SQL/XML, XQuery, and Native XML Programming Languages

Jonathan **Robie**

Abstract

The abstract was not available at the time the proceedings were created. Please check an updated version [http://www.idealliance.org/papers/dx_xml03/html/abstract/05-02-01.html] of the paper abstracts at the conference proceedings web site.

Table of Contents

1. Introduction 1
2. XML and Relational - Opposites Attract 2
3. XML and Relational: Four Approaches 4
4. SQL/XML 4
    4.1. XML Publishing Functions 4
    4.2. The XML Datatype 7
    4.3. SQL/XML Mapping Rules 8
5. XQuery and Native XML Programming 10
    5.1. Native XML Programming 10
    5.2. XQuery and SQL/XML Views 14
    5.3. Spanning Sources: XQuery, Web Messages, and Databases 16
    5.4. XQuery for Java (JSR 225) 17
6. SQL/XML and XQuery: Do we need both? 17
Bibliography 18

1. Introduction

Most web applications have connections to databases and use XML to transfer data from the database to the web application and vice versa. Every major database vendor has proprietary extensions for using XML with relational databases, but they take completely different approaches, and there is no interoperability between them. Many developers need to be able to write applications that work for databases from multiple vendors.

XQuery[XQuery] and SQL/XML[SQL/XML] are two standards that use declarative, portable queries to return XML by querying data. In both standards, the XML can have any desired structure, and the queries can be arbitrarily complex. XQuery is XML-centric, while SQL/XML is SQL-centric.

SQL/XML is an extension of SQL that is part of ANSI/ISO SQL 2003. It lets SQL queries create XML structures with a few powerful XML publishing functions. For a SQL programmer, SQL/XML easy to learn because it involves only a few small additions to the existing SQL language. Since SQL is a mature language, there are a lot of tools and infrastructure for SQL. For instance, SQL/XML uses JDBC to return results, and there is currently no equivalent standard API for XQuery. SQL also has functionality not yet found in XQuery, such as updates or stored procedures.

Note

SQL/XML is completely different from Microsoft's SQLXML, a proprietary technology used in SQL Server. The similarity in names has caused a great deal of confusion in the industry.

XQuery is a completely new query language that uses XML as the basis for its data model[DataModel] and type system [XQuery] [XQuerySemantics]. It is being developed in the XML Query Working Group[XQWG], which is a part of the World Wide Web Consortium. In this paper, we characterize XQuery as a "Native XML Programming Language". XQuery is based on XML in the same way that SQL is based on the relational model or object oriented languages are based on the object oriented model - XML is central to its type system, in which elements and attributes are just as fundamental as integers and strings. Although XQuery per se has no concept of relational data, several products and many projects provide ways to query relational data using an XML view of the database, and the need to make this possible has influenced the design of XQuery throughout its development. XQuery allows you to work in the XML world no matter what type of data you're working with - relational, XML or object data. XQuery is ideal for native XML programming. When used with XML views of relational data, it is also ideal for queries data that must represent results as XML, to query XML stored inside or outside the database, or to span relational and XML sources.

For queries based only on relational data, SQL/XML and XQuery have substantially similar functionality. However, the way in which a given task is done is quite different, since SQL/XML operates on the borderline between SQL and XML, and XQuery lives in a purely XML world. Even when the data is all relational, the two languages appeal to very different audiences - SQL/XML is very much an extension of SQL, designed for SQL programmers, and XQuery takes a purely XML view of the world. For queries that span relational and XML sources, XQuery has important advantages.

This talk uses a series of concrete queries written in each language to show the advantages of each. It explains why we need both languages, discussing the ways in which the languages differ and in which they overlap. It also explores the role of SQL/XML mappings as a way of creating XML views for XQuery.

2. XML and Relational - Opposites Attract

XML and relational databases are tightly wed in most web applications, but a look at the two models shows that it is an unlikely marriage - though a necessary one. The relational model is based on two dimensional tables which have neither hierarchy nor significant order. XML is based on trees in which order is significant. In the relational model, neither hierarchy nor sequence may be used to model information; in XML, hierarchy and sequence are the main ways to represent information. Although this is one of the more fundamental differences between the two models, it is by no means the only one.

In many environments, the same information is represented in relational databases when it is stored or queried, but in XML when it is exchanged or displayed on web pages. These representations are often completely different due to the differences in the models.

On web pages, XML is useful because the structure of XML closely matches the structure used to display the same information in HTML. If you look at web pages, they often use a distinctly hierarchical structure to present data for users - after all, users don't want to look at a bunch of tables and do joins in their head. But most of the data for these web pages comes from relational databases, and needs to be converted to appropriate XML hierarchies.

For web messages, the format of a web message is often specified by a standards organization or a trade partner, and these formats are generally hierarchical. Again, the data for a web message generally comes from relational data, and the consumer of a web message often needs to put data into a relational database.

For instance, suppose a consulting company needs to represent a set of projects and the companies for whom the projects are being done. In a relational database, this might be represented by the following tables:

Projects

ProjId	Name	CustId
1	Medusa	1
2	Pegasus	4
8	Typhon	4
10	Sphinx	5

```
        Customers

CustId      Name                   City
----------- ---------------------- --------------
1           Woodworks              Baltimore
2           Software Solutions     Boston
3           Food Supplies          New York
4           Hardware Shop          Washington
5           Books Inc.             New Orleans
```

In SQL, if we want to see the projects associated with each customer, we would do the following query:

```
select *
from Customers c, Projects p
where c.CustId = p.CustId
order by c.CustId, p.ProjId
```

Here is the output of the above query:

```
CustId      CustName           City             ProjId      ProjName
----------- ------------------ ---------------- ----------- ------------
1           Woodworks          Baltimore        1           Medusa
4           Hardware Shop      Washington       2           Pegasus
4           Hardware Shop      Washington       8           Typhon
5           Books Inc.         New Orleans      10          Sphinx
```

Suppose we want to translate this information into XML for use on a web page, in a document, or in a web message. Like most XML applications, we will leverage the hierarchy of XML to express relationships, listing the projects for each customer within the element that represents the customer:

```
<?xml version="1.0" encoding="UTF-8"?>
<customers>
    <customer id="1">
        <name>Woodworks</name>
        <city>Baltimore</city>
            <projects>
                <project id="1"><name>Medusa</name></project>
           </projects>
    </customer>
    <customer id="4">
       <name>Hardware Shop</name>
       <city>Washington</city>
           <projects>
               <project id="2"><name>Pegasus</name></project>
               <project id="8"><name>Typhon</name></project>
           </projects>
    </customer>
    <!-- !!! SNIP !!! -->
</customers>
```

Note that in the original SQL tables, each customer is represented only once. This is also true of the XML. The SQL result set, however, contains multiple rows for a given customer if that customer is associated with more than one project, and these rows contain duplicate information. Translating this result set into the desired XML is tedious for the programmer. And just as a single relational database may be used with an infinite number of queries, it

may also be used to create an infinite number of XML documents with different structures. Today, many programmers spend a great deal of time doing this kind of translation.

3. XML and Relational: Four Approaches

XML applications that use relational data can choose from four approaches, each with distinct advantages and disadvantages. The first three of these are compared in some detail, with code samples, in [SQL/XML-JDBC].

The programmer can use JDBC or ODBC together with SAX or DOM and perhaps XSLT to transform the results of SQL queries to XML. For instance, the program might first query for customers, then perform an additional query to find the projects associated with each customer. This is inefficient because of the number of queries required. Another approach would be to use SQL to create a table that lists customers and their projects, and pick through the rows to determine when a row represents a new customer. This requires more code, but is more efficient. Both of these approaches require significant amounts of tedious code, but they are often used when database independence is important.

The programmer can use the XML extensions provided by the major database vendors. These are based on several different approaches. Some of these are simpler to use or maintainable than others, but they all make the task easier. However, since these extensions are all proprietary, they are not an option when a database-independent solution is needed.

The programmer can use SQL/XML, which is part of SQL 2003. For a SQL programmer, this approach requires little new learning - a small set of XML publishing functions have been added to SQL to allow queries to create any desired XML structure. This approach will be explored with examples in the next section. SQL/XML is being supported by Oracle and IBM, but not by Microsoft. Database-independent implementations of SQL/XML are also available, and can be used with any major relational database. SQL/XML can be used with traditional database APIs such as JDBC.

The programmer can use XQuery, a native XML query language. Since XQuery is a new language, it requires more learning for SQL programmers, but it is likely to be more natural for XML programmers. Unlike SQL/XML, XQuery is optimal for processing XML, and it is also particularly good for applications that must process XML together with relational data, with full support for XML. Most of the major database vendors intend to support XQuery. The first standardized API for XQuery, XQuery for Java (JSR 225), is now being developed under Java Community Process, and is expected to be available shortly after the XQuery Recommendation is released.

4. SQL/XML

SQL/XML refers to the XML extensions of SQL. These are developed by INCITS H2.3, with participation from Oracle, IBM, Microsoft (which does not plan to implement SQL/XML), Sybase, and DataDirect Technologies. In SQL 2003, these extensions include:

- XML Publishing Functions
- The XML Datatype
- Mapping Rules

The XML Publishing Functions are the part that are directly used in a SQL query. The XML Datatype governs the result of a query, and the Mapping Rules determine how SQL data or metadata is represented as XML.

4.1. XML Publishing Functions

The XML Publishing Functions allow SQL to create any desired XML structure. They are part of SQL 2003, and can be used in normal SQL expressions. Here are the XML publishing functions of SQL 2003:

xmlelement() Creates an XML element, allowing the name to be specified.

xmlattributes()	Creates XML attributes from columns, using the name of each column as the name of the corresponding attribute.
xmlroot()	Creates the root node of an XML document.
xmlcomment()	Creates an XML comment.
xmlpi()	Creates an XML processing instruction.
xmlparse()	Parses a string as XML and returns the resulting XML structure.
xmlforest()	Creates XML elements from columns, using the name of each column as the name of the corresponding element.
xmlconcat()	Combines a list of individual XML values to create a single value containing an XML forest.
xmlagg()	Combines a collection of rows, each containing a single XML value, to create a single value containing an XML forest.

Let's compare a traditional SQL query with one that uses an XML publishing function. Here is a traditional SQL query that shows customers and their associated projects:

```
select c.CustId, c.Name as CustName
from customers c
```

Here is is an excerpt of the result:

```

CustId  CustName
------- ------------------
1       Woodworks
4       Hardware Shop
6       Photo Shop
8       Computer Supplies
```

Now let's wrap the result in XML elements using xmlelement(), one of the publishing functions:

```
select xmlelement(name "Customer",
      xmlelement(name "CustId", c.CustId),
      xmlelement(name "CustName", c.Name)
      xmlelement(name "City", c.City))
from Customers  c
```

Each row in the result contains one Customer element. A Customer element looks like this:

```
<Customer>
  <CustId>1</CustId>
  <CustName>Woodworks</CustName>
  <City>Baltimore</City>
</Customer>
```

xmlforest() is an XML publishing function that creates elements from a list of columns, using the name of the column as the name of the element. Using xmlforest() simplifies many queries significantly. For instance, the following query is equivalent to the previous one:

```
select xmlelement(name "Customer",
      xmlforest(c.CustId, c.Name as CustName, c.City))
from Customers  c
```

Now suppose we want to show customers and the projects associated with them. This is easily done with the following SQL query:

```
select *
from Customers c, Projects p
where c.CustId = p.CustId
order by c.CustId, p.ProjId
```

However, the result of this query is that shown in the CustomerProject table in the previous section, with one row for each Customer/Project pair. If a customer is associated with more than one project, there will be a row for that customer for each project. Here is a SQL/XML query that creates the XML equivalent to that table:

```
select xmlelement(name "CustomerProj",
   xmlforest(c.CustId, c.Name as CustName, p.ProjId, p.Name as ProjName))
from Customers c, Projects p
where p.CustId=c.CustId
order by c.CustId
```

Here are the results of this query:

```
<CustomerProj>
  <CustId>1</CustId>
  <CustName>Woodworks</CustName>
  <ProjId>1</ProjId>
  <ProjName>Medusa</ProjName>
</CustomerProj>
<CustomerProj>
  <CustId>4</CustId>
  <CustName>Hardware Shop</CustName>
  <ProjId>2</ProjId>
  <ProjName>Pegasus</ProjName>
</CustomerProj>
<CustomerProj>
  <CustId>4</CustId>
  <CustName>Hardware Shop</CustName>
  <ProjId>8</ProjId>
  <ProjName>Typhon</ProjName>
</CustomerProj>
```

This is a straightforward XML translation of the that a SQL result set shown in the previous section, but for most XML applications it is not what we would want. Instead, we want to represent each customer once, with a list of that customer's projects, as shown in the XML output in the previous section. In SQL/XML, this can be done by using a sub-query. Here is a subquery that retrieves the projects associated with each customer. In this subquery we use xmlattributes(), an XML publishing function that creates attributes within an element. The names of the attributes are taken from the names of the columns.

```
(select xmlelement(name project,
       xmlattributes(p.ProjId as id),
       xmlforest(p.Name as name))
from Projects p
where p.CustId=c.CustId)
```

Here is the output of the above sub-query when c.CustId is 4:

```
<project id='2'>
  <name>Pegasus</name>
</project>
<project id='8'>
  <name>Typhon</name>
</project>
```

This output contains two rows, with one element in each row. Subqueries in SQL/XML are allowed to return only one row; therefore, to return more than one row of values in a SQL/XML subquery, they must be combined to form a single value. xmlagg() is an XML publishing function that produces a forest of elements by collecting the XML values that are returned from multiple rows and concatenating the values to make one value. Here is a query that uses the above subquery to create the XML output from the previous section:

```
select
  xmlelement(name customer,
    xmlattributes(c.CustId as id),
    xmlforest(c.Name as name, c.City as city),
    xmlelement(name projects,
    (select xmlagg(xmlelement(name project,
       xmlattributes(p.ProjId as id),
       xmlforest(p.Name as name)))
      from Projects p
     where p.CustId=c.CustId))) as "customer-projects"
from Customers c
```

The above query illustrates a very common pattern used to create XML hierarchies using SQL/XML.

4.2. The XML Datatype

The XML Datatype is a datatype in the same way that integer, date, or CLOB are datatypes in SQL. Since SQL/XML allows a query to create XML instances, there must be a datatype that corresponds to these instances.

It is anticipated that the XML Datatype will be supported in JDBC 4.0. It is too early to say exactly how it will be used in that specification, but it is likely that it will retrieve XML values much like other values, and that XML values can be retrieved as text, DOM, or SAX events. This is the approach currently taken by DataDirect Connect for SQL/XML. To illustrate this, let's use a SQL/XML query to create a table with two columns, an integer containing the CustId and an XML column containing the XML output from the previous query. Here is the query:

```
select c.CustId,
  xmlelement(name customer,
    xmlattributes(c.CustId as id),
    xmlforest(c.Name as name, c.City as city),
    xmlelement(name projects,
    (select xmlagg(xmlelement(name project,
       xmlattributes(p.ProjId as id),
       xmlforest(p.Name as name)))
      from Projects p
     where p.CustId=c.CustId))) as "customer-projects"
from Customers c
```

Suppose the above query is in a string called sqlxmlString. Then the following Java code can be used to execute the query and retrieve values.

```
Statement statement=con.createStatement();

ResultSet rs=statement.executeQuery(sqlxmlString);

while(rs.next())
  {
   int id=rs.getInt(1);
   com.ddtek.jdbc.jxtr.XMLType xmlC=
     (com.ddtek.jdbc.jxtr.XMLType)rs.getObject(2);
   org.w3c.dom.Document doc=xmlC.getDOM();
   doSomethingUseful(id, doc);
  }
```

The XML Type also plays a second important role - relational databases now routinely store XML in individual column, and the XML Type provides a standard type for such columns, which is useful both in SQL and in JDBC.

4.3. SQL/XML Mapping Rules

The XML publishing functions use SQL values to create XML values, and these XML values have W3C XML Schema types. When we discussed the XML publishing functions, we did not address specifically how the XML representation is determined. The mapping rules of SQL/XML describe in excruciating detail how SQL values can be mapped to and from XML values, and how SQL metadata can be mapped to and from W3C XML Schemas. To give a flavor for the level of detail in which this is specified, here are the equivalent headings from the SQL/XML specification's table of contents:

- Mapping SQL character sets to Unicode.
- Mapping SQL <identifier>s to XML Names.
- Mapping SQL data types (as used in SQL-schemas to define SQL-schema objects such as columns) to XML Schema data types.
- Mapping values of SQL data types to values of XML Schema data types.
- Mapping an SQL table to an XML document and an XML Schema document.
- Mapping an SQL schema to an XML document and an XML Schema document.
- Mapping an SQL catalog to an XML document and an XML Schema document.
- Mapping Unicode to SQL character sets.
- Mapping XML Names to SQL <identifier>s.

These mappings can be parameterized in several ways, including the target namespace for the result, whether to handle nulls using xsi:nil or absence, and whether to map a table to a single element or a forest of elements. Here is an XML representation of the Customers table shown earlier, using a single element for each table and no target namespace:

```
<myschema>

 <Customers>
  <row>
    <CustId>1</CustId>
    <Name>Woodworks</Name>
    <Address>Baltimore</Address>
  </row>
  <row>
```

```
    <CustId>2</CustId>
    <Name>Software Solutions</Name>
    <Address>Boston</Address>
  </row>
  <!-- !!! SNIP !!! -->
 </Customers>
<myschema>
```

Here is an XML representation of the same table using a forest of elements to represent each table:

```
<myschema>

 <Customers>
    <CustId>1</CustId>
    <Name>Woodworks</Name>
    <Address>Baltimore</Address>
 </Customers>
 <Customers>
    <CustId>2</CustId>
    <Name>Software Solutions</Name>
    <Address>Boston</Address>
 </Customers>
  ...
<myschema>
```

These mappings are also defined on the metadata level. For instance, SQL/XML defines how the datatypes of SQL are represented in the equivalent XML Schema. Each SQL type is derived from an equivalent built-in W3C XML Schema type. Where needed, facets are used to represent constraints added to those of the base type:

```

    <xsd:simpleType name="INTEGER">
         <xsd:restriction base="xsd:int" />
    </xsd:simpleType>

    <xsd:simpleType name="CHAR_50">
        <xsd:restriction base="xsd:string">
            <xsd:length value="50"/>
        </xsd:restriction>
    </xsd:simpleType>
```

As mentioned above, there are two ways to represent null values. Suppose the City column may have null values. Here is a row in the Customer's table that represents a null value using the first strategy, a nilled element:

```
 <Customers>
  <row>
    <CustId>1</CustId>
    <Name>Woodworks</Name>
    <Address xsi:nil="true" />
  </row>
  <!-- !!! SNIP !!! -->
```

Here is a row that uses the second strategy, an absent element:

```
 <Customers>
  <row>
    <CustId>1</CustId>
```

```
    <Name>Woodworks</Name>
  </row>
  <!-- !!! SNIP !!! -->
```

5. XQuery and Native XML Programming

The XQuery language was designed for querying or processing XML. Just as a traditional SQL query takes a set of tables as input and returns an XML table as its result, XQuery takes sequences of XML nodes as input and evaluates to a sequence of XML nodes. However, from the very beginning, XQuery was designed to allow XML views of non-XML data, as well as serialized forms of non-XML data. The reason for this is simple: XML is used to represent almost any conceivable kind of information, and it is easiest to integrate information if it is given a common view.

If everything looks like a nail, all you need is a hammer. Conventional Internet applications often store and query data using SQL, process data using Java or C#, and exchange data as XML. Using XQuery, it is possible to store, query, process, and exchange data as XML. This eliminates some of the mismatches that cause complications when working with XML in other environments.

5.1. Native XML Programming

XQuery is a language designed for integrating data from multiple sources, including XML sources like documents or web messages and databases. It does this by leveraging the ability of XML to model virtually any kind of data. To query anything with XQuery, it must be presented as though it were XML, either by serializing it as XML or by creating an XML view of the data through some form of middleware. For relational data, most systems use the SQL/XML mappings for the XML view, since they are quite suitable and have been specified in detail.

XML is the basis of XQuery's type system and data model. The fundamental types of XQuery include the kinds of nodes found in XML documents: document nodes, elements, attributes, processing instructions, comments, and text nodes. XQuery also supports the built-in datatypes of W3C XML Schema for representing integers, strings, dates, and other datatypes - these built-in datatypes are predefined in XQuery, and are available with or without a schema.

Most modern programming languages provide some form of complex user-defined types, such as structures or objects. In XQuery, the only complex types are XML documents, elements, attributes, and W3C XML Schema complex types. There is no need to write a schema to create and manipulate complex XML structures in XQuery. However, if a query needs to ensure consistent use of the types in a schema, a schema may be imported into a query. This has an effect analogous to importing structure or class definitions in an object oriented language.

Programs tend to revolve around data, and the complex datatypes used in a language have a profound effect on the way that a language is used. As a result, languages are sometimes identified by the way they represent complex data; for instance, there are object-oriented languages and relational query languages. In this sense, XQuery can be considered Native XML Programming Language. XSLT and XPath are also Native XML Programming Languages. Most other languages used to process XML, including Java, C#, Perl, and Python are not. SQL/XML is fundamentally an extension to a relational query language, providing a bridge to XML.

The concept of a Native XML Programming Language is new, and many XML programmers are used to thinking of XML in terms of the constructs used in the languages with which they process XML. On XML-related mailing lists it is reasonably common to see beginners assert that XML is fundamentally relational or object-oriented, and even sophisticated XML programmers have been known to assert that XML is just text. In fact, the phrase "XML is Unicode with pointy brackets" has come to identify a vocal part of the XML community.

5.1.1. XML is not Objects!

An XML document can be represented using objects, and this is precisely the approach taken by DOM and JDOM. An XML parser can be used to create an appropriate object representation of an XML document without involving the programmer. However, the fundamental types of XML are not fundamental in object oriented languages, so casting and conversion is frequently required. Similarly, the basic notions of hierarchy and containment are not

directly supported in the object oriented model, so explicit navigation is often required. This causes significant work for the programmer.

Adam Bosworth pointed this out with the following example [Bosworth]. Suppose a programmer wants to compute price/earnings ratios from an XML feed. An individual stock might be represented as follows:

```
<stock>
  <name>Cindy's Snowshoes</name>
  <ticker>NASDAQ:RAKD</ticker>
  <price>20.00</price>
  <revenues>2.00</revenues>
  <expenses>1.00</expenses>
</stock>
```

To compute the price/earnings ratio, we use the formula "pe = price / (revenues - expenses)". To do this with the DOM, we also need to parse the XML, navigate to the places where this information is found, and convert the text of the document to the appropriate datatype. Here is the DOM code Adam provides for this:

```
Tree t = ParseXML("stock.xml");
PERatio = number(t.getmember("/stock/price"))
     /((number(t.getmember("/stock/revenues") -
      number(t.getmember("/stock/expenses"))
```

This solution would have been much messier if Adam had not used the path expressions of XPath, a simple Native XML language. In XQuery, path expressions are part of the language, and numeric conversions are automatically done for untyped data. If the data is validated against a schema, the types assigned by the schema are used. This makes it possible to solve the same problem much more simply:

```
let $stock := document('stock.xml')/stock
return $stock/price div ($stock/revenue - $stock/expenses)
```

For XML-centric applications, an object-oriented representation of an XML document imposes unneeded overhead that complicates programs.

5.1.2. XML is not just text!

To many intelligent and articulate XML programmers, "XML is just Unicode with pointy brackets" is almost a statement of faith. Predictably, these people also complain that it is difficult to process XML without a parser. For instance, Joe Gregorio [Gregorio1] notes that in XML this document:

```
<item xmlns:dc="http://purl.org/dc/elements/1.1/">
  <title>MetaData</title>
  <dc:date>2003-01-12T00:18:05-05:00</bc:date>

  <link>http://bitworking.org/news/8</link>

  <description>Upon waking, the dinosaur...</description>
</item>
```

must be treated as identical to this document:

```
<root:item xmlns:bc="http://purl.org/dc/elements/1.1/" xmlns:root="" >
  <root:title>MetaData</root:title>

  <bc:date>2003-01-12T00:18:05-05:00</bc:date>
```

```
  <root:link>http://bitworking.org/news/8</root:link>
  <description>Upon waking, the dinosaur...</description>
</root:item>
```

To many of us, this is merely an indication that XML must first be parsed and converted to an appropriate data model before it can easily be processed. In fairness to Joe, he initially assumed this as well, but then changed his mind:

> More XML experience is gained by yours truly and on many occasions I have found myself pining for the ability to do regular expression processing of XML. If only the pathologies of the above examples didn't exist then I could use a combination of XPath and regular expressions to perform XML manipulations that would be easier for me to implement, understand and maintain.
>
> Today I reached the breaking point. The problem isn't with regular expressions, the problem is with XML. The pathologies in XML that preclude the use of regular expressions are just that, pathologies, and ones that need to be excised.

As a result, he suggests that XML be subsetted as follows:

1. All namespace declarations must be done in the root element.
2. Never a declaration for the "" namespace. I.e. if an element sits the "" namespace then the element name will never have a namespace qualifier.
3. No CDATA sections.
4. No DTDs.

The above restrictions would make it easier for a programmer to work with XML without using an XML parser, but it is unlikely that the XML community will replace XML with something along these lines - especially since there are important usage scenarios for features like DTDs, schemas, and the ability to build compound documents without knowing, at the root level, all of the namespaces that may be used in a document. More to the point, Joe's original reason for trying to solve these problems with XPath and regular expressions was that the standard APIs do not make it easy to solve many simple problems. Looking at his article as a whole, and other articles he has written, we believe that many of these difficulties are caused by the same kind of semantic mismatches that a Native XML Programming Language is designed to solve.

In this paper, we assume that XML will remain as is, and that for general processing, the best approach is to use an XML parser to build a data model instance from the XML documents, and query the data model instance. Not everybody believes this is the best approach. Tim Bray, one of the editors of the original XML specification, objects to the Native XML Programming solution because he objects to the notion of an XML data model: [Bray]

> The notion that there is an "XML data model" is silly and unsupported by real-world evidence. The definition of XML is syntactic: the "Infoset" is an afterthought and in any case is far indeed from being a data model specification that a programmer could work with. Empirical evidence: I can point to a handful of different popular XML-in-Java APIs each of which has its own data model and each of which works. So why would you think that there's a data model there to build a language around?

Tim first says that there is no data model for XML, then argues that there are several. The differences among these data models, while annoying, are not great, and could have been avoided if XML had had a full-fledged data model. The differences between the DOM data model and the XPath data model are well known in the XML world. XQuery, XPath, and XSLT now use one common data model, which can represent both XML and the XML Schema datatypes. Although it would have been convenient if XML had defined a data model, there is no requirement that the data model used by a Native XML Programming Language be the same as any particular data model used in a Java API. As long as the data model supports the structure of XML directly, without losing or adding information in violation of the XML spec, it can be used as the basis for processing.

Tim also suggests that XML is "syntactic", as though this implies that there is no data model. This implies that syntax and structure are opposites, which is rather surprising, since the purpose of a syntax is to describe the structure of a language. In the XML Recommendation, the structure that corresponds to a data model is called the logica structure:

> Each XML document has both a logical and a physical structure. [. . .] Logically, the document is composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup.

Like most modern computer languages, XML uses a BNF to describe the syntactic representation of these structures. For instance, here is a production from the XML Recommendation:

```
[39]  element  :=  EmptyElemTag| STag content ETag
```

The XML Recommendation is largely a description of these logical structures and the relationships among them. For instance, consider the following text:

> Example: The **element** structure of an **XML document** may, for validation purposes, be constrained using element type and attribute-list declarations. An element type declaration constrains the element's **content**.

Element, XML document, and content all refer to logical structures that are represented in the BNF. These logical structures, taken together with the relationships among them as described in the XML Recommendation, come very close to being a data model, but the data model was not fully described.

The whole point of parsing is to create structures from a sequence of characters, using a grammar to determine which structures to create. When a parser is used to interpret the characters of a program in Java, it creates an Abstract Syntax Tree. When it is used to interpret the characters of XML, it creates a data model instance. We use parsers because (1) the parsed structure is more convenient for further processing, (2) the parsed structure distinguishes information from noise, eliminating differences in the character representation that are not significant in the relevant model, and (3) the parsed structure can fill in information not explicitly represented in the serialized form.

However, an XML parser is not enough. A parser creates a convenient representation of XML. We need a Native XML Programming Language to provide convenient processing of this XML.

5.1.3. What should a Native XML Programming Language do?

A Native XML Programming Language must provide the fundamental operations needed for XML. Some of these operations are required because of the structure of XML itself.

A Native XML Programming Language should be able to easily find anything in an XML structure. XQuery, like XSLT, uses XPath for this purpose. Every XPath expression is also an XQuery expression. For instance, if the variable $cust is bound to a Customers element that contains the rows of a relational table, represented using the SQL/XML mappings, then the following path expression finds all the CustIds from that table:

```
$cust/row/CustId
```

A Native XML Programming Language should be able to easily create any XML structure. XQuery uses the syntax of XML for this purpose. For instance, the following XQuery expression creates a Customer element:

```
<Customer>
  <CustId>17</CustId>
  <Name>Ferd Berfle</Name>
</Customer>
```

When XQuery uses the syntax of XML, a curly brace escapes to the syntax of XQuery, allowing dynamic expressions to be inserted. Here is an example that creates a customer with a new unique identifier:

```
<Customer>
  <CustId>{ max( $cust/row/CustId) + 1 }</CustId>
  <Name>Ferd Berfle</Name>
</Customer>
```

A Native XML Programming Language should be able to easily combine and restructure information from XML sources, operating at the logical level without requiring the programmer to think about the internal representation of the XML. For instance, if we are operating on the SQL/XML views of the customers database, the following XQuery combines customers and projects to show the name of a customer and all projects associated with that customer:

```
for $c in $cust/row
let $p := $proj/row[CustId = $c/CustId]
return
  <customer>
     <custName>{ string($c/name) }</custName>
     <projName{ string($p/name) }</projName>
  </customer>
```

A Native XML Programming Language should be able to easily use XML data in expressions. For instance, arithmetic operations should be able to work directly with XML content, observing the data types of typed data and converting appropriately when they encounter untyped data. It should be able to leverage schemas that have been imported into a query, but work well on XML structures for which no schema has been imported.

In short, a Native XML Programming Language should be able to work with XML the way XML users think of it, easily performing the kinds of tasks that XML users need to have done. XQuery attempts to do just that, based on the usage scenarios we gathered in the ### XML Query Use Cases.

5.2. XQuery and SQL/XML Views

Some people seem to believe that the purpose of XQuery is largely the same as that of SQL/XML - to allow XML structures to be created from relational data. Although XQuery is useful for this task, it has relatively few advantages over SQL/XML when this is all that is required. The reason for this is simple: SQL is a language designed for handling SQL data sources, and it does that very well. Adding XML publishing functions to SQL is a simple way to let it create XML. However, it is interesting to note that the SQL/XML views of relational tables have a very constrained structure, and XQuery performed on such views is generally quite similar to the equivalent SQL/XML.

For instance, let's write an XQuery equivalent to the last SQL/XML query we used. This query will operate on a SQL/XML view of the relational tables. The Projects table is represented as follows:

```
<Projects>
  <row>
    <ProjId>2</ProjId>
    <Name>Pegasus</Name>
    <CustId>4</CustId>
  </row>
  <row>
    <ProjId>8</ProjId>
    <Name>Typhon</Name>
    <CustId>4</CustId>
  </row>
  <!-- !!! SNIP !!! -->
```

The Customers table is represented as follows:

```
<Customers>
  <row>
    <CustId>4</CustId>
    <Name>Hardware Heaven</Name>
    <Address>Washington</Address>
  </row>
  <!-- !!! SNIP !!! -->
```

We want to rename these elements and create a representation that shows customers together with their projects. The output should look like this:

```
<customer id = "4">
    <name>Hardware Heaven</name>
    <projects>
     <project id = "2" name = "Pegasus"/>
     <project id = "8" name = "Typhon"/>
    </projects>
  </customer>
```

Here is an XQuery that creates the desired output:

```
for $c in $cust/row
return
  <customer id="{$c/CustId}">
    <name>{ string($c/Name) }</name>
    <projects>
     {
      for $p in $proj/row
      where $p/CustId = $c/CustId
      return
        <project id="{$p/ProjId}" name="{$p/Name}"/>
     }
    </projects>
  </customer>
```

Let's compare this XQuery to the SQL/XML query from a prior section:

SQL/XML	XQuery
`select xmlelement(name customer, xmlattributes(c.CustId as id), xmlforest(c.Name as name, c.City as city), xmlelement(name projects, (select xmlagg(xmlelement (name project, xmlattributes(p.ProjId as id), xmlforest(p.Name as name))) from Projects p where p.CustId=c.CustId))) as "customer-projects" from Customers c`	`for $c in $cust/row return <customer id="{$c/CustId}"> <name>{ string($c/Name)}</name> <projects> { for $p in $proj/row where $p/CustId = $c/CustId return <project id="{$p/ProjId}" name="{$p/Name}"/> } </projects> </customer>`

Table 1.

In this example, as in most such examples, it is hard to argue that either solution is particularly superior to the other. Either SQL/XML or XQuery handle such tasks quite well. The real strength of XQuery is in the ability to easily process XML, whether or not relational data is being processed, including the XML that is frequently stored in columns of relational databases and the XML of web messages. Since XQuery also works well on SQL/XML views of relational data, it is particularly useful when both XML data and relational data must be used in processing. This is explored in the next section.

5.3. Spanning Sources: XQuery, Web Messages, and Databases

XQuery, when combined with a SQL/XML view of a relational database, is extremely good for processing XML together with relational data. This is a very common requirement in many environments, including web message processing environments. To illustrate this, we will use Example 1 from the SOAP Primer. The task is as follows: an incoming message requests a flight to Los Angeles departing from New York as follows:

```
<!-- Example 1 from SOAP Primer -->
<env:Body>
   <p:itinerary xmlns:p="http://travel.org/reservation/travel">

   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2001-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
</p:itinerary>
```

According to the SOAP Primer, the proper response is to point out that there are three airports that depart from New York, so that the user can be prompted to pick one. Here is the desired output:

```

<env:Body>
  <p:itinerary xmlns:p="http://travel.org/reservation/travel">
   <p:airportChoices>JFK LGA EWR</p:airportChoices>
  </p:itinerary>
 </env:Body>
```

Reading between the lines, we assume that there is a database somewhere that lists the airports for each city. The SQL/XML view of the airports table might look like this:

```
<AIRPORTS>
  <row>
    <CITY>Raleigh / Durham</CITY>
    <AIRPORT>RDU</AIRPORT>
   </row>
   <row>
     <CITY>New York</CITY>
     <AIRPORT>JFK</AIRPORT>
   </row>
   <row>
     <CITY>New York</CITY>
     <AIRPORT>LGA</AIRPORT>
   </row>
```

We will assume that when there is only one airport for a city, the output should simply list that city, and that an error should be raised if there is no airport for a given city. The following XQuery handles all three of these cases:

```
for $city in doc("incoming.xml")//p:departing
let $airports := sql:table("airports")/AIRPORTS/row[CITY = $city]
return
    if (count($airports) = 0)
      then <error> No airports found for {$city}!</error>
    else if (count($airports) = 1)
      then <airport>{ string($airports/AIRPORT) }</airport>
    else if (count($airports) > 1)
      then
       <airportChoices>
        {
          for $c in $airports/AIRPORT
          return (string-value( $c ), " ")
        }
       </airportChoices>
    else ()
```

Note that this code operates at a level very close to the application domain, rather than navigating XML documents and converting from XML to appropriate types in the host language. XML data sources and relational data sources are treated in the same way - to the query, they both look like XML documents.

5.4. XQuery for Java (JSR 225)

SQL programmers are used to using APIs such as ODBC or JDBC to set up the environment, execute queries, and do processing in the business domain using the data returned by a query. Similar APIs are expect to emerge for XQuery. The first standard API for this purpose is now being developed under Java Community Process. It is known as XQuery for Java (XQJ), or JSR 225.

Significantly, the requirements of JSR 225 ensure that both XML documents and XML views of databases will be supported, and the results of a query can be processed using JAXP and StAX.

6. SQL/XML and XQuery: Do we need both?

Although SQL/XML and XQuery are both XML query standards, they are based on quite different models, and fit best in different architectures. SQL/XML fits cleanly into the relational model as a reasonably small extension to traditional SQL. This means that it works well in traditional SQL environments, providing full access to the existing SQL language, including features like updates and full-text queries that are not going to be part of XQuery 1.0. One of the other advantages of using SQL as a basis is that database manufacturers have many years of experience in optimizing SQL queries, which means that many of the optimization issues are well known. Also, it has existing APIs, including ODBC and JDBC. In short, SQL/XML provides the functionality needed for creating XML from relational data while still fitting cleanly into the existing SQL environment. SQL/XML implementations will be available from Oracle and IBM, but not Microsoft, and a cross-database implementation is available from DataDirect Technologies. Oracle's implementation also provides functionality for querying and processing XML as well as SQL, and there is some interest in adding extensions along these lines to SQL/XML. Some members of the SQL/XML task force would also like to see parts of XQuery added to SQL/XML.

XQuery fits more cleanly into the XML environment, providing Native XML Programming for both XML sources and non-XML sources accessed via an XML view. It is well designed for combining data from multiple sources, and is very efficient for a variety of XML programming tasks. However, XQuery is a brand new language - in fact, at the time of writing, XQuery 1.0 is merely a Working Draft, not likely to emerge until the second half of 2004. There is a great deal of enthusiasm surrounding XQuery, most major database vendors have announced support for it, and there is a great deal of research on optimizing XQuery. However, XQuery is a much younger language, the industry has little experience optimizing it, and it lacks some features, including updates and full-text, that are very important for some kinds of tasks. Also, the API for XQuery, XQuery for Java (JSR 225) is just now being developed.

Both languages will continue to evolve, trying to fill in the functionality found in the other. On the whole, we feel that SQL/XML is best for SQL programmers who think of their task in terms of SQL, but need to create results in XML. SQL/XML is used much like a report writing language, except that the reports are XML documents. XQuery is best for XML programmers who are working only with XML, or need to work with XML and relational data together. In the short term, implementers and users of XQuery should be aware that it is both new and revolutionary - it shows great promise, but we have less industry experience with XQuery than with SQL/XML.

We are confident that both SQL/XML and XQuery will play an important role in XML queries, and that XQuery will become very important for general purpose XML processing. Native XML Programming is a revolution waiting to happen, and XQuery will be key to this revolution.

Bibliography

[Bosworth] "Speaking XML", by Adam Bosworth. December 2002 Column in XML & Web Services Magazine. Available at http://www.fawcette.com/xmlmag/2002_12/magazine/columns/endtag. Shows why DOM and SAX are awkward for many kinds of programming tasks.

[Bray] "XML Is Too Hard For Programmers", by Tim Bray. Blog entry. Available at http://www.tbray.org/ongoing/When/200x/2003/03/16/XML-Prog. Discusses the problems raised by Bosworth and Gregorio, suggests that XML stream processing be made idiomatic in as many programming languages as possible.

[DOM] Document Object Model (DOM) Level 2. Defined in several documents which can be found at http://www.w3.org/DOM/DOMTR.

[Gregorio1] "Regex-able XML", by Joe Gregorio. Available at http://bitworking.org/news/40. Suggests a subset of XML that would be amenable to processing using XPath and Regular Expressions.

[Gregorio2] "DOM (Drudgery Object Model)", by Joe Gregorio. Available at http://bitworking.org/news/22. Explores the shortcomings of the DOM for everyday programming tasks.

[JSR225] XQuery API for JavaTM (XQJ). Java Specification Request 225. Available at http://www.jcp.org/en/jsr/detail?id=225. Contains a list of proposed requirements that give insight into the scope of this project.

[SQL/XML] "(ISO-ANSI Working Draft) XML-Related Specifications (SQL/XML)", edited by Jim Melton. WG3:HBA-010 = H2-2003-312 = 5WD-14-XML-2003-09. August, 2003. Available at ftp://sqlstandards.org/SC32/WG3/Progression_Documents/WD/5WD-14-XML-2003-09.pdf. The current draft of SQL/XML.

[SQL/XML-A] "SQL/XML and the SQLX Informal Group of Companies", by Andrew Eisenberg and Jim Melton. ACM SIGMOD Volume 30, Number 3. September 2001. Available at http://www.acm.org/sigmod/record/issues/0109/standards.pdf. A report on SQL/XML - much more readable than the specification itself.

[SQL/XML-B] "SQL/XML is Making Good Progress", by Andrew Eisenberg and Jim Melton. ACM SIGMOD Volume 31, Number 2. June 2002. Available at http://www.acm.org/sigmod/record/issues/0206/standard.pdf. A report on SQL/XML - much more readable than the specification itself.

[SQL/XML-JDBC] "SQL/XML in JDBC Applications: The simple way for Java applications to generate XML from SQL queries using the SQL/XML features of SQL 2003", by Jonathan Robie and Peter Coppens. Available at http://www.datadirect.com/products/connectsqlxml/docs/sqlxml_whitep.pdf. Compares the code needed to publish relational data as XML using SQL/XML, JDBC+DOM+SQL, and the proprietary extensions of IBM DB2 UDB, Oracle XSU, and Microsoft SQL Server.

[XQueryTour] "XQuery: A Guided Tour", by Jonathan Robie. Chapter from "XQuery from the Experts" (see below). Available on the web at http://www.datadirect.com/news/whatsnew/xquerybook.asp.

[XQWG] XML Query Working Group Home Page. Available at http://www.w3.org/XML/Query.html. Contains pointers to all XQuery-related specifications, 20 XQuery implementations, various articles and publications.

[XQuery] XQuery 1.0: An XML Query Language. W3C Working Draft 22 August 2003. Most recent version is available at http://www.w3.org/TR/xquery/.

[XQueryUseCases] XML Query Use Cases, W3C Working Draft 22 August 2003. Most recent version is available at http://www.w3.org/TR/xquery-use-cases/.

[XQueryReqs] XML Query Requirements, W3C Working Draft 27 Jun 2003. Most recent version is available at http://www.w3.org/TR/xquery-requirements/.

[DataModel] XQuery 1.0 and XPath 2.0 Data Model, W3C Working Draft May 2003. Most recent version is available at http://www.w3.org/TR/xpath-datamodel/.

[XQuerySemantics] XQuery 1.0 and XPath 2.0 Formal Semantics, 22 August 2003. Most recent version is available at http://www.w3.org/TR/xquery-semantics/.

[XQueryFunctions] XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft 2 May 2003. Most recent version is available at http://www.w3.org/TR/xpath-functions/.

[XQueryExperts] XQuery from the Experts: A Guide to the W3C XML Query Language, by Howard Katz, Don Chamberlin, Denise Draper, Mary Fernandez, Michael Kay, Jonathan Robie, Michael Rys, Jerome Simeon, Jim Tivy, Philip Wadler. Addison-Wesley Pub Co; 1st edition (September 12, 2003) ISBN: 0321180607.

Biography

Jonathan **Robie**
DataDirect Technologies
United States of America

Jonathan Robie is the XML Program Manager at DataDirect technologies, responsible for a SQL/XML product that is currently shipping and an XQuery product that is under development. He is a co-inventor of Quilt, which was the immediate predecessor of XQuery, the W3C XML Query language, and is now an editor of many of the specifications which define the XQuery language. He is also a co-inventor of XQL, an earlier XML query language which was a predecessor of XPath. Jonathan has been significantly involved in several other W3C Working Groups, acting as an editor for documents produced by the XML Schema and Document Object Model Working Groups, and has also participated in the W3C XML Information Set and XML Stylesheet Language (XSL) Working Groups. He is well known in the XML world, both as an innovator and as a speaker.

Prior to joining DataDirect, Jonathan worked as an XML Research Specialist at Software AG, where he helped design architectures for XML servers and represented Software AG on the XML Query and XML Schema Working Groups. He has been on the architecture team for three XML databases or repositories, at Software AG, Texcel Research, and POET Software. Since 1985 he has been working professionally with advanced database systems and complex database applications, especially object oriented databases, multimedia databases, workgroup database applications, and XML/SGML databases.