Advancements in SQL/XML

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

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

Introduction

Since we last wrote about SQL/XML in [2], the first edition of that new part of the SQL standard has been officially published as an international standard [1], commonly called SQL/XML:2003. At the time of that earlier column, SQL/XML was just entering its first official ballot, meaning that (possibly significant) changes to the text were expected in response to ballot comments submitted by the various participants in the SQL standardization process.

Since then, SQL/XML completed its ballot and the resulting editing process, with those expected changes incorporated, and has been published as an international standard. The present column summarizes the differences introduced in SQL/XML during that editing process.

In addition, we discuss the new features being added to SQL/XML for its second edition. As a matter of fact, the first official ballot of the second edition [3] is already under way (belying the popular misconception that standards processes inherently operate in geologic time). Publication of the second edition of SQL/XML is anticipated in late 2005.

SQL/XML:2003 – The First Edition

In our previous column addressing SQL/XML [2], we identified the major areas addressed by the document as it existed when its first formal ballot was initiated.

Those areas included:

- Mapping SQL tables, schemas, and catalogs to XML documents
- Generation of an XML schema corresponding to an XML document generated from SQL data
- An XML data type to allow columns of SQL tables to contain XML data
- Publishing functions that allow application writers to create XML directly within SQL queries, including such functions as XMLELEMENT, XMLATTRIBUTES, XMLFOREST, XMLCONCAT, XMLAGG, and XMLGEN

In addition, we also speculated on areas in which future work might be done:

- 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 UDTs 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

As we expected, during the process of resolving ballot comments, a number of changes were made to the SQL/XML specifications, including changes that addressed some of our "future work" items. Among the most interesting of these changes were:

- Removal of the XMLGEN function from the available publishing functions
- Addition of an XMLROOT function to represent the root of an XML structure
- Full alignment of the SQL/XML (first edition) data model with the Information Set, or Infoset, defined by the W3C [4].
- Addition of significant additional power to support the XML type, particularly including a new XMLPARSE function that takes a serialized XML document and converts it into an instance of the XML type
- Addition of support for XML namespaces (version 1.0) [5]
- Addition of a new predicate, DOCUMENT, used to determine whether an XML value is or is not a well-formed XML document
- Addition of a specification for serializing XML values into character strings

SQL/XML:2003 is already implemented in large part by the major SQL database vendors, suggesting excellent prospects for long-term success.

Not all of the "future work" that we mentioned in our earlier column was addressed in the final publication of SQL:2003, so considerable work remained to be done in the next edition.

The Second Edition In Progress

Even before SQL/XML:2003 was finalized in late 2003, the SQL standards community had begun work on the second edition. In this section of this issue's column, we review the most important enhancements and changes that have gone into this next edition of SQL/XML—at the time that it went into its first formal ballot.

XQuery 1.0 and XPath 2.0 Data Model

One of the most significant changes that was made to SQL/XML between the publication of the 2003 standard and the emerging second edition is the change in data mode.

In SQL/XML:2003, as we indicated above, the SQL/XML data model was nothing more than the W3C's Infoset. That decision was made for very pragmatic reasons: the XQuery 1.0 and XPath 2.0 Data Model [6] was not considered stable enough (at that time) to base an international standard on it. By contrast, the Infoset is well understood and has been used as the foundation for several other W3C specifications.

In the ensuing months, however, the XQuery Data Model (as we'll call it in this issue's column) has matured considerably, having reached the W3C's Last Call Working Draft status. Of course, changes continue to be made, but the nature of those changes is substantially less significant than only a year ago.

The change in data model from the Infoset to the XQuery Data Model has had a number of important effects on the SQL/XML specification, as well as some effects on the nature of the language itself. We doubt that our readers have as much interest in the changes to the specification than they do in the changes to the language, so we'll focus on the language aspects.

The XQuery Data Model is based to a large degree on the Infoset, but uses many concepts—as well as the definitions of various atomic types—that are defined in XML Schema [7]. This provides a significant benefit over the Infoset, in which only three atomic types are acknowledged: Boolean, double, and string. By contrast, XML Schema defines a large selection of atomic types that can be used within XML documents. (We discussed XML Schema more extensively in another recent column [8].)

As you'll see later in this column, the XQuery Data Model actually adds new types to those defined by XML Schema. The two types used below (`xdt:untypedAny` and `xdt:untypedAtomic`) are among those new types.

The most visible change to the language is the addition of several "parameters" to the XML type and the syntax necessary to support that now-parameterized type. In SQL/XML:2003, the XML type was identified using only the keyword `XML`.

By contrast, in the second edition, one can identify several variations: `XML(SEQUENCE)`, `XML(ANY CONTENT)`, `XML(UNTYPED CONTENT)`, `XML(ANY DOCUMENT)`, and `XML(UNTYPED DOCUMENT)`. We'll discuss these types in somewhat more detail later, under *XML Type Refinements*.

Another highly visible change is the ability to identify and use the types of atomic values to provide SQL's traditionally strong typing capabilities to queries involving both SQL and XML. Most of these atomic types, as we indicated a few paragraphs above, are taken directly from XML Schema. These include types such as `xs:string`, `xs:float`, and `xs:dateTime`. (We're using the namespace prefix "`xs:`" to identify the XML Schema Part 2 namespace because of its familiarity to many XML practitioners.)

The XQuery Data Model adds a few other new atomic types (in addition to `xdt:untypedAtomic`, mentioned above). In particular, the XML Schema type `xs:duration` causes a number of problems in a data management situation, largely because its values are not fully ordered (*e.g.*, is the value "1 month" greater than, less than, or equal to the value "30 days"?). The XQuery Data Model adds two new types derived from `xs:duration`: `xdt:yearMonthDuration` and `xdt:dayTimeDuration` that resolve the problem by partitioning durations into two fully ordered derived types.

XML Schema support

The second edition of SQL/XML provides support for the association of XML values with XML schemas. This support extends beyond the mere use of the XML Schema [7] type system as part of the XQuery Data Model.

Recall that XML Schema serves multiple purposes. One purpose is to specify the details of a number of atomic types, as we mentioned earlier in this column. A second purpose is to provide a way

for applications to define complex types and assign names to them.

The third (and, some might argue, most significant) purpose is to provide a mechanism by which XML documents can be validated—that is, can be evaluated for proper alignment with a specific XML schema. It is for this third purpose that XML Schema support has been added to this edition of SQL/XML.

SQL/XML now supports a VALID predicate that allows an application to determine whether a given XML value is or is not valid according to a specified schema. The VALID predicate (whose syntax and usage will be explored later in this column) depends on the existence of *registered* XML Schemas.

As a part of the SQL standard, SQL/XML must concern itself with data security. If the VALID predicate allowed XML values to be evaluated against arbitrary XML schemas, wherever they might be found, a variety of security problems could arise, including the possibility that unfriendly entities could "reverse engineer" an organization's metadata through a variety of probing techniques.

To minimize such security problems, SQL/XML requires that every XML schema used to validate XML values must have been previously registered in the SQL environment. Registration requires, at a minimum, provision of a target namespace URI, a schema location URI, a registered name (an ordinary SQL three-part name), a list of the namespaces (including the target namespace) defined by the schema, and a corresponding list of the global element declarations in those namespaces. How schemas are registered (and "unregistered") is left to implementations.

SQL/XML defines three schemas to be permanently registered: the XML Schema namespace commonly referenced by the prefix **xs:**, the XML Schema instance namespace known by the prefix **xsi:**, and the SQL/XML namespace known by the prefix **sqlxml:**.

Registered XML Schemas can be uniquely identified by their target namespace URIs or by a combination of their target namespace URIs and their location URIs (the choice of which is left to the implementation). They are also uniquely identified by their registered names.

XML Type Refinements

As we told you during the data model discussion above, changing the SQL/XML data model (from the Infoset to the XQuery Data Model) made it possible to parameterize the XML type into several subtypes, without eliminating the existing unparameterized XML type. Some of these parameterized types could have been defined in the first edition, using the Infoset as the data model, but the others depend heavily on concepts defined in the XQuery Data Model. Let's look at the various "subtypes" of the XML type defined in SQL/XML's second edition.

Unlike many other types, XML values might be instances of two or even more of these types, as you'll see from reading their descriptions.

XML(SEQUENCE)

The most basic of these XML type variations is `XML(SEQUENCE)`. Every XML value in SQL/XML is either the (SQL) null value or an XQuery sequence and thus is an instance of this type. Even though every XML value is an instance of the type `XML(SEQUENCE)`, not all XML values are instances of any of the other four parameterized types.

For example, SQL/XML supports XML values that are sequences of any combination of XQuery document nodes, ordinary element nodes, and atomic values. A character string representation of such a sequence might look something like this:

```
12, <emp id="12345" sal="65000">Joe
Goodguy</emp>, "It was the best of
times, it was the worst of times.",
1999-05-31T13:20:00-05:00, <?xml
version="1.0" encoding="UTF-8"?>
<library owner="Bob"><books>
<book ISBN="1-234-56789-0"><title>
Gone With My Dreams</title><author>
Johnny Farroute</author><book>
</library>, 0.314159E1
```

That sequence is an instance of no XML type other than `XML(SEQUENCE)`. Sure, it may involve material that is not a well-formed XML value, but it nonetheless represents an XML value that can be managed internally by SQL/XML and XQuery.

XML(ANY CONTENT)

The second variation is `XML(ANY CONTENT)`. Every XML value that is either the null value or an XQuery document node (including, of course, any children of that document node) is an instance of this type. Of course, every instance of this type is also an instance of `XML(SEQUENCE)`.

XML values that are instances of `XML(ANY CONTENT)` are not limited to valid—or even well-formed—documents. Such values might be documents nodes that have several element children, which violates the XML well-formedness rules. Such values might be developed as intermediate results in some query and later pruned to become well-formed document nodes.

XML (UNTYPED CONTENT)

The third of these more specific XML types is `XML(UNTYPED CONTENT)`. Every XML value that is an instance of this type is also an instance of `XML(ANY CONTENT)`, and thus also an instance of `XML(SEQUENCE)`.

However, to be an instance of `XML(ANY CONTENT)`, every XQuery element node that is contained in the tree rooted at the document node has the type `xdt:untypedAny`, and every attribute in that tree has the type `xdt:untypedAtomic`.

In general, XML values that are instances of `XML(UNTYPED CONTENT)` have not been subjected to a Schema validation episode that could be used to determine more precise type information for the elements and attributes in the document. (Even though values of `XML(ANY CONTENT)` or `XML(UNTYPED CONTENT)` might not be well formed, the element descendents of their document nodes can be individually subjected to Schema validation episodes using XQuery construction and validation rules.)

If the XML values *have* undergone some form of Schema validation, then it is likely that at least one element descendent of the document node or at least one attribute in the tree has gained a type annotation. In this case, the value is an instance of `XML(ANY CONTENT)`, but not of `XML(UNTYPED CONTENT)`.

XML(ANY DOCUMENT)

Our fourth variant is `XML(ANY DOCUMENT)`. The difference between an instance of `XML(ANY DOCUMENT)` and an instance of `XML(ANY CONTENT)` is that instances of `XML(ANY DOCUMENT)` are document nodes that have exactly *one* element child (as well as possibly other children that are permitted in document nodes, such as comments and processing instructions).

As a consequence of this definition, all XML values that are instances of `XML(ANY DOCUMENT)` are also instances of `XML(ANY CONTENT)`.

XML(UNTYPED DOCUMENT)

Finally, the fifth XML subtype is `XML(UNTYPED DOCUMENT)`. Every XML value that is either the null value or an XQuery document node that has exactly *one* element child (perhaps including those other children permitted in document nodes, such as comments and PIs) is an instance of this type.

All instances of `XML(UNTYPED DOCUMENT)` are also instances of `XML(UNTYPED CONTENT)`. In addition, all instances of `XML(UNTYPED DOCUMENT)` are also instances of `XML(ANY DOCUMENT)`. Consequentially, instances of `XML(UNTYPED DOCUMENT)` share the restrictions of both "supertypes". In fact, instances of `XML(UNTYPED DOCUMENT)` are instances of all five of these specific variants of the XML type!

New Publishing Functions

As part of the development of the second edition of SQL/XML, several new publishing functions were added to the language.

Each of these new publishing functions allows the return type to be explicitly specified as `RETURNING CONTENT` or for the `RETURNING` clause to be omitted entirely. In either case, the XML type returned by this function is either `XML(ANY CONTENT)` or `XML(UNTYPED CONTENT)` (the choice being left to the implementation). If `RETURNING SEQUENCE` is specified, then the XML type returned is `XML(SEQUENCE)`.

XMLCOMMENT

The simplest of the new publishing functions is XMLCOMMENT, which allows the application to create an XML comment. The syntax of this new publishing function is:

```
XMLCOMMENT ( 'comment content'
  [ RETURNING
    { CONTENT | SEQUENCE } ] )
```

The value of the character string literal (`'comment content'`, in this case) become the content of the comment. In other words, this function creates an XML comment that, if serialized, looks like this:

```
<!-- comment content -->
```

XMLPI

Another simple publishing function added is XMLPI, allowing applications to create XML processing instructions. It syntax is:

```
XMLPI ( NAME target
  [ , string-expression ]
  [ RETURNING
    { CONTENT | SEQUENCE } ] )
```

The `target` is an identifier specifying the target of the created processing instruction. If specified, the `string-expression` is the content of the PI; if it is not specified, then the zero-length string (`''`) is used as the content. In other words, this function creates an XML comment that, if serialized, looks like this:

```
<? target string-expression ?>
```

XMLQUERY

The third publishing function is XMLQUERY, which is arguably the most important of the new functions. The purpose of XMLQUERY is to evaluate an XQuery expression and return the result to the SQL application. The XQuery expression may itself identify the XML value against which it is evaluated using, perhaps, XQuery's fn:doc() function), or the XML value can be passed to the XMLQUERY invocation as a parameter.

The syntax of XMLQUERY is:

```
XMLQUERY ( XQuery-expression
  [ PASSING { BY REF | BY VALUE }
    argument-list ]
  RETURNING { CONTENT | SEQUENCE }
  { BY REF | BY VALUE } )
```

The `XQuery-expression` is, as you might expect, a character string literal containing an XQuery expression. Note that, in this edition of SQL/XML, it must be a literal and not a general character string expression; a future edition might relax this restriction.

The `argument-list` is, of course, a comma-separated list of arguments. Each argument provides a binding between an SQL value (possibly a value of one of the XML subtypes!) and an XQuery global variable that is declared in the `XQuery-expression`. The syntax of each argument is:

```
value-expression AS identifier
  [ BY REF | BY VALUE ]
```

The value of the `value-expression` is the value bound to the argument, which is identified by the `identifier`. If BY REF is specified, then a reference to the value is bound to the variable; if BY VALUE is specified, then the value (more precisely, a *copy* of the value) is bound directly to the variable. The argument passing mechanism specified just before the `argument-list` is applied to each argument for which neither BY REF nor BY VALUE is specified. If the `value-expression`'s type is not an XML type, then the passing mechanism cannot be specified (and the value is bound directly to the variable).

There is one possible exception to the use of an argument as a binding to an XQuery global variable: At most one argument can be used to pass a context item—the context against which the XQuery expression is evaluated. A context item has the syntax:

```
value-expression
  [ BY REF | BY VALUE ]
```

This syntax is very much like the syntax of any other argument, except that no variable name is specified through the use of `AS identifier`. As with other arguments, the value of the `value-expression` can be passed either by reference or by value—unless the context item is not an XML value, in which case it is always passed by value. The context item must always be either the (SQL) null value or an instance of `XML(SEQUENCE)` whose sequence length is one item.

Unlike XMLCOMMENT and XMLPI, the value returned from XMLQUERY can be returned as a reference to a result or as (a copy of) the value of the result itself. There is, however, an interaction between the type of the returned value and the choice of returning by reference or by value: If the return type is `XML(CONTENT)`, then the returning mechanism is implicitly `BY VALUE` (but it cannot be specified explicitly).

If `RETURNING CONTENT` is specified, then the result is serialized before returning it (by value, of course) to the SQL application.

Before we leave the XMLQUERY discussion, here's an example of its use:

```
SELECT top_price,
  XMLQUERY (
    'for $cost in
       /buyer/contract/item/amount
     where /buyer/name = $var1
     return $cost'
    PASSING BY VALUE
      'A.Eisenberg' AS var1,
      buying_agents
    RETURNING SEQUENCE BY VALUE )
FROM buyers
```

XMLCAST

The final item in this section, arguably a publishing function, that has been added to SQL/XML's second edition is XMLCAST, which allows an application to cast a value (either of an XML type or of some other type) to one of the XML types discussed earlier, or to cast a value of one of those XML types to either another one of those XML types or to another type.

The syntax of XMLCAST is very similar to that of the ordinary SQL CAST:

```
XMLCAST ( value-expression AS type )
```

Only values of one of the XML types, or an SQL null value, can be cast to `XML(UNTYPED DOCUMENT)` or `XML(ANY DOCUMENT)`. Neither the type of the `value-expression` nor the specified target `type` can be an SQL collection type, row type, structured type, or reference type. At least one of the types involved—the type of the `value-`

expression or the target type—must be an XML type.

In all cases, the XMLCAST function is syntactically transformed into an ordinary SQL CAST. The reason for providing a separate keyword is to ensure that the various restrictions are sufficiently obvious to application authors that they more easily remember the restrictions associated with casting to and from XML types.

New Predicates

The second edition of SQL/XML provides several new predicates for use by applications, and also enhances an existing predicate.

DOCUMENT predicate

SQL/XML:2003 includes a predicate called the DOCUMENT predicate. The purpose of that predicate was to determine whether an XML value was, in fact, an XML document.

That predicate has been modified slightly for the second edition to align with the change to the XQuery Data Model. Specifically, the DOCUMENT predicate now tests an XML value to see if it is an instance of XML(ANY DOCUMENT) or XML(UNTYPED DOCUMENT). If so, the predicate returns true; otherwise (except for null values, of course), it returns false. The new syntax of this predicate is:

```
XML-value IS [NOT]
  [ANY | UNTYPED] DOCUMENT
```

CONTENT predicate

Similarly, the new CONTENT predicate is used to determine whether an XML value is an instance of XML(ANY CONTENT) or XML(UNTYPED CONTENT). Its syntax is:

```
XML-value IS [NOT]
  [ANY | UNTYPED] CONTENT
```

For both the DOCUMENT predicate and the CONTENT predicate, failure to specify either ANY or UNTYPED has the same result as if you had specified ANY.

XMLEXISTS predicate

The XMLEXISTS predicate serves a somewhat more complex requirement. Its syntax gives a good feel for its purpose:

```
XMLEXISTS ( XQuery-expression
  [ argument-list ] )
```

When you use this predicate in a WHERE clause (or, perhaps less likely, a HAVING clause) in your SQL application, the XQuery expression is evaluated, using the values provided in the argument list, just as though you had invoked the XMLQUERY publishing function. If the value queried by the XQuery expression (perhaps passed as a context item argument) is the SQL null value, then the predicate's result is unknown. If the XQuery evaluation returns an empty XQuery sequence, then the predicate's result is false; otherwise, the result is true.

An important use of the XMLEXISTS predicate is to determine whether an XML document contains some particular content before using a portion of that comment in an expression (in your SELECT list, for example).

In the XMLQUERY example above, if it happened that there is no buyer whose name is "A.Eisenberg" in the document returned from the buying_agents column of the buyers table, then the XMLQuery expression would return the empty sequence, which would be treated by SQL as the null value, and thus the query expression as a whole would return the null value.

Assuming that our application would prefer never to get such null values, but only to return rows with meaningful data in them, we could merely append this WHERE clause to the query expression:

```
WHERE XMLEXISTS (
  XMLQUERY
    ( '/buyer[count(name) = 0]'
      PASSING ... ) )
```

and those undesirable null values are not returned to the application.

VALID predicate

As you saw earlier when we discussed XML Schema support, there is one final new predicate in the second edition of SQL/XML: the VALID predicate.

As we said above, this predicate is used to evaluate an XML value for validity according to an XML schema. The syntax for the VALID predicate is somewhat more complex than the syntax for most predicates. The complexity is due to the flexibility required to make the predicate maximally useful to a variety of application needs.

The high-level view of the syntax is:

```
xml-value IS [ NOT ] VALID
[ identity-constraint-option ]
[ validity-target ]
```

You will see from that syntax that the predicate is used to determine the validity of an XML value (specified as a value expression whose type is one of those five XML types we discussed earlier), that the validity assessment might depend on evaluation of identity constraints (more on that in a moment), and

that an XML schema might be specified for use in the validity assessment.

The `identity-constraint-option` component of that syntax is one of the following choices:

- WITHOUT IDENTITY CONSTRAINTS
- WITH IDENTITY CONSTRAINTS GLOBAL
- WITH IDENTITY CONSTRAINTS LOCAL
- DOCUMENT

If that syntax component is not specified, then the effect is as though WITHOUT IDENTITY CONSTRAINTS had been specified. If DOCUMENT is specified, then the effect is as though a combination of the DOCUMENT predicate and a VALID predicate that specifies GLOBAL had been specified.

WITH IDENTITY CONSTRAINTS GLOBAL means that validity checking is done using the XML rules for ID/IDREF relationships, as well as the XML Schema rules for identity constraint validation. WITH IDENTITY CONSTRAINTS LOCAL means that all of the validity constraints defined in XML Schema (both parts 1 and 2) must be satisfied, but neither the XML ID/IDREF constraints nor the XML Schema rules for identity constraints are evaluated.

It might be surprising to see that the XML schema need not be specified in the predicate. That doesn't mean that the XML value being evaluated is not analyzed with respect to a registered XML schema; it only means that the specific XML schema is selected based on the namespaces of the element node(s) that are being evaluated. The schemas chosen are those schemas whose target namespaces are identical to the namespaces of those element nodes.

The syntax of `validity-target` allows the application to specify whether the schema to be used for validity assessment is identified by its target namespace URI (possibly combined with its location URI) or by its registered name, or whether no namespace is used (NO NAMESPACE, possibly combined with a location URI).

As with any SQL predicate, the result can be true, false, or unknown. The result of unknown can result from a couple of situations. One of these is probably obvious: the XML value expression being evaluated is the SQL null value. But validity assessment can also result in an unknown result if no registered XML schema can be found to validate a particular element or attribute node in a particular namespace.

SQL Tables From XML Values

SQL/XML:2003 focused on turning relational data into XML data and on straightforward storage and retrieval (and minimal processing) of XML documents. In the second edition of this standard, the complementary ability of transforming XML data into relational data is being addressed as well.

The mechanism used to perform that transformation is another pseudo-function, this one named, logically enough, XMLTABLE. XMLTABLE produces a virtual SQL table containing data derived from XML values on which the pseudo-function operates.

The syntax for this new function is:

```
XMLTABLE ( [ namespace-declaration , ]
  XQuery-expression
  [ PASSING argument-list ]
  COLUMNS XMLtbl-column-definitions )
```

The `namespace-declaration` is unchanged from SQL/XML:2003; it is used to declare namespaces that are used in the evaluation of this pseudo-function. The `XQuery-expression` is a character string literal representation of an XQuery expression, exactly as specified in the XMLQUERY pseudo-function we discussed earlier in this column and the `argument-list` is nothing other than the `argument-list` used in the XMLQUERY pseudo-function, except that each argument in the list is always passed by reference.

The `XQuery-expression` is used to identify XML values that will be used to construct SQL rows for the virtual table generated by XMLTABLE.

It is in the `XMLtbl-column-definitions` that things get a bit more interesting. This bit of syntax is a comma-separated list of column definitions derived from the ordinary column definitions used to define ordinary SQL tables. However, this variation comes in two flavors: one produces regular SQL columns, while allowing the provision of another XQuery-expression that specifies the data to be stored in the column being defined; the other creates a special column, an *ordinality column*, that can be used to capture the ordinal position of an item in an XQuery sequence.

The syntax used to specify an ordinality column is:

```
column-name FOR ORDINALITY
```

The `column-name` is the same `column-name` that would be specified in a normal SQL column definition. At most one ordinality column can be defined in a given XMLTABLE invocation.

The syntax used to define a regular SQL column is a bit more complex:

```
column-name data-type
  [ BY REF | BY VALUE ]
  [ default-clause ]
  [ PATH XQuery-expression ]
```

Again, the `column-name` is the same `column-name` that would be specified in a normal SQL column definition. If the `data-type` is XML(SEQUENCE), then either BY REF or BY VALUE must be specified, and the `XQuery-expression` will return a value whose type is XML(SEQUENCE) by reference or by value, respectively. If the `data-type` is anything else, then neither BY REF nor BY VALUE can be specified, and the XQuery-expression returns XML(ANY CONTENT) or XML(UNTYPED CONTENT).

If the PATH clause is not specified, then the column's data comes from an element whose name is the same as the `column-name` and that is an immediate child of the XML value that forms the row as a whole. If PATH is specified, then the `XQuery-expression` is evaluated in the context of the XML value that forms the row as a whole and the result is stored into the column being defined.

The operation of XMLTABLE is very much analogous to *shredding*, which is a mechanism for "disassembling" XML for storage in relational tables. There are many ways to shred XML for relational storage and processing, and XMLTABLE provides a great deal of flexibility to the application author in defining just how that shredding is to be performed.

Once this shredding has taken place, the virtual table can be inserted into a pre-existing SQL base table using an ordinary SQL INSERT statement for persistent storage, or it can just be used in another SQL statement as a virtual table, even possibly in a join expression. The following example illustrates this possibility:

```
INSERT INTO EMPS (ID, NAME, SAL)
  SELECT EMPNO, NAME, SALARY
  FROM XMLTABLE (
    'fn:doc("...emps.xml")//emp'
    PASSING :hv1 AS $dept BY VALUE
    COLUMNS EMPNO INTEGER
            PATH 'badge',
            NAME  VARCHAR(50)
            PATH 'name',
            SALARY DECIMAL(8,2)
            PATH 'comp/salary' )
```

Summary

The second edition of SQL/XML is a significant enhancement over the first edition. The first formal ballot (Final Committee Draft, or FCD) is under way and there is every reason to believe that this new edition will be published in late 2005. By then, work should be fairly far along towards publishing the third edition. It is too early in SQL/XML development to know whether further editions will be required, but we'll definitely keep our readers informed as things develop.

References

[1] ISO/IEC 9075-14:2003 *Information technology – Database languages – SQL – Part 14: XML-Related Specifications,* (SQL/XML)

[2] *SQL/XML is Making Good Progress*, Andrew Eisenberg and Jim Melton, ACM SIGMOD Record, Vol. 31, No. 2, June 2002
Available at: http://www.acm.org/sigmod/record/issues/0206/standard.pdf

[3] ISO/IEC FCD 9075-14, *Information technology – Database languages – SQL – Part 14: XML-Related Specifications,* (SQL/XML)
Available from your country's standards body; in the USA, that body is ANSI (http://www.ansi.org)

[4] *XML Infoset*, John Cowan and Richard Tobin (eds.), 24 October 2001 (W3C Recommendation)
Available at: http://www.w3.org/TR/xml-infoset

[5] *Namespaces in XML*, Tim Bray, Dave Hollander, and Andrew Layman (eds.), 14 January 1999 (W3C Recommendation)
Available at: http://www.w3.org/TR/1999/REC-xml-names-19990114

[6] *XQuery 1.0 and XPath 2.0 Data Model*, Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norm Walsh (eds.), 14 November, 2004 (W3C Last Call Working Draft)
Available at: http://www.w3.org/TR/xpath-datamodel/

[7] *XML Schema Part 0: Primer*, David Fallside (ed.); *XML Schema Part 1: Structures*, Henry Thompson, David Beech, Murray Maloney, and Noah Mendelsohn (eds.); and *XML Schema Part 2: Datatypes*, Paul Biron and Ashok Malhotra (eds.), all 2 May, 2001 (W3C Recommendation)
Available at: http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/, and http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/, respectively

[8] *XML Schema*, Charles E. Campbell, Andrew Eisenberg and Jim Melton, ACM SIGMOD Record, Vol. 32, No. 2, June 2003
Available at: http://www.acm.org/sigmod/record/issues/0306/D7-Standard-JimMelton.pdf