven quite successful in the description of large and fairly well-structured HTML sites, both for site querying and for site development purposes.

When comparing the model to XML, we should keep in mind that XML has its own formalism for describing a document structure, i.e., the DTD. However, XML DTDs do mix together both a description of the logical features of the document, with rather "physical" ones (such as, for example, entities, which can be more appropriately considered as storage units or macros rather than structural components); in fact, XML is rather a format than a data model. If we concentrate on the logical structure, it is possible to see that ADM nicely abstracts the modeling primitives present in XML DTDs, i.e., we can easily associate a database type with a DTD, and a database object with an XML document.

Consider for example a hypothetical XML repository about books, and books reviews, organized according to the DTDs in Figure 1. We have a document containing a list of books (an entry point), pointing to one document for each book; for each book, we have authors, title, price, plus an optional link to a document containing a list of reviews; each review may be either from a customer or from a professional; in the latter case, the reviewer name is provided, and an affiliation (for space reasons, links have been specified simply as a macro `%Xlink`, which we assume corresponds to the full definition of a link in the XLink [5] syntax). These DTDs specify the document structure in terms of a number of elements, i.e., attributes for the associated documents; attributes may be optional, and have a type; this can be either simple or complex; simple type elements are essentially strings (`#PCDATA`) or blobs (`ANY`), or XLinks to other XML documents, like in `<!ATTLIST ToBook %Xlink>`; note that, although there are ways to constrain the destination of an XLink, these are in general unconstrained, i.e., untyped. Complex attributes are based on a limited number of primitives, as follows: (i) *structures*, i.e., typed tuples of elements, like in `<!ELEMENT Book (Author+, Title, Price, ToReviews?)>`; (ii) *union types*, i.e., disjunctions of elements, like in `<!ELEMENT Review (CustReview|ProfReview)>`; (iii) *lists*, i.e., ordered collections of elements (possibly nested) like in `<!ELEMENT BookList (Book+)>`.

It can be seen how these fundamental primitives closely correspond to the ones that are offered by ADM. Figure 1 shows a graphical representation of an ADM scheme corresponding to example above. In the scheme, "stacks" are used to represent classes of documents; edges denote links. Figure 1 also contains an explanation of

[1] The model also offers some rather Web-specific constructs, such as forms and maps, which are outside the scope of this paper.

```
<!DOCTYPE BookList [
  <!ELEMENT BookList  (Book+)>
  <!ELEMENT Book      (Title)>
    <!ATTLIST  ToBook %Xlink>
  <!ELEMENT Title     (#PCDATA)> ]>

<!DOCTYPE Book [
  <!ELEMENT Book (Author+, Title, Price,
                  ToReviews?)>
  <!ELEMENT Author    (#PCDATA)>
  <!ELEMENT Title     (#PCDATA)>
  <!ELEMENT Price     (#PCDATA)>
    <!ATTLIST Price   currency (#PCDATA)>
  <!ELEMENT ToReviews EMPTY>
    <!ATTLIST ToReviews %Xlink>]>

<!DOCTYPE BookReviews [
  <!ELEMENT Title  (#PCDATA)>
  <!ELEMENT BookReviews (Title, Review+)>
  <!ELEMENT Review (CustReview|ProfReview)>
  <!ELEMENT CustReview  (#PCDATA)>
  <!ELEMENT ProfReview  (RevName, Body,
                         Affiliation)
  <!ELEMENT RevName     (#PCDATA)>
  <!ELEMENT Body        (#PCDATA)>
  <!ELEMENT Affiliation (#PCDATA)>]>
```

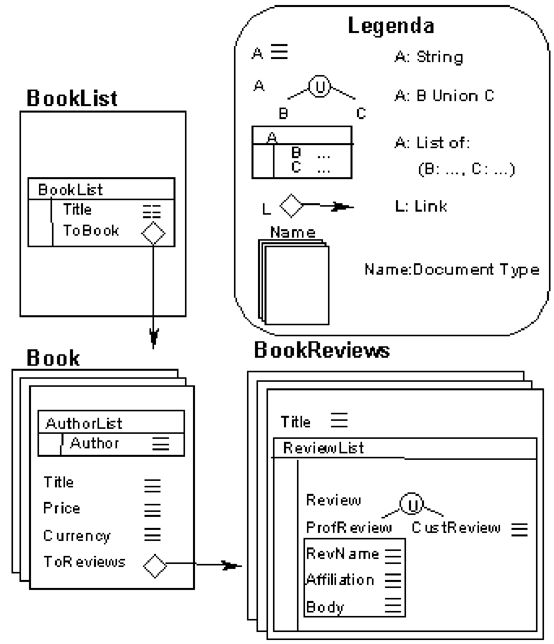


Figure 1: A Sample ADM Scheme

the other graphical primitives.[2]

These ideas show how ADM – and more generally complex-object models with union types and untyped links – provides a natural framework for describing XML repositories and their DTDs, and that we can use the ADM abstraction of DTDs as a description of the repository structure. This holds regardless of the specific kind of application we need to develop on the data, i.e., both in the case in which we need to query an existing data source and want to derive an *a posteriori* high-level description to be used as a basis for the query process, and in the case in which we are designing and developing our own XML data source and prefer to reason at a higher, abstract level, instead of directly working with the XML code. These two classes of applications are discussed in the following Sections.

3 Querying XML Data Sources

In our approach, once a target site has been identified for querying purposes, the ADM description of the site is first derived on the basis of a "reverse engineering" process, in which sets of homogeneous pages in the site are classified into page-schemes. Then, for typical HTML data sources, appropriate wrappers need to be written in order to build, from HTML pages in the site, an internal representation of data as instances of the data model. This process, which in general is rather delicate and demanding, can either be conducted by manually coding the wrappers, or with the help of algorithms and tools for automatic wrapper generation. The ADM description of the site and the corresponding wrappers are then used as a basis for the site querying process.

Consider now the case in which we need to query an XML data source instead of an HTML one. As it is reasonable, we assume that the data source is in general composed of several XML documents, linked using

[2] To see how the model abstracts with respect to the physical features in the DTD, consider for example attribute currency of element price in DTD Book; although in the DTD it was considered rather as a metadata, and therefore stored in a slightly different way with respect to elements (such as the title, for example), in the database abstraction of the DTD we may decide to ignore this difference, and model it as a conventional attribute; the same happens in our example for links; in this way, the model provides a uniform view and query mechanism for data and metadata.

XLinks. As discussed above, ADM may, in principle, model the structure of these sources based on their DTDs. There are however a number of subtleties in this modeling, and its effectiveness is strongly related to the nature of the data, as we discuss in the following paragraphs.

DTDs are Optional Our modeling approach is heavily based on the mapping of DTDs to ADM types. It is well known that DTDs are not mandatory. They are explicitly referenced in *valid* XML documents, and are used by an XML parser to validate the document itself. Other documents, called *well-formed*, do not have an explicitly associated DTD. Due to this freedom, it is still unclear to what extent DTDs will be adopted in XML applications; some consider DTDs useless; on the other side, there is a growing interest by major software vendors in defining standard classes of DTDs to be used, for example, in e-commerce applications to ease data-exchange and integration [2].

However, whatever the usage of DTDs will be, we would like to note that, given the special tagging format of XML documents, a (minimal) DTD for which the document is valid can be derived by a simple syntactic check of the document.[3] We can thus say that any XML document has an associated DTD (either explicit or implicit) describing its content. In the following, we will assume that DTDs are available and use ADM to abstract their logical structure.[4]

DTDs are not Database Types It is true that DTDs are not exactly what we usually consider a type. They have been introduced by the document community as a means for specifying the document structure from the parser viewpoint rather than from the DBMS one. As a consequence, there are several shortcomings associated with their use. For example, elements are ordered in DTDs, and, although this order may in some cases be relevant, it becomes immaterial once the corresponding database type is built (as noted in [19]). Also, elements may in some cases mix free text with subelements, thus complicating the modeling. There are in fact more elegant and elaborated proposals for superimposing schemes on XML data that are now coming from the database community (see, for example, [5]). However, these proposals are still at a preliminary stage, and their success in the XML community is still unclear.

In the meanwhile, DTDs represent a reasonable compromise. DTDs have been used for a long time in text databases as a schema definition formalism for documents (with reference to SGML, of which XML is a derivative; see, for example, [15, 7, 10, 11]). These works mainly concentrate on querying a single SGML document at a time, or several documents from the same DTD, by loading them in a database, and do not consider collections of linked documents coming from different DTDs that are to be queried remotely through the network. Still, such a large body of work shows that it is possible to build quite effective mappings from the DTD structure to a database structure, and adopt the latter as a schema for the data. This approach is also advocated by some of the major database vendors (e.g., Oracle [6]) in their support for XML data.

How (Semi)Structured is XML ? A nice feature of XML is that DTDs also act as natural wrappers for the documents; in fact, they specify a grammar that can be used to parse the documents looking for attribute values. Hence, we do not need to generate wrappers as it was needed for HTML sites, and, once the ADM scheme has been constructed based on the DTDs, it is in principle possible to run queries on the source.

However, there are peculiarities of XML data that depart from the traditional database framework, and impose to re-consider some aspects of the querying process. It can be seen that the approach we have discussed so far is more effective on fairly well-structured sources, and degrades as soon as the data source becomes less structured. It is very likely that, in the same way as it happens for current HTML Web sites, XML repositories available on the Web will range from very structured and regular to highly unstructured (with a whole degree of variants in

[3]This is, in essence, what a browser does when it parses a well-formed document.

[4]Based on these ideas, we have built a tool that automatically associates an ADM scheme with an XML repository, using the existing DTDs and inferring the missing ones.

between). The latter case will most probably happen in all applications in which the XML data source to model and query is remote and autonomous, i.e., a collection of linked documents available on the Web, completely beyond the control of the query system.

In the case or fairly structured repositories, it is possible to use query languages and optimization techniques in the spirit of databases. It is, for example, possible to load the whole repository in an object database according to the ADM types associated with DTDs[5] and query it using SQL. Or – opposite to this "warehousing" approach – in all cases in which freshness of query answers is a requirement, it is possible to extract data on-line from the repository by navigating it through the network. To this end, we have developed a query algebra, called ULIXES [9], for navigating and querying data sources on the Web; queries are written using path-expressions on the ADM scheme of the source: based on the specified path, ULIXES navigates the source starting from some entry-point (whose URL is known to the system) and materializes the query result in a local database. To give one example, suppose that we are interested in finding all books in the repository above having at least one review written by Codd. Based on the scheme in Figure 1, the query can be written in ULIXES as follows[6]:

```
CREATE VIEW CoddBooks (Title, Price)
OVER        http://books.bla.com
AS SELECT Book.Title, Book.Price
   FROM   BookList.ToBook -> Book.ToReviews -> BookReviews
   WHERE  BookReviews.ReviewList.ProfReview.Name LIKE '%Codd%'
```

There are a couple of important points that we want to emphasize. When applicable, this approach has several advantages that are due to the availability of a concise description of the repository structure (in the spirit of a database scheme) and of a query algebra based on that. In fact, this allows to specify more accurate queries, and to apply optimization techniques to make query evaluation more efficient. However, we do not require that the data source is loaded in the database before actually querying it. ULIXES is capable of navigating it remotely and materializing in the database only the portions of interest. Therefore, the XML repository is not a conventional database, and traditional query evaluation techniques need to be reconsidered in this respect [17].

If, on the contrary, the degree of structure in the repository is relatively low, or the scheme tends to evolve quickly, the techniques above will be quite ineffective. In these cases query languages developed for semistructured data, which completely ignore the schema, are to be considered more appropriate (we don't have enough space here to elaborate on the various query languages inspired on semistructured that have been proposed for XML, and refer the reader to [4]).

We therefore envision a scenario in which different XML applications will require the adoption of different data models and query languages, going from very structured to semistructured. A key problem, in this respect, is that of choosing the right approach based on the level of "structuredness" of the target data source. However, a nice feature of XML – which represents a fundamental difference with respect, for example, to HTML – is that it allows to precisely *measure* the level of structuredness of a data source, as follows.

Measuring Regularity and Stability in XML Data There may be two different sources of semistructuredness in XML repositories. The first one is due to low regularity in the structure, the second one to high chances that the structure evolves from time to time. In both cases it is still possible to build an ADM scheme of the repository (possibly inferring the missing DTDs). However, in the case of quite irregular repositories, the number of different DTDs will be large, and therefore the ADM scheme will be large. To see this, consider the book example above: suppose that in a less regular version of the data two (or more) different DTDs are used for books (for instance, `Book` for ordinary books, `OldBook` for old and collectable books, and `NewBook` for books to be published, all with slightly different sets of elements); then, when reconstructing the structure, these would be modeled as

[5] In this respect, some mechanism for modeling union types in conventional databases is needed.

[6] With respect to the version of the language presented in [8], in the last version of the prototype we have changed the syntax to make it more similar to SQL3.

separate page-types, and union types would be introduced for each link towards books. As an extreme, in some cases the number of different types may be in the order of the number of documents. We can in this case measure the *level of regularity* in the repository by the ratio between the size of the scheme (i.e., the number of types and the number of attributes inside types) and the size of the data (i.e., the number of documents).

Evolution of the repository structure (i.e., changes in the existing DTDs or use of new DTDs) of the repository are equally important, since they may require to change the database schema. If the DTDs change, the ADM schema also changes from one repository analysis to the next one. Also in this case we can measure the level of stability in the repository in a given period of time, for example by using the ratio between the size of the schema, and the number of types that have changed, or the speed of change (number of changes over time).

To summarize, we foresee that XML will be used in data-intensive applications of very different nature; in some of these, the data will be very structured and stable, in others quite semistructured or unstable; these different classes will require the adoption of different query languages, which can be based on the schema or not. Once a target XML repository has been selected, the measures of regularity and stability discussed above can be used as a metric and provide a criterion to choose the right approach to query the repository. However, more work and experiments on actual XML data are needed to establish the right tresholds for this choice.

4 Developing XML Data Sources

Beside accessing existing Web data sources, another important class of applications is devoted to developing new ones (see [13] for a survey). This is even more important in the case of XML, since we are witnessing a growing interest in its adoption, but there are very little available data yet. In this section, we discuss how techniques we have developed for HTML sites have been extended to generate XML repositories as well.

Model-Based Development in ARANEUS Differently from market tools – which are mostly rather low-level tools, either oriented to pure-HTML development or to procedural and SQL programming – we have adopted a *model-based approach* to Web site development, in the sense that our approach heavily relies on the adoption of high-level models, both at the conceptual and logical level, for designing and developing sites at all levels of the site design process, namely designing data, hypertext and presentation.

To leverage robust and wide-spread technology, we adopt a relational database as a back-end for the site; as a consequence, data to be published in the site is described at the conceptual level using the well-established Entity-Relationship Model, and at the logical level using relational tables.

The hypertext structure is described using ADM; consider again Figure 1; suppose now we are designing a book site from scratch and want to develop it based on an underlying database – that is, we are now forward-engineering the site, instead of reverse-engineering it as discussed in the previous section; a crucial step would be to specify how the target hypertext is to be organized; it can be seen how ADM is particularly effective in giving a concise description of the site hypertext.

Finally, based on ADM, we also have a precise notion of *style* for describing the graphical layout of data items in a page, and an associated tool, TELEMACO [18] for designing styles. Assuming that we have chosen the scheme in Figure 1 as a scheme for the target site, styles are used to associate a graphical layout with data items in pages. An attribute style is made of two arbitrary pieces of HTML code, between which the attribute values will be enclosed when generating pages. Consider the book title in the example above; a simple style for it might be: `NAME: [<FONT COLOR="red"><B>] [</B></FONT>]`. In this way, when the actual HTML page is generated, the book title – say "*The HTML Sourcebook*" – a piece of HTML code of the form: `<FONT COLOR="red"><B>The HTML Sourcebook</B></FONT>` will be produced.[7]

[7]The styling mechanism provided by TELEMACO is in fact more sophisticated. For example, it offers rapid prototyping facilities, and allows to work with sample HTML pages, from which it generates styles, instead of directly writing style code. We do not elaborate on this here for space reasons.

These models fit into a larger methodological framework [9], in which designers start from a conceptual description of the site domain (an Entity-Relationship scheme) and through a set of precise steps progressively moves to database logical design (this produces the database relational scheme), then hypertext design (this produces the site ADM scheme), and finally presentation design (producing page-styles). This process is assisted by HOMER [18], a CASE tool conceived to simplify this design process, and to automatically produce the code for page generation based on the site design artifacts. One of the main advantages of this model-based approach is that the overall process is in essence independent from the actual tool chosen for page generation; in fact, in its current version, HOMER may either generate code for our own tool for Web site generation, called PENELOPE [8], or Java Server Pages Templates [3]; however, it could easily be extended to generate code for any of the major Web database gateways. Another important advantage of working with high-level models such as ADM and TELEMACO styles is that the site generation phase is independent from the chosen format, HTML or XML. We have in fact re-generated most of the sites originally developed in HTML using XML, as discussed in the following paragraphs.

From HTML Sites to XML Repositories In our approach, each page is seen as an instance of an ADM type, i.e., as a URL-identified nested object. When the page is generated, as a first step the relevant data is extracted from the database and it is nested into the corresponding ADM object; then, as a second step, the actual HTML code is produced, as specified by the corresponding style. Based on the close correspondence between ADM and DTDs, we have easily adapted our tools in order to produce an XML document (possibly with XLinks) from the ADM object — by enclosing each attribute value between the corresponding element tags — and the associated DTD from the ADM page-type. As a result of this step, the system produces a collection of linked XML documents and their DTDs.[8]

XSL in Action As such, the repository could be used for exchange purposes, but it is not browsable. To make it browsable we need to encode the chosen TELEMACO style using one of the stylesheet formalisms that have been proposed for XML. After some experiments, we decided to adopt XSL [5], the "Extensible Stylesheet Language", and most of our experiences were done using Internet Explorer 5, one of the first browsers equipped with XML and XSL processors.[9]

XSL is a full-fledged functional programming language for tree-like structures like XML documents that can be used to restructure them at will. In fact, the straightforward way of associating a browsable layout to an XML document using XSL is to "restructure" it into HTML code. In essence, a stylesheet associated with an XML document is a collection of definitions (called "templates"), that specify how to recurse down the XML tree and what output to produce when a certain element is encountered. This output will go to the browser, which will display it on the screen. To give an example, suppose in documents about books we have an element `<Title>`, like in `<Title>The HTML Sourcebook</Title>`; then we may have a template in the stylesheet that specifies that, whenever element `<Title>` is encountered, a piece of HTML code must be produced by enclosing the corresponding value between certain HTML tags, like, for example, in `<FONT COLOR="red"><B>The HTML Sourcebook</B></FONT>`.

[8] There are several extended features of XML that currently the system does not handle; for example, we do not use XML element attributes, as the distinction between XML elements and their attributes is still unclear to us; we generate IDs and IDREFs, but only to a limited extent; we do not generate extended XLinks, neither XPointers.

[9] There are two main contenders for associating a presentation with XML data and make the documents browsable. One is XSL, currently still at the stage of a proposed W3C recommendation. The other one, already a W3C standard, is CSS [5], for "Cascading Style Sheets"; conceived originally for HTML, it allows to associate with each tag in a document (and therefore with each XML element) a set of layout properties, like, for example, the font size and face or the paragraph alignment. While we write, there is still some discussion about what should be the preferred format for styling XML documents. The two major browsers have chosen different routes: Internet Explorer version 5 has adopted the more flexible XSL, whereas Gecko (code name for Netscape 5) only supports the more stable CSS. Note that CSS, in our experience, may be profitably used in conjunction with XSL, by embedding it into the HTML code generated by XSL to refine the layout of the HTML tags. We have done this in some of our experiments.

It can be seen that, like ADM objects can be straightforwardly mapped to XML documents, similarly TELEMACO styles can be easily implemented as XSL stylesheets. As an outcome, our system can easily generate both plain HTML sites or XML sites with XSL stylesheets. Although in principle completely transparent, this operation has several subtleties. This is due to the fact that the XSL processor is quite strict in enforcing correctness of the HTML code it produces: it is in fact designed to produce XHTML code, the reformulation of HTML 4.0 in XML. As a consequence, some typical errors HTML developers do – like opening and not closing tags like `<P>` or `<TD>` – are not allowed, and produce run-time errors when the page is displayed.[10] As a consequence, a certain effort was needed in order to polish the HTML code in styles, before being able to use XSL.

Some of the XML sites we have generated, including some portions of the SIGMOD On Line Web site (`http://www.acm.org/sigmod`), are available on the ARANEUS Project Web site [1].

References

[1] The Araneus Project Web site. `http://www.dia.uniroma3.it/Araneus`.

[2] The Biztalk Web site. `http://www.biztalk.org`.

[3] Java Server Pages (JSP) home page. `http://www.java.sun.com/products/jsp/`.

[4] The W3C Query Languages Workshop, 1998 `http://www.w3.org/tands/ql/ql98/`.

[5] The W3C Technical Reports W3C Working Draft, March 1998. `http://www.w3.org/TR/`.

[6] Xml support in Oracle8i and beyond, 1999. `http://www.oracle.com/xml/documents/xml_twp`.

[7] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Siméon. Querying documents in object databases. *Journal of Digital Libraries*, 1(1):5–19, April 1997.

[8] P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *VLDB'97*.

[9] P. Atzeni, G. Mecca, and P. Merialdo. Design and maintenance of data-intensive Web sites. In *EDBT'98*.

[10] G. E. Blake, M. P. Consens, P. Kilpeläinen, P. Larson, T. Snider, and F. W. Tompa. Text/relational database management systems: Harmonizing SQL and SGML. In *ADB'94, LNCS 819*, Springer-Verlag, June 1994.

[11] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *SIGMOD'94*.

[12] V. Crescenzi and G. Mecca. Grammars have exceptions. *Information Systems*, 23(8):539–565, 1998.

[13] D. Florescu, A. Levy, and A. O. Mendelzon. Database techniques for the world wide web: A survey. *Sigmod Record*, 27(3):59–74, 1998.

[14] S. Grumbach and G. Mecca. In search of the lost schema. In *ICDT'99*.

[15] P. Kilpeläinen, G. Lindén, H. Mannila, and E. Nikunen. A structured document database system. In *Intern. Conf. on Electronic Publishing, Document Manipulation and Typography (EP'90)*, pages 139–151, 1990.

[16] G. Mecca and P. Atzeni. Cut and Paste. *Journal of Computing and System Sciences*, 58, 1999.

[17] G. Mecca, A. Mendelzon, and P. Merialdo. Efficient queries over Web views. In *EDBT'98*.

[18] G. Mecca, P. Merialdo, P. Atzeni, and V. Crescenzi. The (Short) ARANEUS Guide to Web-Site Development. In *WebDB'99*.

[19] D. Suciu. Managing Web data, 1999. Tutorial given at SIGMOD'99.

[10] Also, empty HTML tags – like `<BR>` – need to be specified as `<BR/>`; entities, like ` ` must become `#032` etc.

Storing and Querying XML Data using an RDMBS

Daniela Florescu
INRIA, Roquencourt
daniela.florescu@inria.fr

Donald Kossmann
University of Passau
kossmann@db.fmi.uni-passau.de

1 Introduction

XML is rapidly becoming a popular data format. It can be expected that soon large volumes of XML data will exist. XML data is either produced manually (like html documents today), or it is generated by a new generation of software tools for the WWW and/or electronic data interchange (EDI).

The purpose of this paper is to present the results of an initial study about storing and querying XML data. As a first step, this study was focussed on the use of relational database systems and on very simplistic schemes to store and query XML data. In other words, we would like to study how the simplest and most obvious approaches perform, before thinking about more sophisticated approaches.

In general, numerous different options to store and query XML data exist. In addition to a relational database, XML data can be stored in a file system, an object-oriented database (e.g., Excelon), or a special-purpose (or semi-structured) system such as Lore (Stanford), Lotus Notes, or Tamino (Software AG). It is still unclear which of these options will ultimately find wide-spread acceptance. A file system could be used with very little effort to store XML data, but a file system would not provide any support for querying the XML data. Object-oriented database systems would allow to *cluster* XML elements and sub-elements; this feature might be useful for certain applications, but the current generation of object-oriented database systems is not mature enough to process complex queries on large databases. It is going to take even longer before special-purpose systems are mature.

Even when using an RDBMS, there are many different ways to store XML data. One strategy is to ask the user or a system administrator in order to decide how XML elements are stored in relational tables. Such an approach is supported, e.g., by Oracle 8i. Another option is to infer from the DTDs of the XML documents how the XML elements should be mapped into tables; such an approach has been studied in [4]. Yet another option is to analyze the XML data and the expected query workload; such an approach has been devised, e.g., in [2]. In this work, we will only study very simple *ad-hoc* schemes; we think that such a study is necessary before adopting a more complex approach. The schemes that we analyze require no input by the user, they work in the absence of DTDs or if DTDs are meaningless, and they do not involve any analysis of the XML data. Due to their simplicity, the approaches we study will not show the best possible performance, but as we will see, some of them will show very good query performance in most situations. Also, there is no guarantee that any of the more sophisticated approaches known so far will perform better than our simple schemes; see [3] for some experimental results in this respect. Furthermore, the results of our study can be used as input for more sophisticated approaches.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Mapping XML Data into Relational Tables

The starting point is one or a set of XML documents. We propose to scan and parse these documents one at a time and store all the information into relational tables. For simplicity, we assume here that an XML document can be represented as an ordered and labeled directed graph. Each XML element is represented by a node in the graph; the node is labeled with the *oid* of the XML object[1]. Element-subelement relationships are represented by edges in the graph and labeled by the name of the subelement. In order to represent the order of subelements of an XML object, we also order the outgoing edges of a node in the graph. Values (e.g., strings) of an XML document are represented as leaves in the graph. In all, we consider six ways to store XML data (i.e., graphs) in a relational database: three alternative ways to store the edges of a graph and two alternative ways to store the leaves (i.e., values), resulting in overall three times two different schemes. Other schemes and variants of the schemes presented in this paper are described and discussed in [3]. In particular, we describe and evaluate a scheme in [3] which would take advantage of an object-relational database system's feature to store multi-valued attributes.

Representing XML data as a graph is a simplification and some information can be lost in this process. The reason is that our graph model does not differentiate between XML subelements and attributes, and it is does not differentiate between subelements and references (i.e., IDREFs). Using one of our schemes, therefore, the original XML document cannot exactly be reconstructed from the relational data. However, these simplifications can easily be alleviated with additional bookkeeping in the relational database.

In order to show how XML data is mapped into relational tables in each scheme, we will use the following XML example which contains information about four persons:

```
⟨person⟩ ⟨id='1' age='55'⟩
    ⟨name⟩ Peter ⟨/name⟩
    ⟨address⟩ 4711 Fruitdale Ave. ⟨/address⟩
    ⟨child⟩
        ⟨person⟩ ⟨id='3' age='22'⟩
            ⟨name⟩ John ⟨/name⟩
            ⟨address⟩ 5361 Columbia Ave. ⟨/address⟩
            ⟨hobby⟩ swimming ⟨/hobby⟩
            ⟨hobby⟩ cycling ⟨/hobby⟩
        ⟨/person⟩
    ⟨/child⟩
    ⟨child⟩
        ⟨person⟩ ⟨id='4' age='7'⟩
            ⟨name⟩ David ⟨/name⟩
            ⟨address⟩ 4711 Fruitdale Ave. ⟨/address⟩
        ⟨/person⟩
    ⟨/child⟩
⟨/person⟩

⟨person⟩ ⟨id='2' age='38' child='4'⟩
    ⟨name⟩ Mary ⟨/name⟩
    ⟨address⟩ 4711 Fruitdale Ave. ⟨/address⟩
        ⟨hobby⟩ painting⟨/hobby⟩
⟨/person⟩
```

2.1 Mapping Edges

2.1.1 Edge Approach

The simplest scheme is to store all edges of the graph that represents an XML document in a single table; let us call this table the *Edge* table. The *Edge* table records the oids of the source and target objects of each edge of the graph, the label of the edge, a flag that indicates whether the edge represents an inter-object reference (i.e., an internal node) or points to a value (i.e., a leaf), and an ordinal number because the edges are ordered, as mentioned above. The *Edge* table, therefore, has the following structure:

[1] We assume that every XML element has a unique identifier. In the absence of such an identifier in the imported data, the system will automatically generate one.

Edge

source	*ordinal*	*name*	*flag*	*target*
1	1	age	int	*v1*
1	2	name	string	*v2*
1	3	address	string	*v3*
1	4	child	ref	**3**
1	5	child	ref	**4**
2	1	age	int	*v4*
...	...	...	...	

V_{int}

vid	*value*
v1	55
v4	38
v8	22
v13	7

V_{string}

vid	*value*
v2	Peter
v3	4711 Fruitdale Ave.
v5	Mary
v6	4711 Fruitdale Ave.
v7	painting
...	...
v15	4711 Fruitdale Ave.

Figure 1: Example: Edge Table with Separate Value Tables

Edge(source, ordinal, name, flag, target)

The key of the *Edge* table is $\{$*source, ordinal*$\}$. Figure 1 shows how the *Edge* table would be populated for our XML example. The figure also shows one particular way to store values in separate *Value* tables; this approach is explained in more detail in Section 2.2. The bold faced numbers in the *target* column of the *Edge* table (i.e., **3** and **4**) are the oids of the target objects. The italicized entries in the *target* column refer to representations of values (explained later).

In terms of indices, we propose to establish an index on the *source* column and a combined index on the $\{$*name, target*$\}$ columns. The index on the *source* column is useful for forward traversals such as needed to reconstruct a specific object given its *oid*. The index on $\{$*name, target*$\}$ is useful for backward traversals; e.g., "find all objects that have a child named John." We experimented with different sets of indices as part of our performance experiments. We found these two indices to be the overall most useful ones.

2.1.2 Binary Approach

In the second mapping scheme, we propose to group all edges with the same label into one table. This approach resembles the binary storage scheme proposed to store semi-structured data in [5]. Conceptually, this approach corresponds to a horizontal partitioning of the *Edge* table used in the first approach, using *name* as the partitioning attribute. Thus, we create as many *Binary* tables as different subelement and attribute names occur in the XML document. Each *Binary* table has the following structure:

B_{name}(source, ordinal, flag, target)

The key of such a *Binary* table is $\{$*source, ordinal*$\}$, and all the fields have the same meaning as in the *Edge* approach. In terms of indices, we propose to construct an index on the *source* column of every *Binary* table and a separate index on the *target* column. This is analogous to the indexing scheme we propose to use for the *Edge* approach.

2.1.3 Universal Table

The third approach we study generates a single *Universal* table to store all the edges. Conceptionally, this *Universal* table corresponds to the result of a full outer join of all *Binary* tables. The structure of the *Universal* table is as follows, if $n_1, \ldots, n_k$ are the label names.

Universal(source, ordinal_{n_1}, flag_{n_1}, target_{n_1}, ordinal_{n_2}, flag_{n_2}, target_{n_2}, $\ldots$, ordinal_{n_k}, flag_{n_k}, target_{n_k})

Figure 2 shows the instance of the *Universal* table for our XML example. As we can see, the *Universal* table has many fields which are set to *null*, and it also has a great deal of redundancy; the value *Peter*, for instance, is represented twice because Object 1 has two *child* edges. (How values are exactly represented is described in the next section.) In other words, the *Universal* table is denormalized—with all the known advantages and disadvantages of such a denormalization. Corresponding to the indexing scheme of the *Binary* approach, we propose to establish separate indices on all the *source* and all the *target* columns of the *Universal* table.

source	...	ord_{name}	$targ_{name}$	...	ord_{child}	$targ_{child}$	ord_{hobby}	$targ_{hobby}$
1	...	2	*Peter*	...	4	**3**	null	null
1	...	2	*Peter*	...	5	**4**	null	null
2	...	2	*Mary*	...	4	**4**	5	*painting*
3	...	2	*John*	...	null	null	4	*swimming*
3	...	2	*John*	...	null	null	5	*cycling*
4	...	2	*David*	...	null	null	null	null

Figure 2: Example Universal Table

B_{hobby}

source	*ord*	val_{int}	val_{string}	*target*
2	5	null	*painting*	null
3	4	null	*swimming*	null
3	5	null	*cycling*	null

B_{child}

source	*ord*	val_{int}	val_{string}	*target*
1	4	null	null	**3**
1	5	null	null	**4**
2	4	null	null	**4**

Figure 3: Example: Binary Tables with Inlining

2.2 Mapping Values

We now turn to alternative ways to map the values of an XML document (e.g., strings like "Peter" or "4711 Fruitdale Ave."). We study two variants in this work: (a) storing values in separate *Value* tables; (b) storing values together with edges. Both variants can be used together with the *Edge, Binary*, and *Universal* approaches, resulting in a total of six possible mapping schemes.

2.2.1 Separate Value Tables

The first way to store values is to establish separate *Value* tables for each conceivable data type. There could, for example, be separate *Value* tables storing all integers, dates, and all strings.[2] The structure of each *Value* table is as follows, where the type of the *value* column depends on the *type* of the *Value* table:

V_{type}(vid, value)

Figure 1 shows how this approach can be combined with the *Edge* approach. The *vids* of the *Value* tables are generated as part of an implementation of the mapping scheme. The *flag* column in the *Edge* table indicates in which *Value* table a value is stored; a *flag* can, therefore, take values such as *integer, date, string*, or *ref* indicating an inter-object reference. In the very same way, separate *Value* tables can be established for the *Binary* and *Universal* approaches. In terms of indices, we propose to index the *vid* and the *value* columns of the *Value* tables.

2.2.2 Inlining

The obvious alternative is to store values and attributes in the same tables. In the *Edge* approach, this corresponds to an outer join of the *Edge* table and the *Value* tables. (Analogously, this corresponds to outer joins between the *Binary* and *Universal* tables for the other approaches.) Hence, we need a column for each data type. We refer to such an approach as *inlining*. Figure 3 shows how inlining would work for the *Binary* approach. Obviously, no *flag* is needed anymore, and a large number of *null* values occur. In terms of indexing, we propose to establish indices for every *value* column separately, in addition to the *source* and *target* indices.

3 Performance Experiments and Results

We carried out a series of performance experiments in order to study the tradeoffs of the alternative mapping schemes and the viability to store XML data in an RDBMS. In this paper, we present the size of the resulting relational database for each mapping scheme, the time to reconstruct an XML document from the relational data,

[2]XML currently does not differentiate between different data types, but there are several proposals to extend XML in this respect.

n	100,000	number of objects
f_n	4	maximum number of attributes with inter-object references per object
f_v	9	maximum number of attributes with values per object
s	15	size of a short string value [bytes]
t	500	size of a long text value [bytes]
p_s	80	percent of the values that are strings
p_t	20	percent of the values that are text
d	20	number of different attribute names
l	10	size of an attribute name [bytes]

Table 2: Characteristics of the XML Document

and the time to execute different classes of XML queries. Other experimental results such as bulkloading times and times to execute different kinds of update functions are presented in [3].

To simplify the discussion, we will only present experimental results for four of the six alternative mapping schemes described in Section 2. We will study the *Edge, Binary*, and *Universal* approaches with separate *Value* tables in order to study the tradeoffs of the different ways to map edges. In addition, we will study the *Binary* approach with inlining in order to compare inlining and the separate *Value* tables variants.

As an experimental platform, we use a commercial relational database system[3] installed on a Sun Sparc Station 20 with two 75 MHz processors and 128 MB of main memory and a disk that stores the database and intermediate results of query processing. The machine runs under Solaris 2.6. In all our experiments, we limited the size of the main memory buffer pool of the database system to 6.4 MB, which was less than a tenth of the size of the XML document. Other than that, we use the default configuration of the database system, if not stated otherwise. (For some experiments, we used non-default options for query optimization; we will indicate those experiments when we describe the results.) All software which runs outside of the RDBMS (e.g., programs to prepare the XML document for bulkloading) is implemented in Java and runs on the same machine. Calls to the relational database from the Java programs are implemented using JDBC.

3.1 Benchmark Specification

3.1.1 Benchmark Database

The characteristics of the synthetic XML document we generate for the performance experiments are described in Table 2. The XML document consists of n objects. Each object has $0..f_n$ attributes containing inter-object references (i.e., IDREFs) and $0..f_v$ attributes with values. The document is flat; that is, there is no nesting of objects. (Given our graph model described in Section 2, flat documents with IDREFs are stored in the same way as documents with nested objects.) All attributes are labeled with one of d different attribute names; we will refer to these names as $a_1, \ldots, a_d$, but in fact each name is l bytes long. There are two types of values: short strings with s bytes and long texts with t bytes. p_s% of the values are strings and p_t% of the values are text. We use a uniform distribution in order to select the number of attributes for each object individually and to determine the objects referenced by an object and the name of every attribute. The graph that represents the XML document contains cycles, but this fact is not relevant for our experiments.

Since the XML document contains values of two different data types (string and text), two *Value* tables are generated in the relational database for the mapping schemes without inlining and two *value* columns are included in the *Binary* scheme with inlining. We index the strings completely, as proposed in Section 2.2, but we do not index the text (for obvious reasons), deviating from the proposed indexing scheme of Section 2.2. Strings and text, as well as attribute names (in the *Edge* table) are represented as `varchars` in the relational database. *flags* are represented as `chars`, and all other information (e.g., *oids, vids, ordinals,* etc.) is represented as `number(10,0)`.

The parameter settings we use for our experiments are also shown in Table 2. We create a database with 100,000 objects. Each object has, on an average, two attributes with inter-object references and 4.5 attributes with values. So, we have a total of approximately 450,000 values; 90,000 texts of 500 bytes and 360,000 short strings of 15 bytes.

[3]Our licenses agreement does not allow us to publish the name of the database vendor.

Query	*Description*	*Feature*
Q1	reconstruct XML object with oid = 1	select by oid
Q2	find objects that have attribute a_1 with value in certain range	select by value
Q3	find objects that have attributes a_1 and a_2 with certain values	two predicates
Q4	find objects that have a_1 and a_2 with certain value or just a_1 with certain value	optional predicate
Q5	find objects that have a_1 or a_2 or a_3 with certain value	predicate on attribute name
Q6	find object that match a complex pattern with seven references and eight nodes	pattern matching
Q7	find all objects that are connected by a chain of a_1 references to an object with a specific a_1 value	regular path expression
Q8	find all objects that are connected by a chain of a_1 or a_2 references to an object with a specific a_1 or a_2 value	regular path expression with a predicate on the attribute name

Table 3: Benchmark Query Templates

Q1	Q2L	Q2H	Q3L	Q3H	Q4L	Q4H	Q5L	Q5H	Q6L	Q6H	Q7L	Q7H	Q8L	Q8H
9	11	1805	3	131	9	1386	50	5556	1	3	11	2309	37	4616

Table 4: Size of Result Sets of Benchmark Queries

3.1.2 Benchmark Queries

Table 3 describes the XML-QL query templates that we use for our experiments. The XML-QL formulation for these queries is given in [3]. These query templates test a variety of features provided by XML-QL, including simple selections by oid and value, optional predicates, predicates on attribute names, pattern matching, and regular path expressions. In all, we test fifteen queries as part of our benchmark. We test each of the Q2 to Q8 templates in two variants: one *light* variant in which the predicates are very selective so that index lookups are effective and intermediate results fit in memory, and one *heavy* variant in which the use of indices is typically not attractive and intermediate results do not fit into the database buffers. Specifically, we set the predicates on a_1 to select 0.1% of the values in the light query variants and to select 10% of the values in the heavy variants. The predicates on a_2 are always set to select 30% of the values. All predicates involve short strings only (no text). For our benchmark database, the size of the result sets for each of these fifteen benchmark queries is listed in Table 4. To execute these XML-QL queries, we translate them into SQL queries. How this translation is done for each mapping scheme is outlined in [3] and beyond the scope of this paper.

To get reproducible experimental results, we carry out all benchmark queries in the following way: every query is carried out once to warm up the database buffers and then at least three times (depending on the query) in order to get the mean running time of the query. Warming up the buffers impacts the performance of the light queries that operate on data that fits in main memory; warming up the buffers, however, does not impact the results of the heavy queries.

3.2 Database Size

Table 5 shows the size of the XML document and of the resulting relational database for each mapping scheme. The size of the XML document is about 80 MB. We see that even without indices every mapping scheme produces a larger relational database. The *Universal* approach, of course, produces the most base data because the *Universal* table is denormalized as described in Section 2.1. Comparing the *Binary* approach with and without inlining, we see that inlining results in a smaller relational database: no *vids* are stored in the inline variant and *nulls* which are produced by the inline variant are stored in a very compact way by our RDBMS. Looking at the size of the indices, we can see that indices can consume up to 40% of the space.

	XML	Binary	Edge	Universal	Bin.+Inline
base data	79.2	105.2	122.3	138.9	86.9
indices	–	71.1	85.6	76.7	52.7
total	79.2	176.3	207.9	215.6	139.6

Table 5: Database Sizes [MB]

	Binary	Edge	Universal	Bin.+Inline
Q1	0.036	0.023	0.074	0.024
Q2(l)	0.104/4.6	0.089/5.3	0.093/4.8	0.011/5.3
Q2(h)	15.7	83.0	62.1	0.644/5.5
Q3(l)	6.0	5.1	5.8	2.0
Q3(h)	15.8	133.7	70.5	3.5
Q4(l)	12.3	9.9	11.7	4.1
Q4(h)	32.0	255.7	132.9	6.7
Q5(l)	0.277/15.4	5.1	14.2	0.028/13.9
Q5(h)	48.6	148.1	185.8	14.8
Q6(l)	0.130/6.5	6.1	0.141/6.3	0.017/2.0
Q6(h)	17.0	123.7	63.7	3.3
Q7(l)	0.111/6.2	0.101/5.4	0.096/6.2	0.012/5.3
Q7(h)	16.8	221.5	62.7	1.060/6.6
Q8(l)	18.3	5.0	91.4	32.7
Q8(h)	47.2	392.0	206.9	36.3

Table 6: Running Times of Queries [secs]; Tuned/Untuned

3.3 Running Times of the Queries

Table 6 shows the running times of our fifteen benchmark queries for each mapping scheme. In most cases, the optimizer of the RDBMS found good plans with the default configuration. In some cases, however, we were able to get significant improvements by using a non-default configuration; for such cases, Table 6 shows the running times obtained using the untuned (default) optimizer configuration and the tuned optimizer configuration. Most of the improvements were achieved for light queries and by forcing the optimizer to use indices instead of table scans and index nested-loop joins instead of hash or sort-merge joins.

The main observation is that the best mapping scheme (*Binary* with inlining) shows very good performance. For all queries, the running time is acceptable. The reason is that today's relational query engines are very powerful, even if the queries involve many joins and recursion.

Comparing the alternative mapping schemes, we can see that the *Binary* approach wins over the *Edge* and *Universal* approaches and that inlining beats separate *Value* tables. Both of these results can be explained fairly easily. The *Edge* approach performs poorly for heavy queries because joins with the (large) *Edge* table become expensive in this case; in effect, most of the data in the *Edge* table is irrelevant for a specific query. For the same reason, the *Universal* approach with its very large *Universal* table performs poorly for heavy queries. In the *Binary* approach, on the other hand, only relevant data is processed. The same kind of benefits of a binary table approach have been observed in the Monet project for (structured) TPC-D data [1]; for XML data the benefits are particularly high. Explaining the differences between inlining and separate *Value* tables is even easier: inlining simply wins because it saves the cost of the joins with the *Value* tables. The results show that inlining beats separate *Value* tables even if very large values (such as text) are inlined. Inlining would also win if many different types are involved and *null* values are stored in a compact way by the RDBMS.

Q8(l) and to some extent Q1 and Q5(l) are exceptions to the above rules. Q8 involves a predicate on the attribute names. The *Edge* approach is attractive for such queries because such predicates can directly be applied to the *Edge* table. Executing such predicates involves the generation of an SQL `UNION` query which carries out duplicate work for the other mapping schemes.

3.4 Reconstructing the XML Document

Table 7 shows the overall time to reconstruct the XML document (and write it to disk) from the relational data for each mapping scheme. In all cases, it takes more than 30 minutes, and this fact is probably the most compelling argument against the use of RDMBSs to store XML data. All mapping schemes need to sort by oid in order to re-group the objects, and this sort is expensive in our environment (it is an 80 MB sort with 6.4 MB of memory). The disastrous running time for the *Universal* approach with separate *Value* tables can also be explained. The *Universal* table must be scanned $d = 20$ times (once for each attribute name) in order to restructure the data and carry out the joins with the *Value* tables. These observations indicate that it might be advantageous to store

Binary	Edge	Universal	Bin.+Inline
56m 52s	40m 56s	1h 41m 17s	32m 8s

Table 7: Reconstructing the XML Document

copies of the original XML documents in the file system in addition to loading the XML data into an RDBMS. In general, there is no way to store data to meet the requirements of all purposes. Depending on the workload, multiple copies in possibly different formats are needed – XML data is no exception to this rule.

4 Conclusion

We studied alternative mapping schemes to store XML data in a relational database. The mapping schemes we studied are extremely simple. Due to their simplicity, they will never be the best choices, but our experiments indicate that even with such simple mapping schemes, it is possible to obtain very good query performance. The only operation which had unacceptably high cost was completely reconstructing a very large XML document; more sophisticated mapping schemes, however, would show poor performance for this operation as well.

This study was only a first step towards finding the best way to store XML data. Our results can be used as a basis to develop and configure more sophisticated mapping schemes. Also, more experiments with different kinds of (real and synthetic) XML data are required. In addition, other characteristics such as authorization, locking behavior etc. need to be studied. Furthermore, performance experiments with OODBMSs and special-purpose XML data stores ought to be conducted. The XML document and the XML-QL and SQL queries we used for our experiments can be retrieved from the authors' Web pages.

References

[1] P. Boncz, A. Wilschut, and M. Kersten. Flattening an object algebra to provide performance. In *Proc. of ICDE*, Orlando, FL, 1998.

[2] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of ACM SIGMOD*, Philadelphia, PN, 1999.

[3] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report, INRIA, France, 1999.

[4] J. Shanmugasundaram et al. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of VLDB*, Edinburgh, Scotland, 1999.

[5] R. v. Zwol, P. Apers, and A. Wilschut. Modelling and querying semistructured data with MOA. *Workshop on Query processing for semistructured data and non-standard data formats*, 1999.

Microsoft's vision for XML

Adam Bosworth Allen L. Brown, Jr.
Microsoft Corporation
http://msdn.microsoft.com/xml/

Abstract

The primary premise of this paper is that, with the advent of the web, it is now possible to design a truly simple, flexible and open architecture which allows all applications, on all machines, on all platforms, to interoperate. Another key premise is that this architectural latitude will only be possible through the aegis of logical views where, conceptually, cooperating applications interoperate through agreed upon logical conventions. This contrasts with their acting directly upon either the database tables or methods of one another. We argue that XML is the central building block for such an architecture. Moreover, several key XML schemas and conventions will be required for this revolution to be realized. We devote the remainder of this paper to spelling out the various building blocks Microsoft believes are required.

1 The Vision

Applications on the web will be easy to make open and interoperable. Goods and services will be easy to find. Any customer will be able to (for example):

- discover all sites that have some used book s/he is seeking. Then order the book from one of the sites.
- vacation in another country for a month, and easily discover who can cut her/his lawn back at home on Saturdays. Then engage someone to do so.
- open a spreadsheet or her/his own custom pages or Java applications, and easily let any of them talk directly to any site that manages her/his portfolios. Then make changes to those portfolios.

In short, it will be easy to discover and interact with data and applications on the web.

2 Why should we believe this?

Whenever one thinks about a new architecture, one should consider what has already worked (or not). If the architecture is as sweeping a one as that required to allow applications to interact across the web, then it is reasonable to take lessons from the successes of the web itself, and as well as from the successes of server-based applications, both two- and three-tier.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

3 So, what *have* we learned from the web?

First and foremost, we have learned that it isn't enough for something to be possible. It must be easy, open and flexible. The web predates HTML of course, but until the advent of HTTP and HTML, it didn't really explode. Why? The answer, succinctly, is empowerment. Once HTML and HTTP arrived, more people could create and use applications more easily. The implementations were not necessarily optimal from the point of view of performance or even robustness. They were optimal from the point of view of ease of getting started. In short, they were "drop-dead" simple.

Many people point out the deficiencies of HTML, noting especially the sloppiness of its grammar. This is true, and even regrettable, but the fact is that the simplicity of the HTML model, where tags were not used abstractly, despite what every SGML book preached. That they were used concretely to describe intended look and feel made HTML immediately approachable. The fact that HTML simply ignored unknown tags made it easy, as well, since mistakes were silently ignored.

Now, as we all know, the lack of formality led to a mess in which few can implement an HTML engine since it is expected to maintain perfect visual and scripting fidelity with what is, essentially, an unwritten, arbitrary, and complex standard. We should also learn from this and retain the simplicity without the mess. Nevertheless, the key point is that HTML exploded because it crossed some threshold of simplicity, making easy things easy, and hard things possible. The lesson is that *simplicity* and *flexibility* beat *optimization* and *power* in a world where connectivity is key.

There is a second lesson which is key. Applications need to be assembled from (coarse-grained) components that can be dynamically loaded rather than from single, large monolithic blocks. In the HTML world, these components are pages. But this lesson applies more generally to applications no matter what the basis for componentization. The reason for this is simple. The application starts more quickly, only consumes the resources it really needs, and, most importantly, can be dynamically loaded from the web. Why is this so important? It is important because of the need to *deploy*. Applications that can be dynamically loaded from a central place don't require some massive, complex and difficult installation processes with access to clients' machines. Note that Java *per se* doesn't give us this. It is easy, as anyone who has built a large and complex Java application can attest, to build one, which requires literally hundreds of classes to run. Such a monolithic application has the very deployment problem we wish to avoid. HTML had the serendipitous effect of forcing application designers to partition their applications. To repeat, the lesson is that applications should be loaded in coarse-grained chunks.

4 What have we learned from servers?

We use a simple analogy here. The analogy is the corner grocery store. Imagine that there is such a neighborhood store and everyone buys her or his weekly groceries there. Now imagine that all of a sudden, everyone's buying patterns changed. Instead of buying a few days' supplies at a time, they bought a single item at a time. In short, a customer would come in, buy a quart of milk and leave. Then return, buy a stick of butter and leave. Then return, buy a bag of apples and leave. And so on. In short order, there would be huge lines trying to buy their groceries at the store. If each time, the customers were buying their goods using electronic cards or checks, the situation would be even worse. Conversely, suppose they were to panic about an impending inflation, coming in to the store and trying to buy a year's supply of groceries. An entirely new and different set of problems would surface. Happily customers don't behave in either extreme fashion.

It turns out that servers are much like grocery stores. Servers can only handle so many clients at once (check out counters). More clients mean longer queues whether at a grocery store or on a server. Servers cannot scale beyond a certain point (you cannot build checkout counters on the fly), and servers cannot just run more and more processes concurrently. If processes increase in an unbounded fashion, their eventual competition for the CPU increases overheads, and overall performance actually degrades.

The cost of starting a new conversation with each customer (client)–or finishing one–has very real costs. Furthermore, if the transactions are too fine-grained, these costs can dwarf the other costs. Moreover, servers aren't designed to serve up really enormous quantities of data (a year's supply of groceries) in a single transaction. Servers bottleneck, TCP/IP complains, and so on. We also learned that processes shouldn't hang onto state while waiting on clients, or other slow processes. As an example, if the clerk doesn't know the price of an item, it would be unfortunate if the clerk simply stopped until the price were found or went her-/himself to find retrieve the posted price. The whole queue would block. Instead, the clerk should get someone else to find the price *and* start helping the next customer if the estimated delay will be significant. In server-speak this means that the process shouldn't hang onto state. What we learned from servers is that they should conduct coarse-grained, interruptible conversations with their clients.

This is very important. It means that when talking to a medical system, or a bank's payments system, or a purchase order system, the client should exchange information in coarse-grained fashion–chunks large enough to let the client go away and do useful work for a while. Indeed, often the serving process should unload all of its state information to the client, and reload it on return. This allows the distribution of state over all the served clients. Combined with the partitioning of applications into components, we can minimize the state information carried by the client, and in need of recovery by the server. In short, it means that good application design for talking to applications on servers involves relatively few methods that can accept and return complex sets of information such as the complete patient record, or the complete portfolio description, or the complete purchase order. This is not how we normally design classes and their methods. Typically, we design for encapsulation, making every access to every property or element a method invocation.

There is another key lesson to be learned from servers. Server software is designed to provide service efficiently and quickly. This is achieved, in part, by sharing transaction protected resources like the databases that underly systems such as those for payment authorization or airline reservations. Server software typically assumes that it is connected through distributed transactions to these resources. This in turn obliges rapid turn-around. Why? Because while any process is holding an open transaction involving these resources, the latter are locked and unavailable to other clients, perhaps even for reading. Notice that any attempt to ameliorate performance by a broader distribution of server code connects more processors to the transacted resources. If a banking transaction, for example, is debiting an account from one system and crediting an account from another, all inside of a distributed transaction, it must not run over slow or unreliable lines. To do so would increase the risk of locking resources for longer or indefinite periods of time. This means that the code running on the client should not be running within the same transaction as that running on the server. It shares the same information (purchase orders, portfolios, personnel records), but it is code dedicated to some different process such as letting the user view the information. The lesson is, in short, to move (to the client) the information, and possibly some simple portable validation logic for the information, but not the custom code that handles this information on the server. We tend to refer to this kind of distribution as an object to object bridge. Remote procedure call facilities are a traditional mechanism for effecting such bridges. So, to repeat, share data, not code.

To recapitulate, the lessons to be learned are as follows:

- Simplicity and flexibility beat optimization and power when connectivity is key.
- Applications should be partitioned into, and dynamically loaded in, coarse-grained chunks.
- Servers should conduct coarse-grained, interruptible conversations with their clients.
- The distributed interoperation among servers and clients is best supported by the sharing of data, rather than code.

5 Microsoft's opinion

XML is a tailor-made vehicle to move the sort of coarse-grained information (that we have been describing) around the web, whether the information is purchase orders, personnel records, portfolios, or just parameters. It is open, easy, and flexible. It is self-describing through DTDs and (eventually) through XML schemas. When the proposals before the W3C XML Schema Working Group[1] coalesce into a standard, such schemas will support rich extensible datatypes and additional metadata that an application will need in order to understand how to use or interpret data presented to it as an XML document. Virtually any logical view's data can be described and transported using XML. Indeed, we view XML as the *lingua franca* for gluing together the computation, communication and storage components of web-distributed applications.

6 What is left to do to enable this interoperation?

6.1 Enhance XML to support extensible and rich typing and metadata

The logical views described by the data that applications will deliver or accept must include a rich and extensible model for data types. As soon as one tries to use XML, even for something really simple like transporting the arguments that an RPC implies, or transporting a simple set of rows from a relational database, one discovers the need for data types. The recipient needs both to be able to discover the data types, and to be able to reliably interpret them as types appropriate to each programming language. Furthermore, the implied negotiation process will often involve the conveyance of other metadata important for the recipient. For example, if one transmits task descriptions as part of a project description, one may have a default Java class or COM class that should be materialized from this information. Similarly, a middleware search engine may want to get enough metadata about the elements to know which should be shown in a summary view, and which elements should be used as links to related information–and how. In short, the amount of ancillary information that might be desired for XML data is essentially unlimited. This fact argues for a the extensibility of the XML mechanisms for describing XML document instances. While the current XML DTDs *do* allow for describing multiple classes of document instances, the specification language for DTDs is not itself extensible. It is this sort of meta-extensibility that *is* provided by XML-Data, and *will* be provided by the eventual XML Schema standard.[2]

6.2 Describe the schemas

We believe that robust, scalable web-distributed applications will be built around a message passing paradigm. In order for us to enable the interoperation and graceful evolution of these applications, there will need to be explicit descriptions of the messages interchanged. Indeed, we can think these messages as being instances of various languages, where the grammars for those languages are given as XML schemas.

In order to motivate our ideas about schemas and schema extensibility, we ask the reader to imagine a scenario in which a customer is trying to search for providers of used books. The customer has her/his heart dearly set upon some out of print tome by Thomas Carlyle, and wants to know which sites have this book. An eager entrepreneur has decided to deliver such a service, and has managed to convince many used book providers, and even some online book retailers, to publish their inventories in a form specified by a particular XML schema. Instances of this schema will describe books. It need not even be a particularly good or comprehensive schema for describing books. Remember that sites can support multiple schemas (logical views) against a single implementation.[3] The enterprising entrepreneur has also convinced Yahoo to point to the site for book searching. This, in turn, lends a certain urgency to the book purveyors to actually publish a book description schema. Actually, several other schemas are necessary to this enterprise. What are they?

---

[1] Microsoft is an active participant in the Working Group, having contributed many submissions both formal and informal.

[2] There are three layers to be considered here. There is an XML document instance, an XML schema for that document instance, and a schema (the *meta-schema*) for describing XML schemas. It is commonly agreed in the XML Schema Working Group that both the schemas *and* the meta-schema should themselves XML document instances!

[3] The fact that book descriptions are explicitly schematized means that books descriptions can be marshalled according a schema in the request.

First, of course, there must be a schema for the book descriptions themselves, and most likely a schema for describing the book selling capabilities of sites. Instances of the latter schema would describe a site's physical ability to deliver goods and take payments. Today's XML DTD mechanism is sufficient for the required schematization.

The next schema to be considered is a schema for the services offered. Instances of such schemas (catalogues) provide a way for a search engine to quickly and efficiently determine for each site whether that site has any URL that accesses the book selling services. In addition to the catalogue schema, some protocol or convention will be required for launching the search and returning results. The catalogue schema is one that enumerates which URLs return which book schemas (if any). The protocol or convention is either a magic URL or an HTTP header or verb. This is used to make it clear to the site that the desired information is the list of resources with the filtering restrictions, if any (e.g. which resources support the book schema), that are being passed in through a particular XML schema. Thus we are likely to need yet other XML schemas, one to describe a filter, and one to present a logical view of a site in terms of URLs and schema that URLs can return. Let's call this latter one "Site description" and the former one "Filters" for the moment.

Third, let's assume that the search engine has a way to now discover which URLs return book schemas. Now the search engine may do one of two things:

1. Pull down the inventory and merge it into a database, maintained by the search engine, and tracking which sites have which books. In this case the search engine would like some additional information. It would like to know if the book-purveying site would let the search engine "subscribe" to this URL. In this case, the search engine is likely to want to "subscribe" to the site. This would mean that, from time to time, the site would be told that the inventory on hand has changed. The site would then either ask for the entire inventory over again or ask for essentially a "difference" to the inventory already on file. If the site did the latter, we would need yet another XML schema to describe such difference messages. We'll call this schema "Updates". Presumably, the search engine would discover whether the book- purveying site was willing to accept subscriptions to the URL through the same "Site description" schema that described the URL in the first place.

2. Remember that the search engine site is tracking books. Consequently, the search site would like to find out some additional information. It would like to ask whether the book purveyor site was willing to accept "Filter" requests (as specified by an appropriate schema) asking for filtered sets of these books. Then, if a request for books came in, the search engine would ship such a request directly to the book-purveying site using the "Filter" schema. This model is slower of course, but is a lot easier to implement and less likely to fail.

In either case, as we hope this example shows, "published" XML schemas for describing sites, for asking for filtered subsets and for sending changes are likely to be extraordinarily useful.

However, this model may sometimes be too simplistic. Let's imagine that the site isn't willing to support a general query language, no matter how limited. What it *is* willing to do is expose certain URLs which, when sent the appropriate parameters of Title or Author, return a list of books for those Titles or Authors or both. Notice that this is basically a remote procedure call. Now, even in this case, the search engine would want to send the parameters to the book-purveying site in a simple, easy to engineer manner. If the "Site description" schema assumed above describes the desired shape of the parameters of the "method" and there is a standard schema for marshalling parameters for RPCs, or the "Site description" schema describes the form of the search request, then the search engine knows what XML to send. Ideally, the marshalling schema for RPC will also describe how synchronously the results may be returned and by what mechanism to allow for delayed return. Note that all of the schemas mentioned thus far are well within the purview of facilities already specified in the working draft of the XML Schemas Working Group.

There is another standard language (amenable to XML schematization) that shows great promise for our imagined application. This is XSL or Extensible Stylesheet Language. The search engine we have posited has now developed lists of books. But there is another challenge that faces the search engine: how does it handle the case where different sites support different book schemas? Suppose that there are several competing schemas, or even different releases of an evolving schema. The search engine would like to note them all, and be able to convert between them into whichever common one it prefers. XSL provides mechanism for specifying and performing such transformations.

Similarly, the client receiving the results concerning book availability has a challenge. How does the client dynamically generate the HTML that would be required to display the book list in which vendors might want to include as links back to their sites, or the little blurbs about their site, and so on? The client could, of course, take direct advantage of DHTML and JavaScript, or Java, or any COM language to solve this problem, writing either script or a component to custom generate the appropriate DHTML from the data. The client could also use the data binding features of DHTML. But it would be handy to have a standard way to describe how to build the HTML from the XML, and a standard component (converter) that knew how to execute such instructions. Again, XSL can provide such a transformation.

Lastly, the site vending books has a problem. How does it map its book information, undoubtedly stored as a relational database, into the appropriate XML logical view? Interestingly, the XML schema for the view could itself contain suitable metadata to help compute the answer to this question. For true interoperability such a metadata decoration of an XML schema should be standardized. Even so, undoubtedly a "plan" would be computed for actually building the appropriate XML logical view and it would be an instance of an XML schema, call it the "Database conversion". Also required would be a general converter which given such a schema, would actually talk to the database, submit the requisite SQL, and build the appropriate XML. This model would be extremely useful on any server mapping relational data to HTML clients, even sometimes building the HTML directly.

So, to recapitulate, the following XML schemas would be necessary:

- "Site description" schema: Instances of such a schema would describe the "methods" of a site and the "schemas" it returns. Extensions to this model would also be needed to determine for any of these returned schemas, whether the site is also willing to accept the "Filters" or the "Updates" schema.
- "Updates" schema: An XML schema whose instances describe changes which have been made or should be made to an information repository.
- "Filters" schema: An XML schema whose instances describe desired subsets of an XML logical view.
- "RPC" schema: An XML schema whose instances describe signatures of methods supported by a particular RPC interface.

The following schemas and augmentation of schemas would be useful:

- XSL: It would be helpful to pay some attention to ensuring that the part of XSL that is a tree transformation language is sufficiently easy and powerful.
- Standard metadata in the XML schema language to describe mappings of elements to relational stores.
- "Database conversion" schema: A schema whose instances specify the construction of logical XML views from relational databases.

6.3 Ensure that the programming models for XML stay simple and appropriate

Any model for moving information around the web will require that two classes of applications developers be able to access this information, developers using serious structured languages such as Java or C++, and developers using scripting languages such as JavaScript. The requirements for the two classes are not identical. The scripting programmer tends to have neither pointers nor types, and to want to treat XML as a single big tree of nodes which can be navigated using a collection syntax. Something like `root[foos[3][bars[2][sams[1]]]` would mean that the script writer wanted the first node with tagname `sams` within the second node with tagname `bars` within the third node with tagname `foos` within the `root`. By overloading constructs that are basic to most scripting languages, such a model can be surfaced without requiring all XML data emitters to provide and maintain such indexed collections. What this user code would really translate to is intermediate language code that finds the indicated node(s) using simple enumerators. Conversely, the C++ and Java developers will probably happily deal with a lightweight enumeration model, and build their own object models on top of it, rather than depending upon the layers provided by the XML emitter. Indeed, in many cases, these developers may simply want the nodes pushed out to them, as they will be building their own data structures. It is important that we not overburden each implementation with some top-heavy API which is neither fish nor fowl, but rather build the right low-level API on top of which all can build.

What will be required, in addition to simple enumeration, are ways to extract data based upon the types described in the associated XML schemas, ways to make changes to the data or tree, and ways to validate that the changes conform to the schema of a target XML document. Lastly, as XML schemas described above emerge, facilities to process against them automatically will start to be expected as a service layer on top of the base API. The underlying model, however, allows the implementors to support them directly and natively. In other words, while a service layer might be necessary to find some node using simple predicates, a provider might natively implement support for this in a more efficient manner using internal indexes or hash tables or what have you.

6.4 Stores

We are often asked about XML "stores". We believe that it is unlikely that there will be a single XML store. Different stores will serve different purposes. The cheapest store, of course, is the file system. Many standard components are springing up to provide DOM (Document Object Model) API access to XML stored in streams or files. This includes, of course, our own component which we will make available ubiquitously on all Windows platforms, on Unix, and in Java.

Relational stores are superb at exposing multiple distinct logical views on the same data, and by now have very good scaling and transactional characteristics. Typically, mission critical data will live in a relational store. But this doesn't prohibit the existence of "converters" from relational stores to logical views presented as XML document instances. We expect to see explosive growth in the provision of such views. Ultimately, we would expect that the database stores themselves would be able to make these conversions happen, but that in the near term these converters would be part of middleware living on middle tier servers. Elucidating requirements and presenting the likely successful architectures for such XML view services merit a paper in their own right.

What about object-oriented stores? Many such stores have been applied to new purposes unanticipated by their designers/marketers. These systems are being deployed as "staging" stores for XML data as it is cached on the middle tier or the client. Some will undoubtedly turn out to do an excellent job of providing XML caching. What will be critical here, if we want to see interoperability, is that we avoid a profusion of APIs for talking to these caches. The XML DOM standard helps here by describing how to talk to a particular XML document/object, but doesn't handle the larger issues that are likely to be of concern in caches: versioning, transactions, searching, and so on. We hope that the distributed authoring and versioning (DAV) standard will play a major role in driving convergence in this arena. We also expect the object-oriented technology companies to play key roles here.

6.5 Converters architecture

We believe that for the foreseeable future most data will *not* be stored in XML. Indeed, the main thrust of this paper is that XML acts as a convenient mechanism for fostering application integration at the logical view level, not the physical level. Some data will come from relational databases. Some from Teletype feeds. Some from mainframe databases through CICS. Some from SGML. Some from object-oriented databases. Some from mail stores. Much will be synthesized dynamically by objects written by programmers. But if we want a model for interoperation, we will need a standard component model for transforming between any data whatsoever and XML. We call such components *converters*.

A typical middle tier server will have mechanisms for connecting such converters to queues of XML messages. For example, clients may be sending requests for available classes to a university and expecting a specific logical view (as described by an XML schema). The server at the university would dequeue such a request, pass the XML request to the appropriate converter, probably hooked up to the class schedule (a database on the back end), take the resulting XML, and route it back to the requester. The requester/recipient would display the information, review it, and then, perhaps, want to enroll. The requester would send a message described by yet another ("enroll") schema to the middle tier. The middle tier would dequeue this message and quite possibly hand it off to a completely different converter component talking to a backend student database.

We expect to help produce three sorts of converters to and from XML. These sorts are distinguished by the level of granularity at which they operate and carry out conversions.

6.6 The programming object level

This requires a component model abstracting the concept of an XML converter. These converters will enable the rendering of objects as XML. They will do so by being able to pass back and forth between structures described by object class definitions and XML structures described by schemas. Such converters will also need to act a lot like any other XML provider. They will deliver up XML on request. They will support some standard discovery mechanism such as the "Site discovery" schema. They will need to be able to stream results out. We need to make it easy for people to build such components, even using scripting languages.

6.7 The database level

An extraordinarily common case will be mappings between XML logical views and SQL databases. This is sufficiently common that it should be possible to author such transforms without having to write code or even understand the database schema(s). Indeed, as we suggested earlier, it should be possible to simply send a request for a particular schema described using the syntax of the forthcoming XML schema standard.

6.8 The XML level

XSL has been traditionally viewed as a component that took XML in and emitted nicely printed output using whatever print-medium was chosen. We have looked at XSL somewhat differently. Our chosen output medium is DHTML, whether for dynamic interactive output or for producing rich Office documents. However, we don't want to require that everyone who is trying to produce user a interface for XML be a programmer. We believe that it should be easy to produce DHTML from XML.

Furthermore, we believe that there will not be universal agreement about schemas. It is possible, for example, that 3 or 4 or 5 schemas (rather than one) will be used by sites providing information about their book inventories. However, any sensible program accessing or viewing such data will want to pick a single one. This will require a mechanism for converting from one XML schema to another. If sufficient semantic information is available, it will be possible to do this automatically. But frequently, such information will not be available, and a customer will have to describe how to convert. Again, we don't want to limit this to programmers. We believe it should be easy to produce XML from XML.

Hence, we want a standard extensible schema for converting from XML trees to either other XML trees or to DHTML trees. To Microsoft, the interesting part of XSL is the part that helps define how a given XML tree should be translated into a quite different tree.[4] We anticipate standard converters which use XSL instances to

[4]Indeed, the W3C XSL working group has recently aligned itself this way as well, splitting the standard into transformation and rendering components.

decide how to translate from instances of one XML schema into instances of another, or into DHTML.

7 Implications for the web

XML is the basic building block. A suitably rich XML schema language is the required next step. Then, as we have seen, we do need agreement upon some specific XML schemas and we must remember to keep these schemas simple, open, and easy. If we do all this, the power we unleash to the typical developer is incalculable. It becomes possible to build systems for any line of business application, for collaboration, and systems for intelligent information retrieval including for goods and services.

This revolution will do for applications what SQL partially did for databases. It will open up the flood gates because the number of people who can interact with data in any form will increase enormously. Many difficult problems remain to be worked out, including the details of these schemas, the security issues, transactions, and so on. But in the spirit of the web, we have submitted our proposals for most of these pieces to allow us to start and learn, rather than trying to get it perfect and never shipping.

But the revolution involves not just the "how", but also the "what". Well behaved sites will deliver not only complete corpora of data, but filtered sets of data, updates to data (notifications), and requests to update their data (data entry). In all these cases, they will support these services using standard XML schemas. It will open up an entire new business, namely converters with a myriad of specialized XML schemas that make sense for particular types of transformations.

8 Predictions

The requisite schemas will be defined and implemented. The DOM will converge and streamline. Many database storage systems will start to support both the schemas and the DOM. XML will become a widespread solution to interoperable RPC on the net. Microsoft will actively support all of the above.

Tools will emerge to:

- help author the schemas, be they for retrieval, update, or description;
- view, edit, and manage XML;
- define mappings between XML logical views and databases;
- help programmers use all these components and automatically inter-operate with these components.

Server components will emerge to:

- act as stores and managers and queues for XML;
- act as converters between XML and relational databases;
- act as converters between XML instances described by different schemas;
- marshal objects directly from XML.

The programming model for building applications will change to one in which XML is the standard message format used for transmitting data and for RPC. There will be standard ways to manage these messages through queues, and to register events for handling these messages. Components on the server will routinely save their state into XML data-stores virtually transparently.

9 Conclusions

We're only at the very beginning of the web revolution. The most exciting part is still to come. Soon it will become as easy to interact with programs and data all over the net as it currently is to view shared presentation and content.

Data Management for XML: Research Directions

Jennifer Widom
Stanford University
widom@db.stanford.edu, http://www-db.stanford.edu/˜widom

Abstract

This paper is a July 1999 snapshot of a "whitepaper" that I've been working on. The purpose of the whitepaper, which I initially drafted in April 1999, was to formulate and put into prose my thoughts on the research opportunities XML brings to the general area of data management. It is important to know that **this paper is not a survey**. *It offers my personal opinions and thoughts on Data Management for XML, fully incorporating my biases and ignorances. Related work is not discussed, and references are not provided with the exception of a handful of URLs. Furthermore, I expect the whitepaper to evolve over time; please see [1] for the latest version.*

1 The XML Revolution

XML—the *eXtensible Markup Language*—has recently emerged as a new standard for data representation and exchange on the Internet [2]. The basic ideas underlying XML are very simple: tags on data elements identify the meaning of the data, rather than, e.g., specifying how the data should be formatted (as in HTML), and relationships between data elements are provided via simple nesting and references. Yet the potential impact is significant: Web servers and applications encoding their data in XML can quickly make their information available in a simple and usable format, and such information providers can interoperate easily. Information content is separated from information rendering, making it easy to provide multiple views of the same data. (XML data files can be rendered via specifications in *XSL*, the *eXtensible Stylesheet Language* [3].) Laborious, error-prone, and unmaintainable "screen-scraping" as a method for extracting useful data from HTML Web pages is greatly reduced, since XML is designed for data representation—XML is simple, easily parsed, and self-describing.

As an example, consider the following HTML fragment (extracted from my own publications Web page [4] without modification), describing two publications:

```
<UL>
<LI>
R. Goldman, J. McHugh, and J. Widom.
<A href="ftp://db.stanford.edu/pub/papers/xml.ps">
From Semistructured Data to XML: Migrating the Lore Data Model
and Query Language
</A>.
Proceedings of the 2nd International Workshop on the Web and Databases
```


Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

```
(WebDB '99), pages 25-30, Philadelphia, Pennsylvania, June 1999.
<LI>
T. Lahiri, S. Abiteboul, and J. Widom.
<A href="ftp://db.stanford.edu/pub/papers/ozone.ps">
Ozone: Integrating Structured and Semistructured Data
</A>.
Technical Report, Stanford Database Group, October 1998.
</UL>
```

One way of encoding the same information in XML is:

```
<Publication URL="ftp://db.stanford.edu/pub/papers/xml.ps" Authors="RG JM JW">
   <Title>From Semistructured Data to XML: Migrating the Lore Data Model
       and Query Language</Title>
   <Published>Proceedings of the 2nd International Workshop on the Web
       and Databases (WebDB '99)</Published>
   <Pages>25-30</Pages>
   <Location>
      <City>Philadelphia</City>
      <State>Pennsylvania</State>
   </Location>
   <Date>
      <Month>June</Month>
      <Year>1999</Year>
   </Date>
</Publication>
<Publication URL="ftp://db.stanford.edu/pub/papers/ozone.ps" Authors="TL SA JW">
   <Title>Ozone: Integrating Structured and Semistructured Data</Title>
   <Published>Technical Report</Published>
   <Institution>Stanford University Database Group</Institution>
   <Date>
      <Month>October</Month>
      <Year>1998</Year>
   </Date>
</Publication>
<Author ID="SA">S. Abiteboul</Author>
<Author ID="RG">R. Goldman</Author>
<Author ID="TL">T. Lahiri</Author>
<Author ID="JM">J. McHugh</Author>
<Author ID="JW">J. Widom</Author>
```

Clearly the XML encoding, although more verbose, provides the information in a far more convenient and usable format from a data management perspective. Furthermore, the XML data can be transformed and rendered as desired using simple XSL specifications.

There is great excitement in industry over XML. True believers think XML will radically change the face and uses of the Web. Leading software vendors are committed to XML and are quickly moving towards using XML internally as well as creating XML-oriented tools and products. XML technology startups are proliferating. Commercial enterprises and scientists alike are generating their data in XML, and we expect that the amount and variety of data made available in XML form, and the tools to accompany that data, will grow rapidly.

2 Commercial Perspective (A Disclaimer)

The remainder of this paper is written from a true research standpoint. For better or worse, I've ignored or cast aside certain important considerations from a commercial perspective. For example:

- Although it is clear that XML will have a significant impact on Internet information management, it is still unclear precisely how XML will be used. Will XML be used primarily as a data *exchange* format, or will it also be used as a data *storage* format? Will most XML documents be governed by *Document Type Definitions* (*DTDs*) or will many be without? For that matter, how important will the concept of an XML "document" be? It may be years before the answers emerge. Nevertheless, we researchers can enjoy the freedom of attacking problems associated with any or all of the possible outcomes. As illustrated in the next section, I believe database-style technology applied to XML can play an important role in all of them.
- Despite all of the hype, including my own, XML will not be the proverbial magic bullet that solves all of the problems associated with application interoperation and data integration on the Internet. Some applications, even if they encode their data in XML, may not wish to expose it that way, so "screen-scraping" will not disappear entirely. (In fact, some information providers purposely make their current HTML pages difficult to parse, in order to protect the information from being copied—these providers certainly won't expose their data in XML even if they store it that way.) Furthermore, there is the very significant issue of ensuring that applications use commonly-understood tags for data, or at least have practical methods and tools for detecting and resolving discrepancies among tags. Fortunately, in many cases Web information providers *will* have a vested interest in cooperating with each other, so we anticipate that such methods and tools, as well as agreed-upon, domain-specific DTDs, will proliferate.
- This paper is based on the core XML specification. It does not consider the various proposed mechanisms for inter-document references (e.g., *XLink*, *XPointer*), although it's my feeling that the differences among these mechanisms will have little significant impact on database-oriented XML research. The various proposed extensions or alternatives to DTDs for richer schema definitions (e.g., *DCDs*, *XML-Schema*, and others) also are not discussed here. These schemes obviously can have an impact on database-oriented XML research (and database researchers probably should try to have an impact on these schemes), but the field is too crowded at this point to choose a particular winner. Also, I believe that most of the technical issues discussed below with respect to DTDs are equally valid for richer schema definition languages.

3 Database Research Opportunities

In addition to the promise of greatly facilitating information integration on the Internet, as discussed in Section 1, another exciting promise of XML is that it "turns the Web into a database." Migrating Web information to XML is a significant first step in enabling efficient execution of ad-hoc, expressive queries over large amounts of Web data—a core feature of traditional database management systems. Consider the current state of query processing over information on the Web:

- Data embedded within HTML pages needs to be preprocessed by special-purpose, page-specific parsers before meaningful queries can be posed, a limited technology at best. Otherwise, ad-hoc queries are limited to simple keyword-based searches (as provided by search engines, for example) that understand documents as streams of words and little more.
- Data stored within traditional database management systems generally is accessed on the Web only through simple and rigid forms-based interfaces.

Most of us are familiar with Web sites that contain vast databases of useful information, but whose query and search facilities are surprisingly primitive. As a specific query scenario, consider a large collection of publication information such as that in the examples of Section 1, drawn from one or more data sources.

- If the information is provided in HTML, we could attempt to parse out the relevant data elements, provided the formats don't change and our parsers are amenable to the inevitable inconsistencies and omissions across publications. Otherwise, we are limited to keyword-based searches.
- If the information is provided via a traditional DBMS, then simple, fixed, parameterized queries are the usual mode of external access. Query capabilities aside, much of the information made available on the Web is not well-structured (even our tiny examples in Section 1 illustrate some of the problems), and thus the data may not be amenable to using a traditional DBMS.
- Suppose the information is provided in XML, and let us be optimistic and assume that if multiple data sources are involved, the XML encodings are compatible (see Section 2). In this case, the structure and "meaning" of the data (at least to the extent that meaning can be embodied in tags), as well as the data itself, is readily parsable and available, setting the stage for powerful queries. Some relatively simple examples of such queries are:
  - *Find all authors with two or more SIGMOD publications in the same year.*
  - *Find the earliest publication with "semistructured data" in its title.*

Encoding information in XML is a first step to enabling expressive, database-like queries over the information, but many query processing issues still need to be addressed, as discussed in the bullets below. Furthermore, the tendency to mix traditional data elements with free text in XML, the ability to encode data ranging from fully structured to highly unstructured, and the inherent dichotomy between documents and databases, poses new challenges in combining techniques from database systems and information retrieval.

A few sample research topics for the database community follow, with a primary focus on database-like treatment of XML (as opposed to focusing on XML-based information integration, clearly a very important topic as well). There has been preliminary work in several of these areas, while some topics are still virtually untouched.

- Since XML is a document format and not a data model, we need the ability to map XML-encoded information into a true data model.
- More generally, we need to resolve the various conflicts that arise when we try to mix the concepts of documents and databases. For example, while some applications may wish to view a large set of XML documents as exactly that—a set of documents—other applications may prefer to think of each document as a database "load file," where all document contents are merged into a single large database. In fact, we may wish to simultaneously view a body of XML information in both ways.
- Theoretical results and practical techniques for designing XML databases are needed, to the extent possible within the relatively free-form nature of XML. For example, when should attributes be used and when should subelements be used? Is a one-to-one relationship best represented using element nesting or IDREFs? Is there an analogy to relational functional dependencies in the XML world? There are a number of questions of this nature that arise when translating a conceptual model of a database into an XML encoding.
- The relationship between XML's optional Document Type Definitions (DTDs) and traditional database schemas needs to be understood and exploited.
- An appropriate query language (or set of languages) for XML needs to be defined. This task is made particularly difficult because the true requirements for XML query languages will not be known until a significant number of data-intensive XML applications are built. Here too, there is an inherent conflict between the document and database view of XML-encoded information, and thus an opportunity to merge formerly separate technologies.
- Database updates in an XML setting must be considered, with a focus on environments that are heavily read-oriented.
- Efficient physical layout and indexing mechanisms are required for large stores of XML data. At the same time, we should be able to provide the illusion of an XML data store when the data actually is stored elsewhere (such as in a traditional DBMS), and make the two modes work together.

- All facets of traditional query processing must be considered, from semantic checking (when appropriate) through plan generation and optimization to efficient access methods. We also need to consider non-traditional query processing that can meaningfully and efficiently handle a mixture of data elements (both structured and semistructured) and free text.
- *View* mechanisms are important in conventional databases, and are likely to be important in XML databases as well. Virtual views, involving query rewriting techniques, are likely to be quite a bit more complex in XML than in the relational world. Similarly, the incremental maintenance problem for materialized views is likely to pose new problems when we consider XML data. Even the view definition language itself needs to be considered carefully. At one extreme, *XSL* [3] could be used as a view definition language, posing especially challenging view management problems due to its expressive power and procedural nature.
- Everything needs to scale to Web proportions (!).

4 The Lore Project at Stanford

The *Lore* project at Stanford [5] began around 1995, with the premise of building a complete database management system for *semistructured data*. We defined semistructured data as data that may be irregular or incomplete, and whose structure may change rapidly and unpredictably. Information integrated from heterogeneous sources, structured text, and "screen-scraped" HTML pages were our original motivating sources of data for Lore.

By 1998 we had largely achieved our goal. We had built a complete, robust (as university prototypes go), multi-user database system based on a fairly traditional DBMS architecture, with a number of extra features such as *dynamic structural summaries* (*DataGuides*), *keyword and proximity search*, and an *external data manager*. Our schema-less, self-describing data model, called the *Object Exchange Model* (*OEM*), was essentially a directed labeled graph—or equivalently, nested tagged data with references. Our query language, *Lorel*, was based on OQL, with modifications and extensions suitable for semistructured data. Many aspects of building a DBMS top-to-bottom needed to be revisited in the context of semistructured data. We published papers, distributed our system, and continued to work on a variety of issues.

When XML came along, the similarity between OEM and XML was striking, and exciting. In late 1998 and early 1999 we migrated Lore to be based on a true XML-oriented data model and modified our query language accordingly. (See our short paper [6] on the topic, which also happens to be one of the publication examples in Section 1.) A number of subtleties were involved in both design and implementation, and the first public release of the XML version of Lore was made in May 1999.

While some aspects of Lore could still benefit from refitting and tuning for XML, by and large we have one of the only complete database systems designed specifically for storing and querying "native" XML. We believe that Lore provides an ideal testbed for further research in this area.

5 Personal Research Agenda

As outlined in Section 3, there is a broad range of research issues to be explored in XML data management. Here I list a number of more specific topics that are on my personal research agenda. In some cases the topics are obvious follow-ons to previous work in Lore, in other cases the topics are just plain interesting.

5.1 Storage and Indexing

Lore uses a simple storage manager that parses XML into the basic units of elements, attributes, and text strings, and stores them using a straightforward depth-first clustering heuristic. We can build a wide variety of indexes on Lore databases. The types of indexes we support were designed for Lore's original data model, OEM, but with minor changes the indexes also work for XML. There are several further avenues to pursue in storage and indexing:

- Different clustering schemes for storing XML data, preferably customizable for different databases and applications.
- New index types designed specifically for XML data, taking into account element ordering, new kinds of comparison operations, and a few other subtle differences between XML and our original data model.
- More radically, we plan to explore using XML documents themselves as a storage medium in Lore, augmented with auxiliary structures such as: (1) an indexing scheme for quickly finding certain elements, attributes, and more complex structural patterns in the document; and/or (2) a data store containing some of the XML data parsed to some level (and possibly created "lazily", i.e., on an as-needed basis), with a mapping maintained between the database and documents.
- Depending on how applications tend to encode their data in XML, we may find that we can take data that is primarily encoded as tree structures and convert it to a more graph-structured representation, e.g., by merging identical text values or entire subelements, then inserting appropriate IDREFs. It will be interesting to see whether any significant compression is achieved, but more importantly to observe the effect on query processing and even query semantics.
- Another opportunity for compressing XML data is to exploit regularity in the structure, in the extreme case storing the XML data essentially as relations. (See also Sections 5.5 and 5.6 below.)

5.2 DataGuides and DTDs

Lore builds and dynamically maintains a *DataGuide* for every database, which is a summary of the current structure of the database and serves some of the functions a schema serves in a traditional DBMS. Since an XML DTD (Document Type Definition) is a set of grammar rules that restrict the form of an XML document, there is a close relationship between DataGuides and DTDs. Note that a DTD acts more as a traditional schema, since it restricts the allowable XML data, while a DataGuide infers rather than imposes structure.

Currently we can store DTDs in Lore databases, and we can use a DTD to build an "approximate" DataGuide. There can be a significant performance advantage to building DataGuides from DTDs instead of from the database itself: in some cases, particularly in highly connected and cyclic databases, building a DataGuide can be prohibitively expensive. However, the DTD does not capture the structure of a database as accurately as a DataGuide: attributes and elements included in a DTD need not actually appear in the database, and DTDs cannot specify restrictions on the types of elements referenced by IDREF attributes. Furthermore, the lack of accuracy in DTDs inhibits other ways we use DataGuides in Lore, such as storing statistics and encoding path indexes.

There are several avenues to pursue in the general area of DTDs, DataGuides, and their relationship, listed here in no particular order:

- Validating parsers can check that an XML document conforms to its DTD, and we can build the same functionality into an XML database system. However, there are some interesting related problems: Can we perform validation incrementally as portions of an XML database are updated? Can we perform validation on the update statements themselves, instead of on the database? When we pose a query to an XML database, can we infer a DTD for the query result?
- Currently we do not encode subelement ordering in our DataGuides (even the DataGuides we build from DTDs). If subelements of a given element type always appear in the same order, it would be nice to reflect that ordering in the DataGuide. However, if subelements do not always appear in the same order, we probably do not want to expand the size of the DataGuide to encode that fact. One possibility is to order subelements in the DataGuide based on a "most frequent" ordering from the database.
- We would like to further investigate the performance and functionality tradeoffs between using (exact) DataGuides inferred from the database versus (possibly inexact) DTDs.

- There may be scenarios where DTDs are available for specific portions of an XML database, but not for all of the data. In that case, for maximum flexibility we might want to build a DataGuide for the portion without a DTD, and link to DTDs in the appropriate places.
- Our DataGuide serves as the basis of a convenient interface by which users can browse the current structure of the database and even formulate queries "by example." Independent of XML, for some time we've had the idea of trying to blur the distinction between browsing structure (the DataGuide) and data (the database). We envision interactions where predicates are specified within the DataGuide, which produces a smaller DataGuide based on the constrained database, through which additional predicates may be specified, and so on. When the constrained database becomes sufficiently small, we should automatically begin browsing the data along with the structure.
- Along similar lines, when browsing the structure of a database through the DataGuide, we might want to specify certain updates that should propagate to the database. For example, we might "cleanse" the data by modifying or merging tags, or by identifying portions of the database that are uninteresting or erroneous and can be deleted.

5.3 Databases and Information Retrieval

The typical paradigm for querying document collections is to use information retrieval style searches based on keyword matching and word position within documents. By contrast, the typical paradigm for querying databases is through an expressive, declarative query language (such as SQL) that relies on database structure.

Lore implements the declarative OQL-based query language *Lorel*, along with a *keyword and proximity search* feature. Keyword search in Lore is straightforward: we find all objects (elements or attributes) whose tag, name, or value contains the specified keyword. Proximity search is more interesting. In a document collection, a typical "*X near Y*" search might return all documents containing both *X* and *Y*, ranked by how near *X* and *Y* are to each other within the document. There are two problems with this approach:

1. If the documents are structured, as XML documents are, then nearness based on the linear encoding of the document, rather than on the document's structure, may not work well. For example, in our sample XML data of Section 1, the year of a publication is closer in a document sense to the title of the following publication in the list than it is to its own title, and author names are nowhere near the relevant references.

2. Document divisions may be artificial: items that are near each other semantically may not even appear in the same document, and the concept of separate documents may be lost when loading XML into a database.

In Lore we solve both of these problems by encoding all XML structure in the database, and by measuring proximity based on (weighted) path length in the database graph.

We're excited about how the techniques we've developed for proximity search in Lore might be used to improve searching over XML on the Web. Still, there is more work to be done on proximity search in Lore, as well as generally integrating database and information retrieval concepts:

- While we've developed some novel indexing structures and algorithms designed to make Lore's proximity search feature scale to very large databases, scaling to Web proportions is still a significant leap away.
- In Lore, keyword/proximity searches versus Lorel queries so far have been completely separate. We would like to integrate searching into Lorel queries, which (among issues) requires us to combine standard set-based query results with ranked results. A related idea is to use proximity search "under the covers" as a pre-filtering step during query processing.
- Our experience so far has been that Lore's proximity search feature yields intuitive and useful results, and that keyword and proximity search in general are a good way for casual users to interact with Lore databases. One feature that's missing, however, is the ability to explain why objects rank where they do in proximity search

results. Identifying and saving the relevant "near" objects is both a technical issue during search processing (primarily an efficiency problem), as well as a user-interface issue of presenting the explanations in an intuitive fashion.

- We have begun investigating the related problem of *similarity search* in Lore. While proximity search is useful for identifying objects that are related based on closeness in a graph sense (due to shared subobjects, for example), it doesn't identify objects that are similar structurally but distant within the database structure. In similarity search, given a set of XML objects (elements and/or attributes), we want to rank them based on how similar they are to one or more other objects. Similarity of objects might be based on values, substructure (children, outgoing IDREFs), and/or super-structure (parents, incoming IDREFs). Needless to say, coming up with a workable definition of similarity measure, much less developing efficient evaluation algorithms, is a significant challenge. In addition, the issues of scalability, integration with the Lorel query language, and providing an "explain" feature to the user, described in the previous three bullets for proximity search, are relevant here as well.

5.4 Other Database Features

Three useful features provided by most traditional database management systems are *views*, *constraints*, and *triggers*. We have done some initial work on materialized views in Lore, and we also have done some work on *change management*—tracking, representing, and querying changes in semistructured databases. However, full view support for XML, including both virtual and materialized views, is a largely unexplored area. The same is true of constraints; in fact, even the notion of keys has not been introduced into XML or semistructured database work in a significant way, as far as I know. With respect to triggers (or active database capabilities in the Web/XML context in general), it is important to consider the relationship with recent developments in publish-subscribe technology.

5.5 Mixing Semistructured and Structured Data

A frequent and valid criticism of work in the area of semistructured data management is that *all* data is assumed to be semistructured in nature, i.e., (as defined earlier) the data may be irregular or incomplete, and its structure may change rapidly and unpredictably. As a result, when structure does happen to be present and stable in a semistructured database, the structure is not exploited to the advantage of the user or the system. Lore is certainly guilty on this count.

There are two issues to address: (1) finding the structure; (2) exploiting the structure. There has been some work on issue (1), although with XML the presence of a DTD certainly alleviates this problem to a great extent. The second issue—how to exploit structure when it is present, but not require structure for effective or efficient processing—is largely yet to be addressed. Structure might be exploited at all levels of the system: object layout, indexing, query processing, query formulation, etc. We have done some preliminary work in this general area by "marrying" the ODMG and OEM data models (and corresponding query languages) in a system called *Ozone*, but clearly there is more to be explored.

5.6 XML in/on a Traditional DBMS

We decided to build Lore from scratch so that we could explore every nook and cranny of building a complete DBMS for semistructured data (now XML). In retrospect, we probably should have used somebody else's low-level page or object storage manager, since we haven't done much work at that level, and writing a transaction manager was a pain. More generally, it would be interesting to explore just how much of an XML-savvy DBMS actually needs to be built from scratch, and how much can be lifted from existing DBMSs. (For example, at the other extreme, we and others have considered schemes for translating semistructured XML data to relations and translating Lorel queries to SQL accordingly.) While I expect there are some interesting research problems to explore all along the spectrum, it's also a certainty that database companies are working fast and furious to figure out how XML data and queries can fit into their systems.

5.7 Performance Evaluation

We have performed some preliminary evaluations of Lore's performance, particularly in the context of query optimization. However, we have had great difficulty figuring out what an appropriate benchmark for XML data should look like, in terms of the data itself (e.g., regular versus irregular, tree versus DAG versus general graph) as well as the type of queries and mix of queries and updates. So far we have not found any large, semistructured XML data sets to work with other than those we have constructed ourselves, although this dearth of interesting XML data surely will change. Meanwhile, flexible synthetic data generation for graph-structured data is a hard and interesting problem.

6 More Information and Related Work

Information related to many of the topics discussed here, ranging from how to download Lore to research papers to links to commercial product sites, can be found by starting at the Lore project's home page [5]. There's also a lot of good stuff to be found from the W3C's QL '98 Web site [7]. I have not attempted to cover the very interesting and fast-growing body of great work from other researchers in XML and databases—I hope I haven't stepped on anyone's toes.

Alon Levy has created an interesting follow-up document to this one, *More on Data Management for XML* [8], covering two important topics omitted here: the role of XML in data integration, and new measures of complexity and scalability for XML query processing.

Acknowledgements

Many people have had a direct influence on my thinking and work on XML, including but not limited to (in alphabetical order): Serge Abiteboul, Adam Bosworth, Roy Goldman, Jason McHugh, Michael Rys, and Frank Tompa. For comments on this paper, thanks go to Roy Goldman, Alon Levy, Jason McHugh, and Dan Suciu.

Funding for my research in semistructured data has come in the past from DARPA and the Air Force Rome Laboratories, and currently is supported by NASA and by the National Science Foundation under grant IIS-9811947.

References

[1] http://www-db.stanford.edu/˜widom/xml-whitepaper.html
[2] http://www.w3.org/XML
[3] http://www.w3.org/TR/WD-xsl
[4] http://www-db.stanford.edu/˜widom/pubs.html
[5] http://www-db.stanford.edu/lore
[6] ftp://db.stanford.edu/pub/papers/xml.ps
[7] http://www.w3.org/TandS/QL/QL98/
[8] http://www.cs.washington.edu/homes/alon/widom-response.html

25th International Conference on Very Large Databases

Edinburgh - Scotland - UK

7th - 10th September 1999

Invitation to attend VLDB'99

VLDB'99, Edinburgh, UK marks the 25th anniversary of the leading international conference on databases. VLDB'87 was also held in UK (at Brighton) and was one of the largest VLDB conferences and produced some of the most cited VLDB papers. We expect VLDB'99 to be no less successful. For database technology practitioners and theorists VLDB'99 is not to be missed.

The conference has an excellent programme of Tutorials, Panels and Invited Talks, and the selection of papers to be presented from the many submitted will ensure - as usual for VLDB - the highest quality. A programme of demonstrations ensures awareness of the leading edge R&D and the exhibition provides an opportunity to see and discuss products. Furthermore, there are workshops and events organised before and after the conference, so making the travel to attend VLDB'99 even more worthwhile.

VLDB'99 is being held in Edinburgh, a city world-renowned for its beauty, style, culture and nightlife. VLDB'99 is being held just after the Edinburgh Festival - a truly amazing collection of art, drama, music, revue and cinema. Edinburgh can be reached easily - see the VLDB99 website for details. VLDB'99 attendees will find themselves not only at an interesting spatial location; the temporal location is also important as Scotland gains some level of independence from the rest of the UK through devolution. This is the time to visit Edinburgh, Scotland's capital city.

The VLDB Social programme has also been carefully arranged to provide attendees not only with a taste of Edinburgh and Scotland, but also those opportunities for discussion and conversation so necessary to the advancement of database technology and to the maintenance and further building of the community.

So, on behalf of the Programme Committee led by Malcolm Atkinson and the Organising Committee led by Jessie Kennedy it is my pleasure to invite you to attend VLDB'99. Registration details, and details of travel, accommodation, programme and social programme are to be found in this Call for Attendance or on the VLDB'99 Website http://www.dcs.napier.ac.uk/~vldb99/

I look forward to meeting you in Edinburgh

Keith G Jeffery
General Conference Chair VLDB'99

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398