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 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. Similarily, `$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
```

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.

	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.

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] pS. 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.