# SQL/XML is Making Good Progress

Andrew Eisenberg
IBM, Westford, MA 01886
andrew.eisenberg@us.ibm.com

Jim Melton
Oracle Corp., Sandy, UT 84093
jim.melton@acm.org

## Introduction

Not very long ago, we discussed the creation of a new part of SQL, XML-Related Specifications (SQL/XML), in this column [1]. At the time, we referred to the work that had been done as "infrastructure". We are pleased to be able to say that significant progress has been made, and SQL/XML [2] is now going out for the first formal stage of processing, Final Committee Draft (FCD) ballot, in ISO/IEC JTC1.

In our previous column, we described the mapping of SQL <identifier>s to XML Names, SQL data types to XML Schema data types, and SQL values to XML values. There have been a few small corrections and enhancements in these areas, but for the most part the descriptions in our previous column are still accurate.

The new work that we will discuss in this column comes in three parts. The first part provides a mapping from a single table, all tables in a schema, or all tables in a catalog to an XML document. The second of these parts includes the creation of an XML data type in SQL and adds functions that create values of this new type. These functions allow a user to produce XML from existing SQL data. Finally, the "infrastructure" work that we described in our previous article included the mapping of SQL's predefined data types to XML Schema data types. This mapping has been extended to include the mapping of domains, distinct types, row types, arrays, and multisets.

The FCD ballot that we mentioned began in early April. This will allow the comments contained in the ballot responses to be discussed at the Editing Meeting in September or October of this year. We expect the Editing Meeting to recommend progression to Final Draft International Status (FDIS) ballot, which suggests that an International Standard will be published by the middle of 2003.

## Mapping Tables to XML Documents

SQL/XML defines a mapping from tables to XML documents (proposed in [4]). The mapping may take as its source an individual table, all of the tables in a schema, or all of the tables in a catalog. The mapping takes place on behalf of a specific user, so only those tables that contain a column for which the user has SELECT privilege will be included in this mapping.

This mapping produces two XML documents, one that contains the data in the table or tables that were specified, and another that contains an XML Schema that describes the first document.

This mapping builds on the "infrastructure" that we discussed in our previous article. SQL <identifier>s, SQL data types, and SQL values are mapped to XML in the way that we described.

### *Mapping the Data*

An EMPLOYEE table is mapped to an XML document in the following way:

```
<EMPLOYEE>
   <row>
      <EMPNO>000010</EMPNO>
      <FIRSTNAME>CHRISTINE</FIRSTNAME>
      <LASTNAME>HAAS</LASTNAME>
      <BIRTHDATE>1933-08-24</BIRTHDATE>
      <SALARY>52750.00</SALARY>
   </row>
   <row>
      <EMPNO>000020</EMPNO>
      <FIRSTNAME>MICHAEL</FIRSTNAME>
      <LASTNAME>THOMPSON</LASTNAME>
      <BIRTHDATE>1948-02-02</BIRTHDATE>
      <SALARY>41250.00</SALARY>
   </row>
   .
   .
   .
</EMPLOYEE>
```

The root element has been given the name of the table. Each row is contained in a `<row>` element. Each `<row>` element contains a sequence of column elements, each with the name of the column. Each column element contains a data value. The names of the table and column elements are generated using the fully escaped mapping from SQL <identifier>s to XML Names. The data values are produced using the mapping from SQL data values to XML.

The tables of a schema are mapped to an XML document in the following way:

```
<ADMINISTRATOR>

   <DEPARTMENT>
      <row>
         <DEPTNO>A00</DEPTNO>
         <DEPTNAME>Accounting</DEPTNAME>
         <MGRNO>000010</MGRNO>
         <ADMRDEPT>A00</ADMRDEPT>
      </row>
      .
      .
      .
   </DEPARTMENT>

   <ORG>
      <row>
         <DEPTNUMB>10</DEPTNUMB>
         <DEPTNAME>Head Office</DEPTNAME>
         <MANAGER>160</MANAGER>
         <DIVISION>Corporate</DIVISION
         <LOCATION>New York</LOCATION>
      </row>
      .
      .
      .
   </ORG>

</ADMINISTRATOR>
```

This document reflects the DEPARTMENT and ORG tables in the ADMINISTRATOR schema.

The mapping of the tables contained in a catalog will have as its root an element that represents the catalog. This element will contain an element for each of its schemas.

There are many ways to represent a table in XML. The committees decided to use elements to represent column values because the values of columns that use SQL's non-scalar data types could not all be represented by attribute values. A column could, for example, be defined using a row data type with 3 fields of different data types, which doesn't map onto XML's attributes.

The `<row>` element may seem a bit artificial. The committees could have used the name of the table as the name of the element that contains the columns. Doing so would have required the creation of an artificial name for the element that contains all of the table's rows. This might have looked something like the following:

```
<table>
   <EMPLOYEE>
      <EMPNO>000010</EMPNO>
      <FIRSTNAME>CHRISTINE</FIRSTNAME>
      <LASTNAME>HAAS</LASTNAME>
      <BIRTHDATE>1933-08-24</BIRTHDATE>
      <SALARY>52750.00</SALARY>
   </EMPLOYEE>
   .
   .
   .
</table>
```

## Null Values

In addition to specifying the source tables for this mapping, a user must also specify how null values are to be mapped. The two behaviors that are provided are termed "nil" and absent".

If a user chooses "nil", then the attribute `xsi:nil="true"` is used to mark column elements that represent null values. An employee with a null value in her birthday column would appear as follows:

```
<row>
   <EMPNO>000010</EMPNO>
   <FIRSTNAME>CHRISTINE</FIRSTNAME>
   <LASTNAME>HAAS</LASTNAME>
   <BIRTHDATE xsi:nil="true" />
   <SALARY>52750.00</SALARY>
</row>
```

This user could also have chosen "absent", which causes columns with null values be represented by the absence of the corresponding column element. With this choice, the row above would appear as follows:

```
<row>
   <EMPNO>000010</EMPNO>
   <FIRSTNAME>CHRISTINE</FIRSTNAME>
   <LASTNAME>HAAS</LASTNAME>
   <SALARY>52750.00</SALARY>
</row>
```

## Generating an XML Schema

There are many XML schemas that could be written to describe the mapped tables that we have shown. A choice that the committees rejected was to create a single, monolithic element definition with anonymous types. This choice might have generated the following schema for the employee table:

```
<xsd:schema>
  <xsd:element name="EMPLOYEE">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="row"
           minOccurs="0"
           maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="EMPNO">
                <xsd:simpleType>
                  <xsd:restriction
                      base="xsd:string">
                    <xsd:length value="6"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="FIRSTNAME">
                 ...
              </xsd:element>
```

```
                .
                .
                .
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

The committees decided instead to create globally-named XML Schema data types for every type that is required to describe the table or tables that are being mapped.

## Naming all of the Types

Let's look at the schema that is generated for the EMPLOYEE table with "nil" chosen for the mapping of null values.

```
<xsd:schema>

  <xsd:simpleType name="CHAR_6">
    <xsd:restriction base="xsd:string">
      <xsd:length value="6"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="DECIMAL_9_2">
    <xsd:restriction base="xsd:decimal">
      <xsd:totalDigits value="9"/>
      <xsd:fractionDigits value="2"/>
    </xsd:restriction>
  </xsd:simpleType>
  .
  .
  .
  <xsd:complexType
    name="RowType.HR.ADMIN.EMPLOYEE">
    <xsd:sequence>
      <xsd:element
         name="EMPNO" type="CHAR_6"/>
      .
      .
      .
      <xsd:element
         name="SALARY" type="DECIMAL_9_2"
         nillable="true"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType
       name="TableType.HR.ADMIN.EMPLOYEE">
     <xsd:sequence>
       <xsd:element name="row"
         type="RowType.HR.ADMIN.EMPLOYEE"
         minOccurs="0"
         maxOccurs="unbounded"/>
     </xsd:sequence>
   </xsd:complexType>

  <xsd:element name="EMPLOYEE"
     type="TableType.HR.ADMIN.EMPLOYEE"/>

</xsd:schema>
```

We have generated named XML Schema types for each of the column data types. These are used in the definition of the elements that represent each column. The names used for the XML Schema data types that correspond to SQL scalar data types are quite straightforward. Some additional examples of these type names are shown below:

| SQL Data Type | XML Schema Name |
|---|---|
| CHARACTER (5) | CHAR_5 |
| CHARACTER (5) CHARACTER SET LATIN1 | CHAR_5 |
| INTEGER | INTEGER |
| TIMESTAMP(6) | TIMESTAMP_6 |
| TIMESTAMP(0) WITH TIME ZONE | TIMESTAMP_WTZ_0 |
| INTERVAL HOUR TO SECOND (1) | INTERVAL_HOUR_2_SECOND_1 |

A single `DECIMAL_9_2` data type will be included in the XML Schema, regardless of how many additional columns use the `DECIMAL(9,2)` data type. All columns that use `CHAR(6)` will share a `CHAR_6` XML Schema data type, no matter which character sets or collations they use. This choice was made because the XML Schema data type `xsd:string` consists of Unicode characters only.

A named XML Schema type is created for the type of the rows in a table, and for the type of the table as a whole. The name of the row type was constructed by concatenating "`RowType`", the catalog name, the schema name, and the table name, each separated by a ".". To avoid ambiguities, periods within names are escaped. For example, if the catalog name had been "`H.R.`", then the periods in the catalog name would be represented by "`_x002E_`", producing the following XML Name:

```
RowType.H_x002e_R_x002e_.ADMIN.EMPLOYEE
```

These naming rules have been chosen to prevent any conflict among names that are generated for an XML Schema document.

The example above showed the mapping of a table with the choice for nulls of "nil". You'll see this if you look at the definition of the `SALARY` element. If "absent" had been chosen, then the following element definition would have been generated:

```
<xsd:element name="SALARY"
   type="DECIMAL_9_2"
   minOccurs="0"
/>
```

## *Table Mapping and XML Query*

One very interesting use of the table mapping that has been added to SQL/XML is the ability to generate a virtual XML document that can be used by XML

Query [3]. We'll hypothesize a `table` function that returns an XML document produced by the mapping of a single table in the following example:

```
<highemps>
   { for $e in table("Sample_db",
                     "Andrew",
                     "*password*",
                     "HR.ADMIN.EMPLOYEE"
                     )/EMPLOYEE/row
     where $e/SALARY > 40000
     return
        <emp>
           { $e/FIRSTNME, $e/LASTNAME }
        </emp>
   }
</highemps>
```

## XML Data Type

Today, a user might store an XML document as either a VARCHAR or a CLOB value. XML documents can also be decomposed on a client or a middle-tier and stored in scalar columns in one or several tables. The addition of an XML data type in SQL/XML (proposed in [5]) provides the potential for greater capability for users and for greater performance as well.

The XML data type is identified simply as `XML`. Columns, variables, and parameters can be defined using this new data type.

The legal values for this data type consist of documents, elements, forests of elements, text nodes, and mixed content. Attributes can exist within an element, but they are not legal XML values themselves. XML comments and processing instructions are not currently allowed within the XML data type.

Several operators have been provided that produce XML values. These operators are syntactically similar to functions, but they differ from functions in some subtle ways. These operators, XMLELEMENT, XMLFOREST, XMLGEN, XMLCONCAT, and XMLAGG, are each discussed in turn.

We will show SELECT statements returning XML values throughout this section. We are taking a bit of license here in two ways. The XML data type does not yet have a mapping to host language data types, so our SELECT statements are presumed to retrieve values in a pure SQL context. Also, to make these examples readable, we have shown the XML results broken across several lines and indented. This is something that a client tool might do, but SQL would return the value "unformatted".

### *XMLELEMENT*

This operator creates an XML element. The first argument to XMLELEMENT provides the name of the element that is being constructed. This argument is an SQL <identifier>, preceded by `NAME`. The second argument, if it is specified, provides the attributes for the element that is being constructed. It has the form `XMLATTRIBUTES(...)`. The subsequent arguments provide the content for the element that is being constructed. They may be of any SQL data type.

Let's consider the following example (we'll use "→" to indicate possible output from a query):

```
SELECT e.id,
       XMLELEMENT ( NAME "Emp",
                    e.fname
                       || ' ' || e.lname
                  ) AS "result"
FROM   employees e
WHERE  ... ;
```

→

| ID | result |
|---|---|
| 1001 | <Emp>John Smith</Emp> |
| 1206 | <Emp>Bob Martin</Emp> |

The content of the element is provided by a VARCHAR expression that concatenates the employee's first and last name. SQL/XML already contains the mapping from SQL values to XML values that is used to construct this element.

The following example shows the construction of an element with attribute values:

```
SELECT e.id,
       XMLELEMENT (NAME "Emp",
                   XMLATTRIBUTES (
                      e.id,
                      e.lname AS "name"
                      )
                  ) AS "result"
FROM   employees e
WHERE  ... ;
```

→

| ID | result |
|---|---|
| 1001 | <Emp ID="1001" name="Smith"/> |
| 1206 | <Emp ID="1206" name="Martin"/> |

Two attributes have been created in the `<Emp>` element. The name of the first attribute is taken from the name of the ID column. The name of the second attribute is being provided explicitly with `AS "name"`. An attribute name that is taken from a column name will use the fully-escaped mapping from an SQL <identifier> to an XML Name. An attribute name that is provided explicitly will use the partially-escaped version of this mapping.

Nested elements can be created by using nested XMLELEMENT operators. The following example creates elements with mixed content:

```
SELECT e.id,
       XMLELEMENT
           ( NAME "Emp",
             'Employee ',
             XMLELEMENT (NAME "name",
                         e.lname ),
             ' was hired on ',
             XMLELEMENT (NAME "hiredate",
                         e.hire )
           ) AS "result"
FROM   employees e
WHERE  ... ;
```

→

| ID | result |
|---|---|
| 1001 | <Emp><br>Employee **<name>Smith</name>**<br>was hired on<br>**<hiredate>2000-05-24</hiredate>**<br></Emp> |
| 1206 | <Emp><br>Employee **<name>Martin</name>**<br>was hired on<br>**<hiredate>1996-02-01</hiredate>**<br></Emp> |

Null values must always be taken into consideration in SQL. If the value of an attribute is the null value, then that attribute does not appear in the element. If a value that provides content for an element is the null value, then that content is not included in the element.

XML documents increasingly make use of namespaces. An element with a particular namespace would be created as follows:

```
XMLELEMENT
   (NAME "hr:emp",
    XMLATTRIBUTES ('http://www.hr.com/hr'
                   AS "xmlns:hr")
    ...
   )
```

## XMLFOREST

XMLFOREST produces a forest of elements. Each of its arguments is used to create a new element. Like the XMLATTRIBUTES clause that we discussed above, an explicit name for the element can be provided, or the name of the column can be used implicitly.

```
SELECT e.id,
       XMLFOREST ( e.hire,
                   e.dept AS "department"
                 ) AS "result"
FROM employees e
WHERE ... ;
```

→

| ID | result |
|---|---|
| 1001 | <HIRE>2000-05-24</HIRE><br><department>Accounting</department> |
| 1206 | <HIRE>1996-02-01</HIRE><br><department>Shipping</department> |

## XMLGEN

XMLGEN is very similar to an XML Query element constructor. The user provides as its first argument a template that contains placeholders for values that will be supplied later. The placeholders have the form "{$name}". The subsequent arguments provide values with associated names that are used to instantiate the template.

```
SELECT e.id,
   XMLGEN ( '<Emp name="{$NAME}">
               <hire_date>{$HIRE}</hire_date>
               <dept>{$DEPT}</dept>
             </Emp>',
            lname AS name,
            e.hire,
            e.dept
          ) AS "result"
FROM   employees e
WHERE  ... ;
```

→

| ID | result |
|---|---|
| 1001 | <Emp name="Smith"><br><hire_date>2000-05-24</hire_date><br><dept>Accounting</dept><br></Emp> |
| 1206 | <Emp name="Martin"><br><hire_date>1996-02-01</hire_date><br><dept>Shipping</dept><br></Emp> |

Placeholders can be used to specify element content and attribute values, as we have just shown. They can also be used to specify element names and attribute names. This is something that cannot be done with XMLELEMENT.

## XMLCONCAT

XMLCONCAT produces a forest of elements by concatenating its XML arguments.

```
SELECT e.id,
       XMLCONCAT
         ( XMLELEMENT ( NAME "first",
                        e.fname),
           XMLELEMENT ( NAME "last",
                        e.lname)
         ) AS "result"
FROM employees e ;
```

→

| ID | result |
|---|---|
| 1001 | <first>John</first><br><last>Smith</last> |
| 1206 | <first>Mary</first><br><last>Martin</last> |

An argument that is the null value is dropped from the result. If all of the arguments are the null value, then the result is the null value.

### XMLAGG

XMLAGG is an aggregate function that produces a forest of elements from a collection of elements.

```
SELECT XMLELEMENT
         ( NAME "Department",
           XMLATTRIBUTES
              ( e.dept AS "name" ),
           XMLAGG
              ( XMLELEMENT
                   ( NAME "emp", e.lname )
              )
         ) AS "dept_list"
FROM employees e
GROUP BY dept ;
```

→

| dept_list |
|---|
| <Department name="Accounting"><br><emp>Yates</emp><br><emp>Smith</emp><br></Department><br><Department name="Shipping"><br><emp>Oppenheimer</emp><br><emp>Martin</emp><br></Department> |

This query produces a <Department> element with the department's employees contained inside it. Each group of employee rows is evaluated by XMLAGG, producing a collection of XML values. These values are then concatenated to produce a forest of XML values.

The values can also be sorted before concatenation takes place. To sort the employees of each department by their last name, we would write the following:

```
XMLAGG ( XMLELEMENT ( NAME "emp", e.lname )
           ORDER BY e.lname
       )
```

## Mapping non-Predefined Data Types

In our previous paper on SQL/XML, we described the mapping of SQL's predefined data types to XML Schema data types. Since then, a proposal was accepted [6] that added the mapping of some of SQL's non-predefined data types, specifically DOMAIN, Distinct UDT (User-defined Data Type), ROW, ARRAY, and MULTISET.

We will show the XML Schema type definition that is generated for each of these types and the elements that are generated.

Annotations are defined to reflect the SQL metadata that caused the XML Schema type to be generated. The xsd:annotation elements that we show may be generated by an implementation or they may be omitted.

### Domain

Let us consider the definition of a JOBCLASS domain and a column that uses this definition:

```
CREATE DOMAIN jobclass AS INTEGER
DEFAULT 1
CHECK VALUE BETWEEN 0 AND 14

CREATE TABLE employee (
   ...,
   level jobclass,
   ...
   )
```

The XML Schema that is generated for this column includes the following:

```
<xsd:simpleType
   name='DOMAIN.ADMIN.HR.JOBCLASS'>

   <xsd:annotation>
      <xsd:appinfo>
         <sqlxml:sqltype kind='DOMAIN'
             catalogName='ADMIN'
             schemaName='HR'
             typeName='JOBCLASS'
             mappedType='INTEGER'
             final='true'/>
      </xsd:appinfo>
   </xsd:annotation>

   <xsd:restriction base='INTEGER'/>

</simpleType>
```

The domain's underlying data type is used to create the <LEVEL> element.

```
<EMPLOYEE>
   <row>
      ...
      <LEVEL>12</LEVEL>
   </row>
   .
   .
   .
</EMPLOYEE>
```

### Distinct UDT

Stronger typing could be achieved if the creator of the employee table used a Distinct UDT instead of a domain:

```
CREATE TYPE jobclass AS INTEGER FINAL
```

The following XML Schema type definition is generated:

```
<xsd:simpleType
   name='UDT.ADMIN.HR.JOBCLASS'>

   <xsd:annotation>
      <xsd:appinfo>
      <sqlxml:sqltype kind='DISTINCT'
         catalogName='ADMIN'
         schemaName='HR'
         typeName='JOBCLASS'
         mappedType='INTEGER'
         final='true'/>
      </xsd:appinfo>
   </xsd:annotation>

   <xsd:restriction base='INTEGER'/>

</simpleType>
```

The element that is created for a column of this type is the same as it was for a domain.

## Row

A ROW type might be used to define the location of an employee's birth. SQL's ROW types are anonymous. This makes constructing a unique name using only the row's definition impossible. Instead, "ROW." is followed by a unique identifier that is chosen by an implementation. Two columns that use exactly the same ROW type could share an XML Schema definition of the row type, or they could use two separate definitions.

```
CREATE TABLE employee (
   ...,
   birth ROW (city VARCHAR(30),
              state CHAR(2))
   ...
   )
```

The XML Schema that is generated for this table includes the following:

```
<xsd:complexType name='ROW.001'>

  <xsd:annotation>

    <xsd:appinfo>
      <sqlxml:sqltype kind='ROW'>
        <sqlxml:field name='CITY'
          mappedType='VARCHAR_30'/>
        <sqlxml:field name='STATE'
          mappedType='CHAR_2'/>
      </sqlxml:sqltype>
    </xsd:appinfo>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name='CITY'
      nillable='true' type='VARCHAR_30'/>
    <xsd:element name='STATE'
      nillable='true' type='CHAR_2'/>
  </xsd:sequence>

</xsd:complexType>
```

An element that might be generated for a column of this type is:

```
<BIRTH>
   <CITY>Long Beach</CITY>
   <STATE>NY</STATE>
</BIRTH>
```

## Array

An employee might have several phone numbers at which he or she can be reached. A user might choose to use an ARRAY to represent this information.

```
CREATE TABLE employee (
   ...,
   phone CHAR(10) ARRAY[4],
   ...
   )
```

The XML Schema data type that is generated for this SQL data type is:

```
<xsd:complexType name='ARRAY_4.CHAR_10'>

  <xsd:annotation>
    <xsd:appinfo>
      <sqlxml:sqltype
        kind='ARRAY'
        maxElements='4'
        mappedElementType='CHAR_10'/>
      </xsd:appinfo>
    </xsd:annotation>

  <xsd:sequence>
    <xsd:element name='element'
      minOccurs='0' maxOccurs='4'
      nillable='true' type='CHAR_10'/>
    </xsd:element>
  </xsd:sequence>

</xsd:complexType>
```

An element that might be generated for a column of this type is:

```
<PHONE>
   <element>1-333-555-1212</element>
   <element xsi:nil="true"/>
   <element>1-444-555-1212</element>
</PHONE>
```

xsi:nil="true" is used to represent an element of the array whose value is the null value. This use of xsi:nil="true" is independent of the user's choice for the mapping nulls in the table or tables that he or she has chosen.

## Multiset

The employee's phones might be defined using a MULTISET instead of an ARRAY:

```
CREATE TABLE employee (
   ...,
   phone CHAR(10) MULTISET,
   ...
   )
```

The XML Schema data type that is generated for this SQL data type is:

```
<xsd:complexType name='MULTISET.CHAR_10'>

  <xsd:annotation>
    <xsd:appinfo>
      <sqlxml:sqltype kind='MULTISET'
        mappedElementType='CHAR_10'/>
    </xsd:appinfo>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name='element'
      minOccurs='0' maxOccurs='unbounded'
      nillable='true' type='CHAR_10'/>
  </xsd:sequence>

</xsd:complexType>
```

An element that might be generated for a column of this type is:

```
<PHONE>
   <element>1-333-555-1212</element>
   <element xsi:nil="true"/>
   <element>1-444-555-1212</element>
</PHONE>
```

## Future Work

Additional features will likely be introduced before the document progresses from FCD to DIS (Draft Information Standard). These features might include some of the following:

- An operator to create an XML document
- An operator to parse an XML document contained in a CHAR or CLOB (Character Large Object) value, producing an XML value
- An operator to serialize an XML value, producing a CHAR or CLOB value
- An operator that produces a table with scalar columns from an XML value
- CAST to and from the XML data type
- Define the mapping of Structured UDT's to XML
- Predicates to test XML values (is this an element, is this a document, does this contain mixed content, etc.)
- An operator to check the validity of an element or document according to an XML Schema
- An operator that executes an XPath or XQuery expression using one or more XML values

## References


[1] *SQL/XML and the SQLX Informal Group of Companies*, Andrew Eisenberg and Jim Melton, ACM SIGMOD Record, Vol. 30 No. 3, Sept. 2001, http://www.acm.org/sigmod/record/issues/0109/standards.pdf.

[2] *Database Languages – SQL - Part 14: XML-Related Specifications (SQL/XML) – Final Committee Draft*, H2-2002-063, WG3:ICN-011, Jim Melton (Editor), March 2002, ftp://sqlstandards.org/SC32/WG3/Progression_Documents/FCD/4FCD1-14-XML-2002-03.pdf.

[3] *XQuery 1.0: An XML Query Language*, Don Chamberlin, James Clark, Daniela Florescu, Jonathan Robie, Jérôme Siméon, Mugur Stefanescu, Dec. 20, 2001, http://www.w3.org/TR/xquery/.

[4] *Mapping Tables to XML Documents*, H2-2001-373r1, WG3:YYJ-038r1, Andrew Eisenberg, Fred Zemke, Murali Krishnaprasad, Oct. 11, 2001, ftp://sqlstandards.org/SC32/National_Bodies/USA_NCITS_H2/2001docs/H2-2001-373r1.pdf.

[5] *The XML Data Type*, H2-2002-020r2, WG3:VIE-018r1, Amelia Carlson, et al, Feb. 14, 2002, ftp://sqlstandards.org/SC32/National_Bodies/USA_NCITS_H2/2002docs/H2-2002-020r2.pdf.

[6] *Mapping non-predefined SQL types to XML*, H2-2002-018, WG3:VIE-017, Fred Zemke, Jan. 4, 2002, ftp://sqlstandards.org/SC32/National_Bodies/USA_NCITS_H2/2002docs/H2-2002-018.pdf.


## Web References

International Committee for Information Technology Standards (INCITS)
http://www.incits.org/

NCITS H2 – Database Committee
http://www.incits.org/tc_home/h2.htm

ISO/IEC JTC 1/SC32
http://www.jtc1sc32.org

SQLX http://www.sqlx.org

W3C http://www.w3.org