XMLTABLE by example, Part 1: Retrieving XML data in relational format

Harness the power of this useful SQL/XML function in DB2

Matthias Nicola
Vitor Rodrigues

August 30, 2007

DB2® pureXML™ provides hierarchical storage for XML data and advanced XML query capabilities such as SQL/XML and XQuery. SQL/XML is a standardized extension of the SQL language so that XQuery or XPath can be embedded in SQL. This allows for integrated processing of XML and relational data, or conversion from one to the other. One of the most versatile functions in SQL/XML is XMLTABLE. In part 1 of this series, learn to use XMLTABLE to retrieve XML data in relational format. Discover how to manage repeating or missing XML elements, and how to handle namespaces in the XMLTABLE function. Part 2 describes common XMLTABLE usage scenarios such as XML shredding, splitting large documents into smaller ones, and relational views over XML data.

View more content in this series

XMLTABLE Overview

To understand this article, you should be familiar with the pureXML support in DB2 and with the basics of querying XML data in DB2. If you do not understand these topics, please see the Resources section of this article for a list of helpful articles on this subject.

XMLTABLE is an SQL/XML function that evaluates an XQuery expression and returns the result as a relational table. While XQuery expressions always return sequences of XML nodes, XMLTABLE returns this sequence as a set of rows in relational format. The returned table can contain columns of any SQL type, including the XML type.

Figure 1. XMLTABLE overview
Like any SQL/XML function, XMLTABLE is embedded in an SQL statement. The evaluation of an XMLTABLE function returns a rowset where each column has an SQL data type. This means it is a table function, not a scalar function.

To learn about the XMLTABLE function in more detail, view the following sample table, which contains two rows with one XML document per row:

Listing 1 is an example of a simple XMLTABLE statement.

**Listing 1. Sample table and data**

```xml
<dept bldg="101">
  <employee id="901">
    <name>
      <first>John</first>
      <last>Doe</last>
    </name>
    <office>344</office>
    <salary currency="USD">55000</salary>
  </employee>
  <employee id="902">
    <name>
      <first>Peter</first>
      <last>Pan</last>
    </name>
    <office>216</office>
    <phone>905-416-5004</phone>
  </employee>
</dept>

<dept bldg="114">
  <employee id="903">
    <name>
      <first>Mary</first>
      <last>Jones</last>
    </name>
    <office>415</office>
    <phone>905-403-6112</phone>
    <phone>647-504-4546</phone>
    <salary currency="USD">64000</salary>
  </employee>
</dept>
```

Listing 2 is an example of a simple XMLTABLE statement.

**Listing 2. A simple XMLTABLE example**

```sql
SELECT X.*
FROM emp,
XMLTABLE ('$d/dept/employee' passing doc as "d"
COLUMNS
  empID INTEGER PATH '@id',
  firstname VARCHAR(20) PATH 'name/first',
  lastname VARCHAR(25) PATH 'name/last') AS X
```

Running this query in DB2 returns the following result:
Curious about how that works? The XMLTABLE function is used in the *FROM* clause of the *SELECT* statement together with the table *emp* that it operates on. The XMLTABLE function is implicitly joined with the table *emp* and applied to each of its rows.

The XMLTABLE function contains one *row-generating* XQuery expression and, in the COLUMNS clause, one or multiple *column-generating* expressions. In Listing 1, the row-generating expression is the XPath `$d/dept/employee`. The *passing* clause defines that the variable `$d` refers to the XML column *doc* of the table *emp*.

The row-generating expression is applied to each XML document in the XML column and produces one or multiple *employee* elements (sub-trees) per document. The output of the XMLTABLE function contains one row for each *employee* element. Hence, the output produced by the row-generating XQuery expression determines the cardinality of the result set of the SELECT statement.

The COLUMNS clause is used to transform XML data into relational data. Each of the entries in this clause defines a column with a column name and a SQL data type. In the example above, the returned rows have 3 columns named *empID*, *firstname* and *lastname* of data type Integer, Varchar(20) and Varchar(25), respectively. The values for each column are extracted from the *employee* elements, which are produced by the row-generating XQuery expression, and cast to the SQL data types. For example, the path *name/first* is applied to each *employee* element to obtain the value for the column firstname. The row-generating expression provides the context for the column-generating expressions. In other words, you can typically append the column-generating expressions to the row-generating expression to get an intuitive idea of what a given XMLTABLE function returns in its columns.

Be aware that the path expressions in the COLUMNS clause must not return more than one item per row. If a path expression returns a sequence of two or more items, the XMLTABLE execution will typically fail, as it is not possible to convert a sequence of XML values into an atomic SQL value. This scenario is discussed later in the article.

The result set of the XMLTABLE query can be treated like any SQL table. You can query and manipulate it much like you use regular rowsets or views. Instead of using the "passing column as" clause, you can specify the data input to the XMLTABLE function with the *db2-fn:xmlcolumn()* or *db2-fn:sqlquery()* functions (DB2 LUW only). For example, Listing 2 above can also be written as shown in Listing 3 to produce the same result.
XMLTABLE by example, Part 1: Retrieving XML data in relational format

### Listing 3. A different notation for Listing 1:

```sql
SELECT X.*
FROM
  XMLTABLE ('db2-fn:xmlcolumn("EMP.DOC")/dept/employee'
  COLUMNS
    empID INTEGER PATH '@id',
    firstname VARCHAR(20) PATH 'name/first',
    lastname VARCHAR(25) PATH 'name/last') AS X
```

### Missing elements

XML data can contain optional elements that are not present in all of your documents. For example, in Listing 1 employee Peter Pan does not have a salary element since it's not a required data field in our sample scenario. It's easy to deal with that because the XMLTABLE function simply produces null values for missing elements, so you can write XMLTABLE queries as if the salary element was always present. Listing 4 illustrates this:

### Listing 4. An extension of Listing 2 to also produce a salary column

```sql
SELECT X.*
FROM emp,
  XMLTABLE ('$d/dept/employee' passing doc as "d"
  COLUMNS
    empID INTEGER PATH '@id',
    firstname VARCHAR(20) PATH 'name/first',
    lastname VARCHAR(25) PATH 'name/last',
    salary INTEGER PATH 'salary') AS X
```

This query returns the following result where the salary column in the returned relational table has a null value for employee Peter Pan:

<table>
<thead>
<tr>
<th>empID</th>
<th>firstname</th>
<th>lastname</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>901</td>
<td>John</td>
<td>Doe</td>
<td>55000</td>
</tr>
<tr>
<td>902</td>
<td>Peter</td>
<td>Pan</td>
<td>-</td>
</tr>
<tr>
<td>903</td>
<td>Mary</td>
<td>Jones</td>
<td>64000</td>
</tr>
</tbody>
</table>

If you want a value other than "null" to appear for missing elements, such as the number zero, you can define a default value that is returned in case the expected element is missing. This is shown in Listing 5, which returns "0" as the salary for Peter Pan. Note that the default value must match the target data type of the column. Since "salary" is mapped to an integer column, the default value must be an integer.

### Listing 5. Using a default value in the column-generating expression for salary

```sql
SELECT X.*
FROM emp,
  XMLTABLE ('$d/dept/employee' passing doc as "d"
  COLUMNS
    empID INTEGER PATH '@id',
    firstname VARCHAR(20) PATH 'name/first',
    lastname VARCHAR(25) PATH 'name/last',
    salary INTEGER default 0 PATH 'salary') AS X
```

XMLTABLE by example, Part 1: Retrieving XML data in relational format
Generating rows for a subset of the data

Often you want to produce rows only for a subset of the employees based on some filtering predicate. An easy solution is to add a WHERE clause with an XMLEXISTS predicate to your query. (See the Resources section of this document for articles related to this topic). Another solution is to use filtering predicates in the row-generating expression of the XMLTABLE function. Say you need to produce rows only for employees in building 114. You can add a corresponding predicate to any of the queries above which returns only a single row for Mary Jones, who is the only employee in building 114. Listing 6 illustrates adding a row-filtering predicate to Listing 2.

Listing 6. Adding a row-filtering predicate to Listing 1

```sql
SELECT X.*
FROM emp,
XMLTABLE ('$d/dept[@bldg="114"]/employee' passing doc as "d"
COLUMNS
    empID INTEGER PATH '@id',
    firstname VARCHAR(20) PATH 'name/first',
    lastname VARCHAR(25) PATH 'name/last',
    salary INTEGER default 0 PATH 'salary') AS X
```

Handling multiple values per cell

As mentioned earlier, the path expressions in the COLUMNS clause must not produce more than one item per row. In the sample documents in Listing 1, notice that the employee Mary Jones has two phone numbers. If you need to query this data and return a relational table with each employee’s name and phone number, the query you would write might look like this:

Listing 7. Extracting XML data into relational

```sql
SELECT X.* FROM emp,
XMLTABLE ('$d/dept/employee' passing doc as "d"
COLUMNS
    first VARCHAR(25) PATH 'name/first',
    last VARCHAR(25) PATH 'name/last',
    phone VARCHAR(12) PATH 'phone') AS X
```

For the sample documents that are used in this article, this query fails; it produces the following error message:

SQL16003N An expression of data type "( item(), item()+ )" cannot be used when the data type "VARCHAR_12" is expected in the context.

This message means that the query is trying to cast an XML sequence of multiple items to a single Varchar value. A value of data type "(item(), item()+)" means the value is an item followed by one or more additional items. In simpler terms, this means that the value is a sequence of two or more items. This happens because the path expression "phone" returns two phone elements for the employee Mary Jones.

Next, the article describes five options that help you avoid receiving this error:

- Return only one of multiple phone numbers
• Return a list of multiple phone numbers in a single Varchar value
• Return a list of multiple phone numbers as an XML type
• Return multiple phone columns
• Return one row per phone number

Each of these options has its benefits, so you can decide which one to use based on your needs.

**Return only one of multiple elements**

One way to deal with this issue is to return only one of the multiple phone numbers. If you need summarized information for each employee, having just one phone number may be enough. Returning only one occurrence of the phone element can be done with a positional predicate in the XPath expression for the column phone, as shown in Listing 8:

**Listing 8. Returning the first occurrence of phone element for each employee**

```sql
SELECT X.* FROM emp,
    XMLTABLE ('$d/dept/employee' passing doc as "d"
    COLUMNS
        first VARCHAR(25) PATH 'name/first',
        last  VARCHAR(25) PATH 'name/last',
        phone VARCHAR(12) PATH 'phone[1]' ) AS X
```

Square brackets [] in XPath are used to specify predicates. To obtain the first phone element for an employee, use a positional predicate, written either as [1] or [fn:position()=1]. The former notation [1] is an abbreviated version of the latter. Listing 7 returns the following result set:

<table>
<thead>
<tr>
<th>first</th>
<th>last</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Peter</td>
<td>Pan</td>
<td>905-416-5004</td>
</tr>
<tr>
<td>Mary</td>
<td>Jones</td>
<td>905-403-6112</td>
</tr>
</tbody>
</table>

3 record(s) selected.

**Return a list of multiple values in a single Varchar**

If you need to return all phone numbers, you can list them within a single column. Since VARCHAR(12) is too small for multiple phone numbers, the SQL type for the returned column needs to be changed. Use VARCHAR(100) here, which allows you to produce multiple phone numbers separated by a comma, as in Listing 9:

**Listing 9. Listing all the phone numbers from a single employee**

```sql
SELECT X.* FROM emp,
    XMLTABLE ('$d/dept/employee' passing doc as "d"
    COLUMNS
        first VARCHAR(25) PATH 'name/first',
        last  VARCHAR(25) PATH 'name/last',
        phone VARCHAR(100) PATH 'fn:string-join(phone/text(),",")' ) AS X
```

For the sample data, this query returns:
The phone column contains the two phone numbers listed for employee Mary Jones. The function `fn:string-join` links these values and requires two parameters: a sequence of string values and a separator character. In this example, the two parameters are the sequence of the phone elements' text nodes and the character ",".

### Return multiple elements as an XML sequence

Another option to return multiple phone numbers for a single employee is to return an XML sequence of phone elements, as Listing 10 illustrates. To achieve this, the generated phone column needs to be of type XML, which allows you to return any XML value as the result of the XPath expression. This value can be an atomic value or a sequence.

#### Listing 10. Returning all the phone elements as an XML sequence

```sql
SELECT X.* FROM emp ,
XMLTABLE ('$d/dept/employee' passing doc as "d"
COLUMNS
  first VARCHAR(5) PATH 'name/first',
  last VARCHAR(5) PATH 'name/last',
  phone xml PATH 'phone'
) AS X
```

This query returns one row per employee with their phone numbers in an XML sequence in the XML column `phone`:

<table>
<thead>
<tr>
<th>first</th>
<th>last</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Peter</td>
<td>Pan</td>
<td>&lt;phone&gt;905-416-5004&lt;/phone&gt;</td>
</tr>
<tr>
<td>Mary</td>
<td>Jones</td>
<td>&lt;phone&gt;905-403-6112&lt;/phone&gt;&lt;phone&gt;647-504-4546&lt;/phone&gt;</td>
</tr>
</tbody>
</table>

The XML value returned in the phone column for Mary Jones is not a well-formed XML document since there is no single root element. This value can still be processed in DB2, but you won't be able to insert it into an XML column or parse it with an XML parser in your application. If you need to produce well-formed XML documents, you can wrap the sequence of phone elements in new root element, for example, by changing the path expression in the columns clause to `<phones>{phone}</phones>`.

### Return multiple phone columns

Combining multiple phone numbers into a single Varchar or XML value may require additional code in your application to use the individual numbers. If you prefer to return each phone number as a separate Varchar value, you can do this by producing a fixed number of phone columns. Listing 11 uses positional predicates to return phone numbers in two columns:
Listing 11. Returning multiple phone columns

```sql
SELECT X.* FROM emp ,
XMLTABLE ('$d/dept/employee' passing doc as "d"
  COLUMNS
    first  VARCHAR(25) PATH  'name/first',
    last   VARCHAR(25) PATH  'name/last',
    phone1 VARCHAR(12) PATH  'phone[1]',
    phone2 VARCHAR(12) PATH  'phone[2]' ) AS X
```

The output for the query in Listing 10 is:

<table>
<thead>
<tr>
<th>first</th>
<th>last</th>
<th>phone</th>
<th>phone2</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Doe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peter</td>
<td>Pan</td>
<td>905-416-5004</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>Jones</td>
<td>905-403-6112</td>
<td>647-504-4546</td>
</tr>
</tbody>
</table>

3 record(s) selected.

An obvious drawback to this approach is that a variable number of items is being mapped to a fixed number of columns. An employee may have more phone numbers than anticipated. Others may have less which results in null values. But, if every employee has exactly one office phone and one cell phone, then producing two columns with corresponding names is very useful.

Return one row per phone number

Instead of returning the phone numbers in separate columns, you can also use XMLTABLE to return them in separate rows. In this case, you need to return one row per phone number instead of one row per employee. This may result in repeated information in the columns for the first and last names. Listing 12 shows what happens when you change the row-generating XPath expression in the XMLTABLE function to create a relational row per phone number.

Listing 12. Producing one row per phone number

```sql
SELECT X.* FROM emp ,
XMLTABLE ('$d/dept/employee/phone' passing doc as "d"
  COLUMNS
    first  VARCHAR(5) PATH  '../name/first',
    last   VARCHAR(5) PATH  '../name/last',
    phone  VARCHAR(12) PATH  '.' ) AS X
```

As compared to the previous queries, Listing 11 uses different XPath both in the row-generating and column-generating expressions. As the context is now a phone element and not an employee element, the XPath expressions in the COLUMNS clause have changed accordingly. The paths for first and last name begin with a parent step because name is a sibling of phone. The result of this query contains two relational rows for employee Mary Jones, each with one of her phone numbers:

<table>
<thead>
<tr>
<th>first</th>
<th>last</th>
<th>phone</th>
<th>phone2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>Pan</td>
<td>905-416-5004</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>Jones</td>
<td>905-403-6112</td>
<td>647-504-4546</td>
</tr>
</tbody>
</table>

3 record(s) selected.
Non-existent paths

You probably wonder why Listing 11 didn't return a row for employee John Doe. What happened is that the row-generating expression in Listing 11 iterates over all the phone elements in the documents and there is no phone element for the employee John Doe. As a result, the employee element for John Doe is never processed. If unnoticed, this could be a problem, creating an incomplete employee list.

To avoid this, a row must be generated for each employee even if the employee does not have a phone element. A possible solution is to produce a row for the name element (a sibling of phone) whenever a phone element is not present. Do this with the following comma-separated list of expressions: '(phone, .[fn:not(phone)]/name)'. The results of both expressions are combined into a single sequence. But, in this case, one of the two expressions always produces an empty result. If an employee has one or more phone elements, this produces all of these phone elements. If the document has no phone element, then and only then, the name element is returned. The fn:not function avoids duplicate rows for those employees which have a name and a phone. This is shown in Listing 13:

Listing 13. Producing rows for phone numbers or names

```sql
SELECT X.* FROM emp ,
XMLTABLE ('$d/dept/employee/(phone,.[fn:not(phone)]/name)' passing doc as "d"
COLUMNS
  first VARCHAR(5) PATH ' ../name/first',
  last  VARCHAR(5) PATH ' ../name/last',
  phone VARCHAR(12) PATH '.[../phone]' ) AS X
```

Listing 12 generates rows not only for phone elements, but also for name elements when no phone element is present. If an employee has several phone numbers, the query returns one row per phone number. If an employee has no phone number, it returns only one row for that employee, without phone information:

<table>
<thead>
<tr>
<th>first</th>
<th>last</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Doe</td>
<td>-</td>
</tr>
<tr>
<td>Peter</td>
<td>Pan</td>
<td>905-416-5004</td>
</tr>
<tr>
<td>Mary</td>
<td>Jones</td>
<td>905-403-6112</td>
</tr>
<tr>
<td>Mary</td>
<td>Jones</td>
<td>647-504-4546</td>
</tr>
</tbody>
</table>

4 record(s) selected.

XMLTABLE with Namespaces

XML namespaces are a W3C XML standard for providing uniquely named elements and attributes in an XML document. XML documents may contain elements and attributes from different vocabularies but have the same name. By giving a namespace to each vocabulary, the ambiguity is resolved between identical element or attribute names. All pureXML features in DB2 9 support XML namespaces, such as SQL/XML, XQuery, XML indexes, and XML schema handling. For more information on querying XML data with namespaces, see Resources.
In XML documents, you declare XML namespaces with the reserved attribute `xmlns`, whose value must contain an Universal Resource Identifier (URI). URIs are used as identifiers; they typically look like a URL but they don't have to point to an existing web page. A namespace declaration can also contain a prefix, used to identify elements and attributes. Below is an example of a namespace declaration with and without prefix:

```xml
xmlns:ibm = "http://www.ibm.com/xmltable/"
xmlns = "http://www.ibm.com/xmltable/"
```

To demonstrate the use of namespaces with XMLTABLE, a sample document is added to Listing 1. Unlike the other documents introduced in Listing 1 at the beginning of this article, this new document contains a namespace declaration with the prefix `ibm`:

**Listing 14. Sample document containing a namespace declaration with a prefix**

```xml
  <ibm:employee id="144">
    <ibm:name>
      <ibm:first>James</ibm:first>
      <ibm:last>Bond</ibm:last>
    </ibm:name>
    <ibm:office>007</ibm:office>
    <ibm:phone>905-007-1007</ibm:phone>
    <ibm:salary currency="USD">77007</ibm:salary>
  </ibm:employee>
</ibm:dept>
```

Execute the query in Listing 1 again and check the output.

Executing this query in DB2 produces the following output:

<table>
<thead>
<tr>
<th>empID</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>901</td>
<td>John</td>
<td>Doe</td>
</tr>
<tr>
<td>902</td>
<td>Peter</td>
<td>Pan</td>
</tr>
<tr>
<td>903</td>
<td>Mary</td>
<td>Jones</td>
</tr>
</tbody>
</table>

As you can see, the information about employee James Bond is not returned. The reason is that Listing 2 references only the element names that have no namespace. In order to return all the employees in the database, you can use the * wildcard for the namespace prefix in the path expressions in Listing 15. This causes all elements to be considered, regardless of namespaces, because this wildcard (*) matches any namespace including no namespace.

**Listing 15. Using a wildcard (*) to match all namespaces**

```sql
SELECT X.*
FROM emp,
XMLTABLE ('$d/*:dept/*:employee' passing doc as "d"
  COLUMNS empID INTEGER PATH '::*:@id',
  firstname VARCHAR(20) PATH '::*:name/*:first',
  lastname VARCHAR(25) PATH '::*:name/*:last') AS X
```

XMLTABLE by example, Part 1: Retrieving XML data in relational format
As a result of using wildcards to match all namespaces in the documents, all employees are returned:

<table>
<thead>
<tr>
<th>empID</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>901</td>
<td>John</td>
<td>Doe</td>
</tr>
<tr>
<td>902</td>
<td>Peter</td>
<td>Pan</td>
</tr>
<tr>
<td>903</td>
<td>Mary</td>
<td>Jones</td>
</tr>
<tr>
<td>144</td>
<td>James</td>
<td>Bond</td>
</tr>
</tbody>
</table>

4 record(s) selected.

For this specific data, the namespace wildcard for the attribute @id was not strictly necessary. The reason is that the @id attribute employee James Bond has no namespace. Attributes never inherit namespaces from their element and also never assume the default namespace. So, unless the attribute name has a prefix, it doesn't belong to any namespace.

The use of the wildcard expression is the simplest way to return all employees, regardless of namespace. Next, see how you can return only the information for the employees in the ibm namespace. There are two ways to specify the namespace in an XQuery or XPath expression; this can be done by:

- declaring a default namespace
- declaring a namespace prefix

### Declaring a default namespace

When all the elements you want to query belong to the same namespace, declaring a default namespace can be the simplest way to write your queries. You just need to declare the default namespace in the beginning of your XQuery expression and, after that, all elements and attribute names you reference are tied to that namespace. This is shown in Listing 16:

**Listing 16. Using default namespace declaration**

```xml
SELECT X.*
FROM emp,
XMLTABLE ('declare default element namespace "http://www.ibm.com/xmltable";
$doc/dept/employee' passing doc as "d"
COLUMNS
  empID INTEGER PATH '@id',
  firstname VARCHAR(20) PATH 'declare default element namespace "http://www.ibm.com/xmltable"; name/first',
```

Using the namespace declarations allows you to filter the employees from the namespace ibm.

The output from Listing 15 is the following:

<table>
<thead>
<tr>
<th>EMPID</th>
<th>FIRSTNAME</th>
<th>LASTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>James</td>
<td>Bond</td>
</tr>
</tbody>
</table>

1 record(s) selected.
Please note that the **column-generating** expressions do not inherit the namespace declaration from the **row-generating** expression. Each column-generating expression is a separate XQuery and needs its own namespace declaration. These namespace declarations may differ from each other, for example, if your document contains multiple namespace. Often there is only one namespace; in which case, it would be convenient to declare a single namespace for all expressions in the XMLTABLE function. This can be achieved by using the function XMLNAMESPACES(). This function allows you to declare a default namespace and/or several namespace prefixes inside XMLTABLE and other SQL/XML functions. The advantage of using the XMLNAMESPACES function is that the declared namespaces are global for all expressions in the XMLTABLE context, so all the XQuery expressions will be aware of these namespaces declarations and repeated namespace declarations are not required.

Let's re-write **Listing 16** using the XMLNAMESPACES() function:

**Listing 17. Using XMLNAMESPACES() to declare the default namespace**

```sql
SELECT X.*
FROM emp,
XMLTABLE (XMLNAMESPACES(DEFAULT 'http://www.ibm.com/xmltable'),
'@d/ dept/ employee' passing doc as "d"
COLUMNS
empID INTEGER PATH '@id',
firstname VARCHAR(20) PATH 'name/first',
lastname VARCHAR(25) PATH 'name/last') AS X
```

The default namespace declared by the XMLNAMESPACES() function applies to both the **row-generating** expression and all the **column-generating** expressions. This way only one namespace declaration is needed for all XQuery expressions in an XMLTABLE() function. The result of **Listing 17** is exactly the same as for **Listing 16**.

**Declaring a namespace prefix**

While a default namespace is a common solution when only one namespace is present in your documents, you need a different approach if your documents contain multiple namespaces. Using a default namespace only allows you to select elements and attributes from that namespace, and using a wildcard selects elements and attributes from all namespaces. If you want to select elements and attributes from multiple specific namespaces, then using namespace prefixes is your best option.

Unless you use the XMLNAMESPACES function, the namespaces prefixes need to be declared for every expression. But, just like for default namespaces, you can use the XMLNAMESPACES function to avoid repeated namespace declarations. **Listing 18** shows how to declare a namespace prefix in the XMLTABLE function.
Listing 18. Using the XMLNAMESPACES() function to declare a namespace prefix

```
SELECT X.*
FROM emp,
XMLTABLE (XMLNAMESPACES('http://www.ibm.com/xmltable' as "ibm"),
'$/ibm:dept/ibm:employee' passing doc as "d"
COLUMNS
empID  INTEGER  PATH '@id',
firstname  VARCHAR(20)  PATH 'ibm:name/ibm:first',
lastname  VARCHAR(25)  PATH 'ibm:name/ibm:last') AS X
```

As expected, Listing 17 returns the same result as Listing 16:

<table>
<thead>
<tr>
<th>EMPID</th>
<th>FIRSTNAME</th>
<th>LASTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>James</td>
<td>Bond</td>
</tr>
</tbody>
</table>

1 record(s) selected.

Summary

In this first part of our 2-part series on XMLTABLE, you have learned how to use XMLTABLE to retrieve XML data in relational format, how to deal with repeating or missing XML elements and non-existing paths, and how to handle namespaces in the XMLTABLE function. This gives you a powerful range of capabilities for querying your XML data in DB2 LUW and DB2/zOS. In part 2 of this series, learn about common XMLTABLE usage scenarios such as shredding XML into relational tables, splitting large documents into smaller ones, and relational views over XML data.

Acknowledgements

Thanks to Cindy Saracco for her review and helpful comments on this paper.
Related topics

- pureXML in DB2 9: Which way to query your XML data?
- Query XML data that contains namespaces
- Native XML Support in DB2 Universal Database
- Series: Get off to a fast start with DB2 9 pureXML

© Copyright IBM Corporation 2007
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)