Title: XMLTable

Author: Fred Zemke, Michael Rys, Krishna Kulkarni, Jan-Eike Michels, Berthold Reinwald, Fatma Ozcan, Zhen Liu, Ian Davis, Keith Hare
Source: U.S.A.
Status: Discussion paper
Date: April 21, 2004

Abstract

This paper presents XMLTable, an operator which functions as a derived table in a FROM clause. XMLTable will enable an XML value to be interpreted relationally as a table. This paper builds on [SIA-040], [SIA-041] and [SIA-042], and is the final element in our vision for querying XML data within SQL. This paper also incidentally defines XMLExists, a predicate to test if the result of an XQuery expression evaluation is a non-empty XQuery sequence.

References

[Foundation:2003] Jim Melton (ed), “ISO International Standard (IS) Database Language SQL - Part 2: SQL/Foundation”, ISO/IEC 9075-2:2003

[SQL/XML:2003] Jim Melton (ed), “ISO International Standard (IS) Database Language SQL - Part 14: SQL/XML”, ISO/IEC 9075-14:2003

[Foundation WD] Jim Melton (ed), “Working Draft (WD) Database Language SQL - Part 2: SQL/Foundation”, ISO/IEC JTC1/SC32 WG3:SIA-003 = ANSI INCITS H2-2003-420

[SQL/XML WD] Jim Melton (ed.), “Working Draft (WD) XML-Related Specifications (SQL/XML)”, ISO/IEC JTC1/SC32 WG3:SIA-010 = ANSI INCITS H2-2003-427

[SIA-040] Fred Zemke, “Moving to the XQuery data model”, ISO/IEC JTC1/SC32 WG3:SIA-040 = ANSI INCITS H2-2004-019r1

[SIA-041] Fred Zemke, “XMLCast”, ISO/IEC JTC1/SC32 WG3:SIA-041 = ANSI INCITS H2-2004-020r1

[SIA-042] Fred Zemke, “XMLQuery”, ISO/IEC JTC1/SC32 WG3:SIA-042 = ANSI INCITS H2-2004-021r1

1. Discussion

1.1 A simple example

Our first example is based on the first example in the [XML Schema Primer], titled “The Purchase Order, po.xml”. Note that the example comes from a W3C specification, with no ties to SQL, which is evidence that our proposal addresses realistic user scenarios.

Here is the user data:

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

There are two <item>s, each having an attribute called partNum, and also nested elements called <productName>, <quantity>, <USPrice>, <shipDate> and <comment>. Note that in the example,

one <item> has a <comment> but no <shipDate> and the other <item> has a <shipDate> but no <comment>.

Suppose that this XML value has been stored in a table called PurchaseOrders in a column called XMLpo in a row that may be located by the predicate XMLpo.Keyfield=1. The user would like a way to treat this XML value as the following relational table:

Seqno	Part #	Product Name	Quantity	US Price	Ship Date	Comment
1	872-AA	Lawnmower	1	148.95	*null*	Confirm this is electric
2	926-AA	Baby monitor	1	39.98	DATE '1999-05-21'	*null*

Seqno gives the sequence number of the corresponding <item> in the XML value.

Note the two nulls in the result. This is because the <shipDate> and <comment> elements are missing in their respective rows.

This example can be handled by our proposed XMLTable operator with the following syntax:

```
-- XMLTable example 1
SELECT X.*
FROM PurchaseOrders PO,
     XMLTable ( '//item'
       PASSING PO.XMLpo
       COLUMNS
         "Seqno" FOR ORDINALITY,
         "Part #"       CHAR(6)      PATH '@partnum',
         "Product Name" CHAR(20)     PATH 'productName',
         "Quantity"     INTEGER      PATH 'quantity',
         "US Price"     DECIMAL(9,2) PATH 'USPrice',
         "Ship Date"    DATE         PATH 'shipDate',
         "Comment"      CHAR(80)     PATH 'comment'
     ) AS X
WHERE PO.KeyField = 1
```

The XMLTable operator works as follows:

1. XQuery is initialized with PO.XMLpo as the context node.
2. XQuery executes the expression '//item'. We shall call this expression the *row pattern*, because it specifies how to find the rows within the XML value. The result is a sequence of nodes, each one consisting of an **<item>** element.
3. Each node in the sequence (ie, each **<item>** element) will become a row in the result. Thus the two items in the sequence are:

```
<item partNum="872-AA">
  <productName>Lawnmower</productName>
  <quantity>1</quantity>
  <USPrice>148.95</USPrice>
  <comment>Confirm this is electric</comment>
</item>
```

and

```
<item partNum="926-AA">
  <productName>Baby Monitor</productName>
  <quantity>1</quantity>
  <USPrice>39.98</USPrice>
  <shipDate>1999-05-21</shipDate>
</item>
```

4. To find the columns of a row, apply the corresponding XQuery expressions (called the *column patterns*, found following the PATH keyword for each column) to the <item> element and cast to the column's declared type. For example, to compute the column "Part #" in the first row, the column pattern is '@partNum', which locates the attibute node partNum="872-AA". To cast this to declared type, we extract the value, a text node, and cast it to CHAR(6), giving the result '872-AA' in SQL. The keyword PATH was chosen because, in most cases, the column pattern will be an XPath expression (a particular kind of XQuery expression).
5. The ordinality column "Seqno" is assigned the sequential item number of the item in the sequence.

1.2 DEFAULT clause

Optionally, the user can specify a default value for each column using a DEFAULT clause. The default value is used if the column pattern of a column does not find anything. For example, suppose that the user wants a default comment saying "-- NO COMMENT--". In that case the column definition for the Comment column might be:

```
"Comment" CHAR(80) PATH 'comment'
          DEFAULT '-- NO COMMENT --'
```

and the result table would look like this:

Seqno	Part #	Product Name	Quantity	US Price	Ship Date	Comment
1	872-AA	Lawnmower	1	148.95	*null*	Confirm this is electric
2	926-AA	Baby monitor	1	39.98	DATE '1999-05-21'	-- NO COMMENT --

1.3 Defaulting the column pattern

Looking at the example, we see that in most cases, the column names are almost the same as the column pattern. If the user is willing to name a columns the same as its column pattern, the user can simply omit the PATH clause. This can be done for any or all columns, on an individual basis. In our example, if we take advantage of this simplification for every column, the example would then be written:

```
-- XMLTable example 2
SELECT X.*
FROM PurchaseOrders PO,
     XMLTable ( '//item'
       PASSING PO.XMLpo
       COLUMNS
         "Seqno" FOR ORDINALITY,
         "@partnum"    CHAR(6),
         "productName" CHAR(20),
         "quantity"    INTEGER,
         "USPrice"     DECIMAL(9,2),
         "shipDate"    DATE,
         "comment"     CHAR(80) DEFAULT '-- NO COMMENT --'
     ) AS X
WHERE PO.KeyField = 1
```

The columns are now named "@partnum", "productName", "quantity", "USPrice", "shipDate", "comment" and "Seqno". Many users may actually prefer to name their columns in this way, since it is closely aligned with what they see in their XML data.

Defaulting the column pattern is done on a column-by-column basis. For example, if the user would prefer not to name a column "@partnum", the user could give that column an explicit path, while allowing all the others to default, like this:

```
-- XMLTable example 3
SELECT *
FROM PurchaseOrders PO,
     XMLTable ( '//item'
       PASSING PO.XMLpo
       COLUMNS
         "Seqno" FOR ORDINALITY,
         "Part #"      CHAR(6) PATH '@partnum',
         "productName" CHAR(20),
         "quantity"    INTEGER,
         "USPrice"     DECIMAL(9,2),
         "shipDate"    DATE,
         "comment"     CHAR(80) DEFAULT '-- NO COMMENT --'
     )
WHERE PO.KeyField = 1
```

Another possibility is to use the AS clause to change the names of the columns, like this:

```
-- XMLTable example 4
SELECT X.*
FROM PurchaseOrders PO,
     XMLTable ( '//item'
       PASSING PO.XMLpo
       COLUMNS
         "Seqno" FOR ORDINALITY,
         "@partnum"    CHAR(6),
         "productName" CHAR(20),
         "quantity"    INTEGER,
         "USPrice"     DECIMAL(9,2),
         "shipDate"    DATE,
         "comment"     CHAR(80) DEFAULT '-- NO COMMENT --'
     ) AS X ("Part #", "Product Name", "Quantity",
             "US Price", "Ship Date", "Comment", "Seqno")
WHERE PO.KeyField = 1
```

I personally do not recommend this approach because it is positional, which becomes difficult to read and maintain when there are many columns, but it is available.

1.4 Namespaces

XQuery expressions may utilize namespaces. These namespaces can of course be declared in the prolog of the XQuery expression. However, if the user has already declared namespaces using XMLNamespaces in a scope that includes XMLTable, then the user probably wants those namespace declarations to be visible inside XQuery. For example,

```
-- XMLTable example 5
WITH XMLNamespaces (DEFAULT 'http://www.example.com/PO1')
SELECT X.*
FROM PurchaseOrders PO,
     XMLTable ( '//item'
       PASSING PO.XMLpo
       COLUMNS
         "Seqno" FOR ORDINALITY,
         "@partnum"    CHAR(6),
         "productName" CHAR(20),
         "quantity"    INTEGER,
         "USPrice"     DECIMAL(9,2),
         "shipDate"    DATE,
         "comment"     CHAR(80) DEFAULT '-- NO COMMENT --'
     ) AS X ("Part #", "Product Name", "Quantity",
             "US Price", "Ship Date", "Comment")
WHERE PO.KeyField = 1
```

Our convention regarding XMLNamespaces has been to allow it as an optional first argument on any pseudofunction that utilizes XMLNamespaces in an outer scope (for example, XMLElement and XMLForest). The one exception has been XMLQuery, where we felt that the user could use the XQuery prolog if he only wishes to declare a namespace local to the XMLQuery invocation.

XMLTable uses multiple XQuery expressions (the row pattern and the column patterns). Each of these XQuery expressions may of course declare namespaces using the XQuery prolog. However, in many instances, the user will want to have the same namespace declarations in all of these XQuery expressions. To support this, we allow an optional XMLNamespaces within XMLTable. For example:

```
-- XMLTable example 6
SELECT X.*
FROM PurchaseOrders PO,
     XMLTable (
       XMLNamespaces (DEFAULT 'http://www.example.com/PO1'),
       '//item'
       PASSING PO.XMLpo
       COLUMNS
         "Seqno" FOR ORDINALITY,
         "@partnum"    CHAR(6),
         "productName" CHAR(20),
         "quantity"    INTEGER,
         "USPrice"     DECIMAL(9,2),
         "shipDate"    DATE,
         "comment"     CHAR(80) DEFAULT '-- NO COMMENT --'
     ) AS X ("Part #", "Product Name", "Quantity",
             "US Price", "Ship Date", "Comment")
WHERE PO.KeyField = 1
```

1.5 Example of the syntactic transformation

The preceding example can be used to illustrate every aspect of the syntactic transformation. The fully expanded equivalent is

```
-- XMLTable example 7
SELECT X.*
FROM PurchaseOrders PO,
     LATERAL (
       WITH XMLNamespaces
              (DEFAULT 'http://www.example.com/PO1')
       SELECT
       -- "Seqno" FOR ORDINALITY,
       I.N AS "Seqno",
       -- "@partnum" CHAR(6),
       XMLCast ( XMLQuery ( '@partnum' PASSING BY REF I.V
```

```
                              RETURNING SEQUENCE BY VALUE )
                        AS CHAR(6)),
          --  "productName" CHAR(20),
          XMLCast ( XMLQuery ( 'productName' PASSING BY REF I.V
                              RETURNING SEQUENCE BY VALUE )
                        AS CHAR(20)),
          -- "quantity" INTEGER,
          XMLCast ( XMLQuery ( 'quantity' PASSING BY REF I.V
                              RETURNING SEQUENCE BY VALUE )
                        AS INTEGER),
          -- "USPrice" DECIMAL(9,2),
          XMLCast ( XMLQuery ( 'USPrice' PASSING BY REF I.V
                              RETURNING SEQUENCE BY VALUE )
                        AS DECIMAL(9,2)),
          -- "shipDate" DATE,
          XMLCast ( XMLQuery ( 'shipDate' PASSING BY REF I.V
                               RETURNING SEQUENCE BY VALUE )
                        AS DATE),
          -- "comment" CHAR(80) DEFAULT '-- NO COMMENT --'
          CASE WHEN XMLExists ( 'comment' PASSING BY REF I.V )
          THEN XMLCast ( XMLQuery ( 'comment'
                                    PASSING BY REF I.V
                                  RETURNING SEQUENCE BY VALUE )
                        AS CHAR(80))
          ELSE CAST ('-- NO COMMENT --' AS CHAR(80))
          END
        FROM XMLIterate ( XMLQuery ( '//item'
                                     PASSING BY REF PO.XMLpo
                                     RETURNING SEQUENCE ) ) )
             AS I (V, N)
WHERE PO.KeyField = 1
```

The attentive reader will notice that the syntactic transformation uses two more primitives that have not been presented yet: XMLIterate and XMLExists. These are discussed in the next two sections.

1.6 XMLIterate

XMLIterate is a primitive, defined solely to support the specification of XMLTable via the syntactic transformation, and not exposed to the user. XMLIterate will take a value *V* of type XML(SEQUENCE), and create a table with two columns. The table will have one row for each item in the sequence which is *V*. The first column of the *n*-th row will be of type XML(SEQUENCE) and will contain the *n*-th item in the sequence *V*. The second column of the *n*-th row will be an exact numeric of scale zero with the value *n*.

There is no need to expose XMLIterate to the user, since it is logically equivalent to a special case of XMLTable. Thus

```
FROM XMLIterate ( V ) AS T (Item, Ord)
```

is equivalent to

```
FROM XMLTable ('for $I in $A return $I'
               PASSING V AS A
               COLUMNS
                 Item XML(SEQUENCE) PATH '.',
                 Ord FOR ORDINALITY)
```

From a specification standpoint, it is easier to isolate XMLIterate as the primitive that is charged with converting a sequence into a table, but theoretically the two operators are equivalent since each can be reduced to the other.

1.7 XMLExists

XMLExists is a predicate that returns ***False*** if the result of an XQuery expression is an empty sequence, and ***True*** otherwise. The syntax is the same ax XMLQuery, except for the name of the operator itself. This is analogous to the EXISTS predicate in conventional SQL, which is provided as a user convenience, even though it is reducible to testing a COUNT.

Unlike XMLIterate, XMLExists adds value for the user, so we have made it accessible via a conformance feature.

1.8 Using general XQuery expressions

We believe that most cases in practice will only need XPath expressions as the row and column patterns. However, any row or column pattern may be an arbitrary XQuery expression, such as a FLWOR expression. The only requirement is that the row pattern return a sequence of nodes, such that each node can be analyzed by the column patterns to find the columns. (This is a semantic requirement that will be imposed at run-time by the GRs. It would be difficult to express this as a syntax requirement.)

The following query correlates Order table and BadCustomer table to find orderitems from bad customers.

```
-- XMLTable example 8
SELECT t.cname, t.description
FROM   Order o,
       BadCustomers b,
       XMLTable ('for $i in $xorder/order/orderitem
                  let $c := $i/../customer
                  where $c = $cname
                  return
                    <row>
```

```
                    $c, $i/description
                </row>'
          PASSING O.XOrder as "xorder",
                  B.CName as "cname"
          COLUMNS
            CName       CHAR(20) PATH "customer",
            Description CHAR(40) PATH "description"
        ) AS t
```

The example works as follows. O.XOrder and B.CName are passed in to the row pattern evaluation, which evaluates an XQuery FLWOR expression (for-let-where-order-return expression, somewhat analogous to a <query specification> in SQL). The return clause of this FLWOR expression creates an XML element called <row>, containing two subelements, $c and $i/description. Note that $c is a <customer> element (by virtue of the let clause) and $i/description is a <description> element. A typical <row> element might look like this:

```
<row>
  <customer>BigFoot</customer>
  <description>Sneakers</description>
</row>
```

The column patterns have been chosen based on the knowledge that each <row> contains one <customer> and one <description> element.

If the user is willing for the column names to be the same as the element names, the user can dispense with the PATH clause, as follows:

```
-- XMLTable example 9
SELECT t."customer", t."description"
FROM   Order o,
       BadCustomers b,
       XMLTable ('for $i in $xorder/order/orderitem
                  let $c := $i/../customer
                  where $c = $cname
                  return
                    <row>
                         $c, $i/description
                    </row>'
          PASSING O.XOrder as "xorder",
                  B.CName as "cname"
          COLUMNS
            "customer"    CHAR(20),
            "description" CHAR(40)
        ) AS t
```

Alternatively, since the user is generating the returned element nodes explicitly, the user can control the names of the subelements of <row>, aligning them with the desired column names, like this:

```
-- XMLTable example 10
SELECT t.cname, t.description
FROM   Order o,
       BadCustomers b,
       XMLTable ('for $i in $xorder/order/orderitem
                  let $c := $i/../customer
                  where $c = $cname
                  return
                    <row>
                      <CNAME>$c/text()</CNAME>,
          <DESCRIPTION>$i/description/text()</DESCRIPTION>
                    </row>'
         PASSING O.XOrder as "xorder",
                 B.CName as "cname"
         COLUMNS
           cname       CHAR(20),
           description CHAR(40)
       ) AS t
```

In this example, the user has forced the subelement names to be <CNAME> and <DESCRIPTION>, to be the same as the default names for the columns cname and description (which will be uppercased by SQL since they have not been surround by double quotes).

1.9 Duplicate element names

Coming back to the sample data from [XML Schema: Primer] with which we started, supppose the user wishes to extract the subelements of the <shipTo> and <billTo> elements. These subelements have the same names (<name>, <street>, <city>, <state> and <zip>). Duplicate element names are not a problem for XML, but they are a problem if they are used as column names in SQL. There are a number of ways to handle this. One way is to use explicit column patterns, like this:

```
-- XMLTable example 11
SELECT X.*
FROM PurchaseOrders PO,
     XMLTable ('/purchaseOrder' PASSING PO.XMLpo
       COLUMNS
         ShipName   CHAR(20) PATH 'shipTo/name',
         ShipStreet CHAR(40) PATH 'shipTo/street',
         ShipCity   CHAR(40) PATH 'shipTo/city',
         ShipState  CHAR(2)  PATH 'shipTo/state',
         ShipZip    CHAR(5)  PATH 'shipTo/zip',
```

```
          BillName   CHAR(20) PATH 'billTo/name',
          BillStreet CHAR(40) PATH 'billTo/street',
          BillCity   CHAR(40) PATH 'billTo/city',
          BillState  CHAR(2)  PATH 'billTo/state',
          BillZip    CHAR(5)  PATH 'billTo/zip'
     ) AS X
WHERE PO.Keyfield = 1
```

Another possibility is to adopt the XPath expressions as the column names:

```
-- XMLTable example 12
SELECT X.*
FROM PurchaseOrders PO,
     XMLTable ('/purchaseOrder' PASSING PO.XMLpo
       COLUMNS
         "shipTo/name"   CHAR(20),
         "shipTo/street" CHAR(40),
         "shipTo/city"   CHAR(40),
         "shipTo/state"  CHAR(2),
         "shipTo/zip"    CHAR(5),
         "billTo/name"   CHAR(20),
         "billTo/street" CHAR(40),
         "billTo/city"   CHAR(40),
         "billTo/state"  CHAR(2),
         "billTo/zip"    CHAR(5)
     ) AS X
WHERE PO.Keyfield = 1
```

Another possibility is to drill down through the <shipTo> and <billTo> elements as separate tables, like this:

```
-- XMLTable example 13
SELECT S.*, B.*
FROM PurchaseOrders PO,
     XMLTable ('/purchaseOrder' PASSING PO.XMLpo
       COLUMNS
         "shipTo"  XML,
         "billTo"  XML
     ) AS X,
     XMLTable ('/shipTo' PASSING X."shipTo"
       COLUMNS
         "name"   CHAR(20),
         "street" CHAR(40),
         "city"   CHAR(40),
         "state"  CHAR(2),
         "zip"    CHAR(5)
```

```
      ) AS S,
      XMLTable ('/billTo' PASSING X."billTo"
        COLUMNS
          "name"   CHAR(20),
          "street" CHAR(40),
          "city"   CHAR(40),
          "state"  CHAR(2),
          "zip"    CHAR(5)
      ) AS B
WHERE PO.Keyfield = 1
```

A different situation is if the duplicate element names have the same path. In that case, they can still be distinguished by their sequence. For example, suppose that a <book> element can have up to four <author> subelements, as in this data fragment:

```
<library>
  <book>
    <title>Introduction to Automata Theory</title>
    <author>John E. Hopcroft</author>
    <author>Rajeev Motwani</author>
    <author>Jeffrey D. Ullman</author>
  </book>
  <book>
    <title>Mastering Regular Expressions</title>
    <author>Jeffrey E. F. Friedl</author>
  </book>
</library>
```

One solution would be to map the <author> elements to an array or multiset column. However, many implementations do not support collection types, in which case the user may want to convert each <author> subelement to a separate column. This might be done using either explicit paths, like this:

```
-- XMLTable example 14
SELECT B.*
FROM Library L,
     XMLTable ('book' PASSING L.XMLvalue
       COLUMNS
         Title CHAR (80) PATH 'title',
         Author1 CHAR (40) PATH 'author[1]',
         Author2 CHAR (40) PATH 'author[2]',
         Author3 CHAR (40) PATH 'author[3]',
         Author4 CHAR (40) PATH 'author[4]'
     ) AS B
```

or by adopting the paths as column names:

```
-- XMLTable example 15
SELECT B.*
FROM Library L,
     XMLTable ('book' PASSING L.XMLvalue
       COLUMNS
         "title" CHAR (80),
         "author[1]" CHAR (40),
         "author[2]" CHAR (40),
         "author[3]" CHAR (40),
         "author[4]" CHAR (40)
     ) AS B
```

1.10 BY VALUE and BY REF

[SIA-040] introduced the syntactic choice BY VALUE or BY REF to indicate whether a value is passed by making a copy (which loses node identity) or not. Value semantics also includes the loss of reverse XPath axes (such as parent) whereas reference semantics retains the ability to perform reverse axes. We need to examine the impact of value vs. reference semantics on XMLTable.

Here is an excerpt of the syntactic transformation shown earlier:

```
SELECT . . .
FROM PurchaseOrders PO,
     LATERAL ( . . .
               SELECT . . .
       XMLCast ( XMLQuery ( '@partnum' PASSING BY REF I.V
                            RETURNING SEQUENCE BY VALUE )
                       AS CHAR(6)),
       . . .
     FROM XMLIterate ( XMLQuery ( '//item'
                                  PASSING BY REF PO.XMLpo
                                  RETURNING SEQUENCE ) ) )
          AS I (V, N)
```

The excerpt shows two XMLQuery invocations: the row pattern ('//item') and one column pattern ('@partnum'). Each of these XMLQuery invocations has one input and one output, for a total of four XML values that we need to review concerning BY REF and BY VALUE.

1. The first value is the input to the row pattern, which is passed in BY REF. As it happens, the row pattern ('//item') does not perform any reverse axes, so in this example, you would get the same result using BY VALUE. However, for maximum generality, the input(s) to the row pattern should be by reference. We have defined XMLTable to always pass the input(s) to the row pattern by reference. Implementations may, if they wish, detect if the row pattern does not actually use reverse axes and substitute value semantics as an optimization.

2. The output of the row pattern is also passed by reference. (This is implicit since the PASSING clause established BY REF as the default and the RETURNING SEQUENCE clauses does not override it.)

3. The output of the row pattern becomes input to XMLIterate, which breaks it into one row per XQuery item. This operator is defined to pass its input and output by reference. Thus the output of XMLIterate is by reference.

4. The output of XMLIterate becomes the input to the column pattern ('@partnum'). This input is received by reference. Thus we have an unbroken chain of values passed by reference to this point. This means that the column pattern can perform reverse XPath axes (not shown in this example).

5. Finally, we come to the output of the column pattern. This output becomes input to an XMLCast. There are three subcases to consider:

   a) The target type is an SQL/Foundation built-in type. In that case, passing the value out of the column pattern by value is sufficient. This is the case in the example, where the target type is CHAR(6) and the return mechanism from the column pattern is RETURNING SEQUENCE BY VALUE.

   b) The target type is XML(CONTENT) or XML(DOCUMENT). In that case, the syntactic transformation will utilize the RETURNING CONTENT option of XMLQuery to insure that the result is an XQuery document node. RETURNING CONTENT is necessarily by value.

   c) The target type is XML(SEQUENCE). In that case, there actually is a meaningful choice for the user, whether to use BY REF or BY VALUE. We could not agree on a default, so we propose mandatory syntax as part of the column definition in that case.

The summary is that all the intermediate stages in the syntactic transformation use reference semantics. The final stage, which results in an actual column value of the result table, is usually handled with value semantics, the one exception being if the target type is XML(SEQUENCE), in which case we propose a mandatory choice of either BY REF or BY VALUE.

1.11 Syntax

```
<XML table> ::=
     XMLTABLE <left paren>
     [ <XML namespace declaration> <comma> ]
     <XML table row pattern>
     <XML table argument list>
     COLUMNS <XML table columns>

<XML table row pattern> ::=
     <character string literal>
```

```
<XML table argument list> ::=
     PASSING <XML query argument>
          [ {, <XML query argument> }... ]

<XML table columns> ::=
       <XML table column definitions>
     | <XML table like clause>

<XML table column definitions> ::=
     <XML table column definition>
     [ { , <XML table column definition> } ]

<XML table column definition> ::=
       <XML table ordinality column definition>
     | <XML table regular column definition>

<XML table ordinality column definition> ::=
     <column name> FOR ORDINALITY

<XML table regular column definition> ::=
     <column name> <data type> [ <XML passing mechanism> ]
     [ <default clause> ]
     [ PATH <XML table column pattern> ]

<XML table column pattern> ::=
     PATH <character string literal>

<XML table like clause> ::=
     LIKE <table name> [ <like options> ]
     [ <XML table column options>
       { <comma> <XML table column options }... ]
```

[NOTE to the proposal reader: <like options> is defined in Foundation 11.3 <table definition>.]

```
<XML table column options> ::=
     <column name> WITH OPTIONS
     [ <default clause> ] [ <XML table column path> ]
```

1.12 LIKE clause

(We have discussed and agreed on the following feature, which is not actually found in the proposal, owing to time constraints. We are entering a Language Opportunity for the time being.)

Instead of writing out column definitions, the user can specify that the result table will be like an existing one using a <like clause> borrowed from <table definition>. In this case, all column patterns will default based on column names, unless overridden by optional syntax shown later. If INCLUDING DEFAULTS is specified, then the defaults are copied, otherwise not. If column default values are copied, they can be overridden by optional syntax shown later. If INCLUDING

IDENTITY is specified, then identity columns are created as well. Identity columns are defined by accessing a sequence generator, so they have no column pattern. If INCLUDING GENERATED is specified, then generated columns are also created. Generated columns are defined by computational expressions on other columns and do not have a column pattern.

As mentioned, the default value and/or column pattern can be overridden for individual columns. The syntax for these overrides is modeled on <column options> in SQL/Foundation 12.3 <table definition>. Such overrides come after the INCLUDING clause.

For example, suppose that MyPOs is an existing table defined as

```
CREATE TABLE MyPOs (
         "part #"      CHAR(6),
         "productName" CHAR(20),
         "quantity"    INTEGER,
         "USPrice"     DECIMAL(9,2),
         "shipDate"    DATE,
         "comment"     CHAR(80)
)
```

We wish to model on this table, but there are two problems: the path for "part #" needs to be '@partnum', and the default value for "comment" needs to be '-- NO COMMENT --'. These problems are solved using overrides, like this:

```
-- XMLTable example 16
FROM PurchaseOrders PO,
     XMLTable ( '//item'
       PASSING PO.XMLpo
       COLUMNS LIKE MyPOs
         "Seqno" FOR ORDINALITY,
         "part #" WITH OPTIONS PATH '@partnum',
         "comment"WITH OPTIONS DEFAULT '-- NO COMMENT --'
     ) AS X
```

1.13 Discussion question: is XMLITERATE a <reserved word>?

This paper introduces XMLIterate as an operator in the FROM clause, purely for definitional purposes — XMLIterate is not exposed to the user. Syntactically, XMLITERATE is a keyword occurring in a context that requires a reserved word. However, this keyword will never be exposed to the user, so there is no actual harm if a user names a table, column, etc., as XMLITERATE.

2. Proposal conventions

This proposal uses the following conventions:

1. SMALLCAPS denote numbered editorial instructions;

~~strikeout~~	denotes existing text to be deleted;
bold red	denotes new text to be inserted;
plain	denotes existing text to be retained;
[Note:...]	brackets enclose italicized notes to the proposal reader;
⌊‾‾‾‾⌋	boxes surround "editing tags," which are part of the document (not instructions to the editor) and may be deleted, inserted, modified or retained, depending on the typeface within the box.

3. Proposal for [SQL/XML WD]

3.1 Changes to 4.3.2, "Operations involving XML values"

1. APPEND THE FOLLOWING PARAGRAPH:

XMLTable is a kind of <derived table> in the <from clause>, which may be used to query an XML value as a table.

3.2 Changes to 5.1, <token> and <separator>

1. ADD THE FOLLOWING TO <NON-RESERVED WORD>:

```
<NON-RESERVED WORD> ::=
      . . .
      | COLUMNS
      | PATH
```

2. ADD THE FOLLOWING <RESERVED WORD>S:

```
<reserved word> ::=
      . . .
     XMLEXISTS | XMLITERATE | XMLTABLE
```

3.3 New Subclause 7.0, <table reference>

1. INSERT THE FOLLOWING SUBCLAUSE IN CLAUSE 7, QUERY EXPRESSIONS

7.n <table reference>

Function

Reference a table

Format

```
<table primary> ::=
       !! all alternatives in 9075-2
     | <XML iterate> [ AS ] <correlation name>
```

```
        <left paren> <derived column list> <right paren>
      | <XML table> [ AS ] <correlation name>
        [ <left paren> <derived column list> <right paren> ]

<XML iterate> ::=
     XMLITERATE
     <left paren> <XML value expression> <right paren>

<XML table> ::=
     XMLTABLE <left paren>
     [ <XML namespace declaration> <comma> ]
     <XML table row pattern>
     [ <XML table parameters> ]
     COLUMNS <XML table column definitions>

<XML table row pattern> ::=
     <character string literal>

<XML table parameters> ::=
     PASSING <XML query argument>
          [ { <comma> <XML query argument> }... ]

<XML table column definitions> ::=
     <XML table column definition>
     [ { , <XML table column definition> } ]

<XML table column definition> ::=
        <XML table ordinality column definition>
     | <XML table regular column definition>

<XML table ordinality column definition> ::=
     <column name> FOR ORDINALITY

<XML table regular column definition> ::=
     <column name> <data type> [ <XML passing mechanism> ]
     [ <default clause> ]
     [ PATH <XML table column pattern> ]

<XML table column pattern> ::=
     <character string literal>
```

— Editor's Note —

It would be desirable for XMLTable to have a LIKE clause, so that a result table could be modeled on an existing table.

Syntax Rules

1) | Replace Syntax Rule 5) | If a <table factor> *TF* is contained in a <from clause> *FC* with no intervening <query expression>, then the *scope clause SC* of *TF* is the <select statement: single row> or innermost <query specification> that contains *FC*. The scope of the range variable of *TF* is the <select list>, <where clause>, <group by clause>, <having clause>, and <window clause> of *SC*, together with every <lateral derived table> that is simply contained in *FC* and is preceded by *TF*, and every <collection derived table> that is simply contained in *FC* and is preceded by *TF*, and every <XML iterate> that is simply contained in *FC* and is preceded by *TF*, and the <join condition> of all <joined table>s contained in *SC* that contain *TF*. If *SC* is the <query specification> that is the <query expression body> of a simple table query *STQ*, then the scope of the range variable of *TF* also includes the <order by clause> of *STQ*.

2) | Insert this SR | <XML iterate> shall not be specified.

NOTE nnn: <XML iterate> exists merely as a specification device; the user should use <XML table> instead.

3) | Insert this SR | The <derived column list> associated with <XML iterate> shall consist of two <column name>s, which shall not be equivalent.

[NOTE to the proposal reader: do we even need Syntax Rules about the proper syntax for non-invocable syntax?]

4) | Insert this SR | If <XML table> is specified, then:

a) Case:

i) If <XML namespace declaration> *XND* is specified, let *XNDC* be

```
WITH XND
```

ii) Otherwise, like *XNDC* be the zero-length string.

b) Let *XTRP* be the <XML table row pattern>.

c) Case:

i) If <XML table parameters> is specified, then let *NA* be the number of <XML query argument>s. Let XQA_i , 1 (one) <= *i* <= *N*, be an enumeration of the <XML query argument>s. Let *XQAL* be

```
PASSING BY REF XQA1, . . ., XQANA
```

ii) Otherwise, let *XQAL* be the zero-length string.

d) Let *NC* be the number of <XML table column definition>s. Let $XTCD_j$, 1 (one) <= *j* <= *NC*, be an enumeration of the <XML table column definition>s, in order from left to right.

e) For each *j* between 1 (one) and *N*, let CN_j be the <column name> contained in $XTCD_j$.

Case:

i) If $XTCD_j$ is an <XML table ordinality column definition>, then let SLI_j be

```
I.N AS CNj
```

ii) Otherwise:

2) Let DT_j be the <data type> contained in $XTCD_j$.

3) Case:

A) If DT_j is XML(SEQUENCE), then <XML passing mechanism> shall be specified. Let XPM_j be this <XML passing mechanism>.

B) Otherwise, <XML passing mechanism> shall not be specified. Let XPM_j be BY VALUE.

4) Case:

A) If DT_j is XML(UNTYPED DOCUMENT), XML(ANY DOCUMENT), XML(UNTYPED CONTENT) or XML(ANY CONTENT), then let XRM_j be RETURNING CONTENT.

B) Otherwise, let XRM_j be RETURNING XPM_j SEQUENCE.

5) If $XTCD_j$ contains a <default option>, let DEF_j be that <default option>; otherwise, let DEF_j be NULL.

6) If $XTCD_j$ contains an <XML table column pattern>, let $PATH_j$ be that <XML table column pattern>; otherwise, let $PATH_j$ be CN_j.

7) Let XQC_j be

```
XMLQUERY ( PATHj PASSING BY REF I.V XRMj )
```

8) Let XE_j be

```
XMLEXISTS ( PATHj PASSING BY REF I.V )
```

9) Let SLI_j be

```
CASE WHEN XE_J
     THEN CAST ( XQE_j AS DT_j )
     ELSE DEF_j
END
```

[NOTE to the proposal reader: Use the default clause if the column is missing, otherwise convert to the SQL type of the column.]

h) Let *CORR* be the <correlation name>.

i) Case:

1) If <derived column list> is specified, then, let *DCL* be that <derived column list>, and let *DCLP* be

```
( DCL )
```

2) Otherwise, let *DCLP* be the zero-length string.

j) The <XML table> is equivalent to

```
LATERAL
  ( XNDC
    SELECT SLI_1, SLI_2, . . . SLI_NC
    FROM XMLITERATE ( XMLQUERY ( XTRP XQAL
                                 RETURNING SEQUENCE BY REF ) )
         AS I ( V, N )
  ) AS R DCLP
```

Access Rules

No additional Access Rules

General Rules

1) | Insert after GR 2) | If a <table primary> *TP* simply contains an <XML iterate> then let *V* be the value of the <XML value expression>.

Case:

a) If *V* is the null value or the empty XQuery sequence, then the result of *TP* is a table with no rows.

b) Otherwise, the result of *TP* is a table consisting of one row for each XQuery item in *V*. The value of the first column of a row is the corresponding XQuery item of *V*. The value of the second column is

the sequential number of the XQuery item in the XQuery sequence *V* (1 for the first XQuery item in *V*, 2 for the second XQuery item, etc.).

Conformance Rules

1) Without Feature Xnnn, "XMLTable", conforming SQL language shall not contain <XML table>.

3.4 Changes to 8.1, <predicate>

1. EDIT THE FORMAT AS FOLLOWS:

```
<predicate> ::=
        !! All alternatives from ISO/IEC 9075-2
      | <XML document predicate>
      | <XML valid predicate>
      | <XML exists>
```

2. EDIT GENERAL RULE 1) AS FOLLOWS:

1) Replace GR 1) The result of a <predicate> is the truth value of the immediately contained <comparison predicate>, <between predicate>, <in predicate>, <like predicate>, <similar predicate>, <null predicate>, <quantified comparison predicate>, <exists predicate>, <unique predicate>, <match predicate>, <overlaps predicate>, <distinct predicate>, <member predicate>, <submultiset predicate>, <set predicate>, <type predicate>, <XML document predicate>, ~~or~~ <XML valid predicate>**, or <XML exists>**.

3.5 New Subclause 8.n, <XML exists>

1. ADD THE FOLLOWING SUBCLAUSE TO CLAUSE 8, "PREDICATES":

8.n <XML exists>

Function

Specify a test, whether an XQuery expression evaluates to one or more XQuery items.

Format

```
<XML exists> ::=
     XMLEXISTS <left paren> <XQuery expression>
     [ <XML query parameters> ] <right paren>
```

Syntax Rules

1) Let *XQE* be the <XQuery expression>.

2) If <XML query parameters> is specified, let *XQP* be that <XML query parameters>. Otherwise, let *XQP* be the zero-length string.

3) The Syntax Rules of Subclause 6.n, "<XML query>", are applied to

```
XMLQUERY ( XQE XQP RETURNING SEQUENCE )
```

Access Rules

1) The Access Rules of Subclause 6.n, "<XML query>", are applied to

```
XMLQUERY ( XQE XQP RETURNING SEQUENCE )
```

[NOTE to the proposal reader: there are no Access Rules for XMLQuery currently. This is just in case any are added in the furture.]

General Rules

1) Let *V* be the value of

```
XMLQUERY ( XQE XQP RETURNING SEQUENCE )
```

2) The value of <XML exists> is

Case:

a) If *V* is an XQuery sequence with one or more XQuery items, then *True*.

b) Otherwise, *False*.

Conformance Rules

1) Without Feature Xnnn, "XMLExists", Conforming SQL language shall not contain <XML exists>.

3.6 Changes to 24.3, Implied feature relationships

1. ADD THE FOLLOWING ROWS TO TABLE 13, "IMPLIED FEATURE RELATIONSHIPS OF SQL/XML":

Feature ID	Feature Name	Implied Feature ID	Implied Feature Name
Xnnn	**XMLTable**	**X010**	**XML type**

Feature ID	Feature Name	Implied Feature ID	Implied Feature Name
Xnnn	**XMLExists**	**X010**	**XML type**

3.7 Changes to Annex D, Incompatibilities

1. ADD THE FOLLOW TO THE LIST OF <RESERVED WORD>S:

— XMLTABLE

— XMLEXISTS

[NOTE to the proposal reader: We are not listing XMLITERATE as an incompatibility since it is not exposed to the user, whatever the decision may be about whether it is a <reserved word>.]

3.8 Changes to Annex E, SQL feature taxonomy

1. ADD THE FOLLOWING FEATURES TO TABLE 14, FEATURE TAXONOMY FOR OPTIONAL FEATURES

Feature ID	Feature Name
Xnnn	**XMLTable**
Xnnn	**XMLExists**

3.9 Changes to Editor's Notes

1. ADD THE FOLLOWING LANGUAGE OPPORTUNITY:

Severity: Language Opportunity

Reference: P14, SQL/XML, 2.2 "Other international standards"

Note at: following the BNF for XMLTable

Source: WG3:SIA-nnn

Possible Problem:

It would be desirable for XMLTable to have a LIKE clause, so that a result table could be modeled on an existing table.

4. Checklist

Concepts	done
Access Rules	done
Conformance Rules	done
Lists of SQL-statements by category	no new statements
Table of identifiers used by diagnostics statements	no new exceptions
Collation derivation for character strings	n/a
Closing Possible Problems	none
Any new Possible Problems clearly identified	one new LO
Reserved and non-reserved keywords	done
SQLSTATE tables and Ada package	no new exceptions
Information and Definition Schemas, including short-name views	no new descriptors
Implementation-defined and –dependent Annexes	none
Incompatibilities Annex	done
Embedded SQL and host language implications	n/a
Dynamic SQL issues: including descriptor areas	n/a
CLI issues	n/a

- End of paper -