DB2 9 Fundamentals exam 730 prep, Part 7: Introducing XQuery

Skill Level: Introductory

C. M. Saracco (saracco@us.ibm.com)
Senior Software Engineer
IBM

24 Jul 2006

DB2® 9 features support for XQuery, an industry-standard language designed expressly for querying XML data. With XQuery and DB2 9 you can retrieve entire XML documents or XML fragments stored in XML columns. You can also specify XML-based filters for queries, transform XML output, and incorporate conditional logic into queries. This tutorial introduces you to DB2's support of XQuery, explains several basic language concepts, and shows how you can write and execute simple XQueries against XML data stored in DB2. This is the seventh in a series of seven tutorials to help you prepare for the DB2 9 Fundamentals exam 730.

Section 1. Before you start

To help you prepare for the DB2 certification exams, this tutorial introduces you to XQuery and its support in DB2. You should already be familiar with DB2 V9 and its pureXML support before taking this tutorial.

This tutorial focuses on using XQuery to query DB2 XML data. It provides very limited discussion on using SQL/XML (SQL with XML extensions) to query DB2 XML data. For more information on DB2's support for industry-standard SQL/XML functions, see Resources.

About this series

Thinking about seeking certification on DB2 fundamentals (Exam 730)? If so, you've come to the right place. This series of seven DB2 certification preparation tutorials covers all the basics -- the topics you'll need to understand before you read the first exam question. Even if you're not planning to seek certification right away, this set of tutorials is a great place to start learning what's new in DB2 9.

About this tutorial

This tutorial explores basic capabilities of DB2's new XQuery support. It reviews key differences between XQuery and SQL, explores XPath and "FLWOR" expressions, and teaches you how to write simple XQueries over DB2 XML data.

This tutorial is for DB2 users who plan to work with XML documents stored in their native hierarchical structures within XML columns of DB2 tables. The material in this tutorial covers XML topics that are addressed in Sections 1, 4, and 5 of the test. You can view these objectives at: http://www-03.ibm.com/certify/tests/obj730.shtml. You should be familiar with basic XML technologies and with DB2's new pureXML support before taking this tutorial. If necessary, consult the Resources for background material.

Objectives

After completing this tutorial, you should be able to:

- Understand fundamental XQuery concepts
- Write simple XQueries using several common expressions

Prerequisites

This tutorial is for people familiar with basic XML technology and DB2's new support for pureXML storage and data management. You should understand the hierarchical nature of XML documents, the concept of well-formedness, and how elements and attributes may be used. You should also understand how you can store well-formed XML documents in their native hierarchical structure using DB2 9.

System requirements

You do not need a copy of DB2 to complete this tutorial, but to run the examples you need a system on which DB2 9 is installed. Any supported platform will be sufficient, but some of the examples are tailored towards the Windows platforms. You will get more out of the tutorial if you download the free trial version of IBM DB2 9 to work along with this tutorial.

Section 2. XQuery overview

DB2 9 introduces support for XQuery, a new query language designed specifically to work with XML data. Part of the W3C industry standard, XQuery lets users navigate through the hierarchical structure inherent in XML documents. As a result, you can retrieve XML documents or document fragments using XQuery. You can also write XQueries that include XML-based predicates to "filter out" unwanted data from the results that DB2 will return. XQuery offers many other capabilities, such as the ability to transform XML output and incorporate conditional logic into your queries.

Before learning how to use XQuery, you need to understand some fundamental concepts about the language.

XQuery fundamentals

An XQuery always transforms one value of the XQuery Data Model into another value of the XQuery Data Model. A value in the XQuery Data Model is a sequence of zero or more items. An item can be:

- Any atomic value
- An XML node such as an element, attribute, or text node (sometimes called an XML document fragment)
- A full XML document

Often, the input to an XQuery is a collection of XML documents.

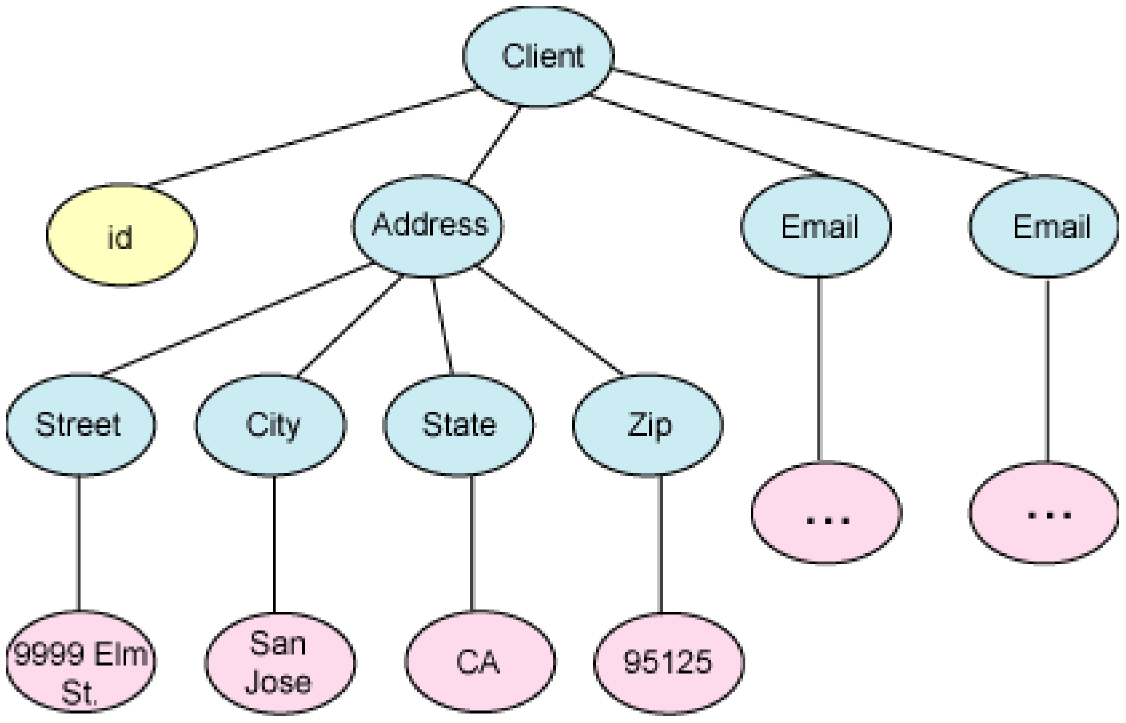
Listing 1 shows an XML document containing eight element nodes, one attribute node, and six text nodes. The element nodes are represented by element tags. Client, Address, street, city, state, zip, and both email elements are all element nodes in this document. If you look closely at the Client element, you'll see that it contains an attribute node for the client's `id`. Some of the document's element nodes have text nodes associated with them. For example, the text node for the city element is San Jose.

Listing 1. Sample XML document

```
<Client id="123">
        <Address>
                <street>9999 Elm St.</street>
                <city>San Jose</city>
                <state>CA</state>
                <zip>95141</zip>
        </Address>
        <email>anyemail@yahoo.com</email>
        <email>anotheremail@yahoo.com</email>
</Client>
```

Figure 1 shows the nodes within this sample document.

Figure 1. Element, attribute, and text nodes in a sample XML document



Blue = element nodes; Yellow = attribute node; Pink = text nodes

The XQuery language draws from other XML standards such as XPath, which defines how users can navigate through an XML document, and XML Schema, which lets users specify the valid structure and data types for their documents. You'll learn how to incorporate XPath expressions into XQueries in this tutorial.

XQuery provides several different kinds of expressions that you can combine in any way you might like. Each expression returns a sequence of values that can be used as input to other expressions. The result of the outermost expression is the result of the query.

This tutorial discusses two important kinds of XQuery expressions:

Path expression
: Enables users to navigate, or "walk through," the hierarchy of an XML document and return nodes found at the end of the path.

FLWOR expression
: Is much like a `SELECT-FROM-WHERE` expression in SQL . It is used to iterate through a sequence of items and to optionally return something that is computed from each item.

How XQuery differs from SQL

Many SQL users mistakenly assume that XQuery is very similar to SQL. However, XQuery differs from SQL in many ways largely because the languages were designed to work with different data models that have different characteristics. XML documents contain hierarchies and possess an inherent order. By contrast, tables supported by relational DBMSs (or, more precisely, SQL-based DBMSs) are flat and set-based, so rows are unordered.

These differences in data models result in significant differences in the query languages designed to support each of them. For example, XQuery lets programmers navigate through XML's hierarchical structure. Plain SQL (without XML extensions) does not have -- or need -- equivalent expressions to "navigate" through tabular data structures. XQuery supports both typed and untyped data, while SQL data is always defined with a specific type.

> These are just a few of the differences between XQuery and SQL. It is beyond the scope of this introductory tutorial to provide an exhaustive list, but Resources has more on this topic.

XQuery lacks null values because XML documents omit missing or unknown data. SQL uses nulls to represent missing or unknown data values. XQuery returns sequences of XML data; SQL returns result sets of various SQL data types. Finally, XQuery operates only on XML data. SQL operates on columns defined on traditional SQL types, and SQL/XML (SQL with XML extensions) operates on both XML data and traditional types of SQL data.

Path expressions in XQuery

XQuery supports XPath expressions that allow users to navigate through an XML document hierarchy to locate portions of interest. It's beyond the scope of this tutorial to discuss XPath in detail, but we'll review a few simple examples here.

XPath expressions look very much like the expressions you use when working with a traditional computer file system. Consider how you navigate through Unix or Windows directories, and you'll get an idea of how you can navigate through an XML document using XPath.

A path expression in XQuery consists of a series of "steps" separated by slash characters. In its simplest form, each step navigates downward in an XML hierarchy to find the children of the elements returned by the previous step. Each step in a path expression may also contain a predicate that filters the elements that are returned by that step, retaining only elements that satisfy some condition. You'll see an example of that in just a moment.

A common task involves navigating from the XML document root (the top level in the XML hierarchy) to a particular node of interest. For example, to retrieve the email elements in the document shown in Listing 2 below, you could write:

Listing 2. Navigating to the email elements

```
/Client/email
```

If the document contained multiple email elements and you only wanted to retrieve the first one, you could write:

Listing 3. Navigating to the first email element

```
/Client/email[1]
```

In addition to specifying element nodes in your path expressions, you can specify attribute nodes using the @ sign to distinguish an attribute from an element. This path expression navigates to the first email element for the Client element that contains an id attribute equal to 123:

Listing 4. Specifying an attribute node and value

```
/Client[@id='123']/email[1]
```

The previous example incorporated a filtering predicate based on an attribute value. You can also filter on other node values. Quite often, XPath users filter on element values, as shown in this expression that returns the zip element for all clients who live in California:

Listing 5. Filtering on an element value

```
/Client/Address[state="CA"]/zip
```

You can use a wildcard ("*") to match any node at the respective step of your path expression. The following example retrieves any city element found beneath any immediate child of the Client element.

Listing 6. Using a wildcard

```
/Client/*/city
```

Given our sample document, this will return one city element with a value of San Jose. A more precise way to navigate to this single city element would be:

Listing 7. A more precise way to navigate to the city element

```
/Client/Address/city
```

Listing 8 shows a few examples of other types of path expressions.

Listing 8. More path expressions and their meanings

```
//*                        (Retrieves all nodes in the document)

//email                    (Finds email elements anywhere in the document)

/Client/email[1]/text()    (Retrieves the text node of the first email element
                                     beneath the Client element)
```

```
/Client/Address/*          (Selects all child nodes of the Address sub-element of
                                      root Client element)

/Client/data(@id)          (Returns the value of the id attribute of the Client
                                      element)

/Client/Address[state="CA"]/../email    (Finds the email elements of clients with an
                                       address in California.
                              The ".." step navigates back to the parent of the Address
                               node.)
```

Note that XPath is *case sensitive*. This is important to remember when writing XQueries, as it represents another way in which XQuery differs from SQL. For example, if you incorporated the path expression "/client/address" into your XQuery, you would not receive any results for the sample document shown in Listing 1.

FLWOR expressions in XQuery

People often refer to FLWOR expressions in XQuery. Like a `SELECT-FROM-WHERE` block in SQL, an XQuery FLWOR expression may contain multiple clauses denoted by keywords. The clauses of a FLWOR expression begin with the following keywords:

- `for`: Iterates through an input sequence, binding a variable to each input item in turn
- `let`: Declares a variable and assigns it a value, which may be a list containing multiple items
- `where`: Specifies criteria for filtering query results
- `order by`: Specifies the sort order of the result
- `return`: Defines the result to be returned

Let's review each keyword briefly. We'll discuss `for` and `return` in one section so you can see a complete example. (Without the `return` clause, the expression would be incomplete.)

`for` and `return`

The `for` and `return` keywords are used to iterate over a sequence of values and to return something for each value. The following is a very simple example:

```
for $i in (1, 2, 3)
return $i
```

In XQuery, variable names are prefixed with a dollar sign ("$"). So the previous example binds the numeric values 1, 2, and 3 to the variable $i, one at a time, and for each binding returns the value of $i. The output from the previous expression is a sequence of three values:

```
1
2
3
```

`let`

People sometimes have a hard time distinguishing when to use the `let` keyword rather than `for`. The `let` keyword does not iterate through a sequence of input, binding each item to a variable in turn (as the `for` keyword does). Instead, `let` assigns a single input value to a variable, but this input value may be a sequence of zero, one, or more items. As a result, `for` and `let` behave quite differently in XQueries.
An example should help clarify the distinction. Consider the following expression that uses the `for` keyword and pay close attention to the output returned:

```
for $i in (1, 2, 3)
return <output> {$i} </output>


<output>1</output>
<output>2</output>
<output>3</output>
```

The final line of the expression causes a new element named `output` to be returned for each iteration. The value for this element is the value of $i. Because $i is set to the numeric values of 1, 2, and 3 one at time, the XQuery expression returns three output elements, each with a different value.

Now consider a similar expression that uses the `let` keyword:

```
let $i := (1, 2, 3)
return <output>{$i}</output>


<output>1 2 3</output>
```

The output is quite different. It consists of a single output element with the value of "1 2 3."

These two examples illustrate an important point to remember: the `for` keyword iterates through items in an input sequence one at a time, binding each to the specified variable in turn. By contrast, the `let` keyword binds all the items in the input sequence at once to the specified variable.

`where`

In XQuery, `where` functions much like the WHERE clause in SQL: it enables you to apply filtering criteria to your query. Consider the following example:

```
for $i in (1, 2, 3)
where $i < 3
return <output>{$i}</output>
```

```
<output>1</output>
<output>2</output>
```

`order by`

Enables you to have results returned in the order you specify. Consider the following simple XQuery expression and its output (which is not sorted according to any user-specified order):

```
for $i in (5, 1, 2, 3)
return $i

5
1
2
3
```

You can use the `order by` keyword to sort results as desired. This example causes results to be returned in descending order:

```
for $i in (5, 1, 2, 3)
order by $i descending
return $i

5
3
2
1
```

DB2 support for XQuery

DB2 treats XQuery as a first-class language, allowing users to write XQuery expressions directly rather than requiring that users embed or wrap XQueries in SQL statements. DB2's query engine processes XQueries natively, meaning that it parses, evaluates, and optimizes XQueries without ever translating them into SQL behind the scenes. If you'd like, you can write "bilingual" queries that include both XQuery and SQL expressions. DB2 will process and optimize these queries as well.

To execute an XQuery directly in DB2, you must preface the query with the keyword `xquery`. This instructs DB2 to invoke its XQuery parser to process your request. You only need to do this if you are using XQuery as the outermost (or top level) language. If you embed XQuery expressions in SQL, you don't need to preface them with the `xquery` keyword.

In this tutorial you'll be using XQuery as the primary language, so all the queries shown here will be prefaced with `xquery`.

Section 3. Sample database environment

To help you learn XQuery, this tutorial refers to a sample "clients" table that contains a few XML documents. The following sections explain this table and its contents in greater detail, and describe the DB2-supplied facilities you can use to issue XQueries.

If you want to set up your DB2 system to include the sample table and contents, a script, tutorial.sql, is available. It contains all the code shown in this section.

Sample table

The clients table in our examples contains columns based on traditional SQL data types (such as integer and varying-length character strings), and a column based on the new SQL "XML" data type.

The first three columns track information about the IDs, names, and status of clients. Typical values for the status column include `Gold`, `Silver`, and `Standard`. The fourth column contains each client's contact information, such as home mailing address, phone numbers, email addresses, and so on. Such information is stored in well-formed XML documents.

Here's how the clients table is defined:

Listing 9. Client table definition

```
create table clients(
id                  int primary key not null,
name                  varchar(50),
status                  varchar(10),
contactinfo          xml
);
```

Sample XML documents

Before exploring how to write XQueries against this table, you need to populate it with some sample data. The SQL statements below insert six rows into the clients table. Each row contains an XML document, and the structure of each XML document varies somewhat. For example, email addresses are available for some customers but not others.

Listing 10. Sample data for the clients table

```
insert into clients values (3227, 'Ella Kimpton', 'Gold',
'<Client>
        <Address>
```

```
                <street>5401 Julio Ave.</street>
                <city>San Jose</city>
                <state>CA</state>
                <zip>95116</zip>
        </Address>
        <phone>
                <work>4084630000</work>
                <home>4081111111</home>
                <cell>4082222222</cell>
        </phone>
        <fax>4087776666</fax>
        <email>love2shop@yahoo.com</email>
</Client>'
);


insert into clients values (8877, 'Chris Bontempo', 'Gold',
'<Client>
        <Address>
                <street>1204 Meridian Ave.</street>
                <apt>4A</apt>
                <city>San Jose</city>
                <state>CA</state>
                <zip>95124</zip>
        </Address>
        <phone>
                <work>4084440000</work>
        </phone>
        <fax>4085555555</fax>
</Client>'
);


insert into clients values (9077, 'Lisa Hansen', 'Silver',
'<Client>
        <Address>
                <street>9407 Los Gatos Blvd.</street>
                <city>Los Gatos</city>
                <state>CA</state>
                <zip>95032</zip>
        </Address>
        <phone>
                <home>4083332222</home>
        </phone>
</Client>'
);


insert into clients values (9177, 'Rita Gomez', 'Standard',
'<Client>
        <Address>
                <street>501 N. First St.</street>
                <city>Campbell</city>
                <state>CA</state>
                <zip>95041</zip>
        </Address>
        <phone>
                <home>4081221331</home>
                <cell>4087799881</cell>
        </phone>
        <email>golfer12@yahoo.com</email>
</Client>'
);


insert into clients values (5681, 'Paula Lipenski', 'Standard',
'<Client>
        <Address>
                <street>1912 Koch Lane</street>
                <city>San Jose</city>
                <state>CA</state>
                <zip>95125</zip>
        </Address>
        <phone>
```

```
                <cell>4085430091</cell>
        </phone>
        <email>beatlesfan36@hotmail.com</email>
        <email>lennonfan36@hotmail.com</email>
</Client>'
);


insert into clients values (4309, 'Tina Wang', 'Standard',
'<Client>
        <Address>
                <street>4209 El Camino Real</street>
                <city>Mountain View</city>
                <state>CA</state>
                <zip>95033</zip>
        </Address>
        <phone>
                <home>6503310091</home>
        </phone>
</Client>'
);
```

Query environment

All the queries in this tutorial are designed to be issued interactively. You can do this through the DB2 command line processor or the DB2 Command Editor of the DB2 Control Center. Examples in this tutorial use the DB2 command line processor. (DB2 also ships with an Eclipse-based Developer Workbench that can help you graphically construct XQueries, but discussion of the Developer Workbench is beyond the scope of this tutorial.)

You can change default settings for the DB2 command line processor to make it easier to work with XML data. For example, the following command (issued from a DB2 command window) will launch the DB2 command processor in a manner that will cause XQuery results to be displayed in a format that's easy to read:

Listing 11. Setting DB2 command line processing options

```
db2 -i -d
```

This command causes DB2 to add extra white spaces to the displayed results of XQueries. DB2 doesn't really add these white spaces to your data. Your applications won't see items returned with extra white spaces -- they will only be visible from the DB2 command line processor window.

Section 4. Simple XML data retrieval operations

In this section you'll learn how to write XQueries that retrieve entire XML documents, and specific portions (or fragments) of XML documents. To do so, you'll use XPath expressions and FLWOR expressions.

Retrieving full XML documents stored in DB2

When running as a top-level language, XQuery needs to have a source of input data. In DB2, one way that you can specify this source of input data is by calling a function named `db2-fn:xmlcolumn`. This function takes an input parameter that identifies the DB2 table and XML column name of interest. The `db2-fn:xmlcolumn` function returns the sequence of XML documents that is stored in the given column. For example, the following query returns a sequence of XML documents containing customer contact information:

Listing 12. Simple XQuery to return customer contact data

```
xquery
db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')
```

You might be wondering why the table and column names specified in this query are in upper case. If you recall the SQL statement used earlier to create this table, you know that the table and column names were written in lower case. Unless you specify otherwise, DB2 folds table and column names into upper case in its internal catalog. Because XQuery is case-sensitive, lower-case table and column names would fail to match upper-case names in the DB2 catalog.

Now, let's consider the output of this XQuery. Given the sample data inserted into the clients table, the output of the query in Listing 12 is a sequence of six XML documents, as shown below.

Listing 13. Output from previous query

```
<?xml version="1.0" encoding="windows-1252" ?>
<Client>
        <Address>
                <street>
                        5401 Julio Ave.
                </street>
                <city>
                        San Jose
                </city>
                <state>
                        CA
                </state>
                <zip>
                        95116
                </zip>
        </Address>
        <phone>
                <work>
                        4084630000
                </work>
                <home>
                        4081111111
                </home>
                <cell>
                        4082222222
                </cell>
        </phone>
        <fax>
                4087776666
        </fax>
        <email>
                love2shop@yahoo.com
        </email>
</Client>
<?xml version="1.0" encoding="windows-1252" ?>
<Client>
        <Address>
                <street>
                        1204 Meridian Ave.
                </street>
                <apt>
                        4A
                </apt>
                <city>
                        San Jose
                </city>
                <state>
                        CA
                </state>
                <zip>
                        95124
                </zip>
        </Address>
        <phone>
                <work>
                        4084440000
                </work>
        </phone>
        <fax>
                4085555555
        </fax>
</Client>
<?xml version="1.0" encoding="windows-1252" ?>
<Client>
        <Address>
                <street>
                        9407 Los Gatos Blvd.
                </street>
                <city>
                        Los Gatos
                </city>
                <state>
                        CA
                </state>
                <zip>
                        95032
                </zip>
        </Address>
        <phone>
                <home>
                        4083332222
                </home>
        </phone>
</Client>
<?xml version="1.0" encoding="windows-1252" ?>
<Client>
        <Address>
                <street>
                        501 N. First St.
                </street>
                <city>
                        Campbell
                </city>
                <state>
                        CA
                </state>
                <zip>
                        95041
                </zip>
        </Address>
        <phone>
                <home>
                        4081221331
                </home>
                <cell>
                        4087799881
                </cell>
        </phone>
        <email>
                golfer12@yahoo.com
        </email>
</Client>
<?xml version="1.0" encoding="windows-1252" ?>
<Client>
        <Address>
                <street>
                        1912 Koch Lane
                </street>
                <city>
                        San Jose
                </city>
                <state>
                        CA
                </state>
                <zip>
                        95125
                </zip>
        </Address>
        <phone>
                <cell>
                        4085430091
                </cell>
        </phone>
        <email>
                beatlesfan36@hotmail.com
        </email>
        <email>
                lennonfan36@hotmail.com
        </email>
</Client>
<?xml version="1.0" encoding="windows-1252" ?>
<Client>
        <Address>
                <street>
                        4209 El Camino Real
                </street>
                <city>
                        Mountain View
                </city>
                <state>
                        CA
                </state>
                <zip>
                        95033
                </zip>
        </Address>
        <phone>
                <home>
                        6503310091
                </home>
        </phone>
</Client>
```

In case you're curious, you can also use plain SQL to retrieve the full set of XML documents contained in the `contactinfo` column. The simple statement "`select contactinfo from client`" will do so.

Retrieving specific XML elements

Quite often, users want to retrieve specific elements from XML documents. Doing so with XQuery can be simple. Imagine that you want to retrieve the fax numbers of all clients who provided this information. Here's one way to write such a query:

Listing 14. FLWOR expression to retrieve client fax data

```
xquery
for $y in db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client/fax
return $y
```

The first line instructs DB2 to invoke its XQuery parser. The next line instructs DB2 to iterate through the fax sub-elements of the Client elements contained in the CLIENTS.CONTACTINFO column. Each fax element is bound in turn to the variable $y. The third line indicates that, for each iteration, the value of $y is returned. The result is a sequence of XML elements, as shown below.

Listing 15. Sample output for previous query

```
<fax>4087776666</fax>
<fax>4085555555</fax>
```

(The output shown is simplified somewhat. The XML version information has been stripped off because it's not important for this tutorial. However, the XQueries you run in DB2 will return such information. See Listing 13 for an example.)

The query shown in Listing 14 could be also be expressed simply as a three-step path expression:

Listing 16. Path expression to retrieve client fax data

```
xquery
db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client/fax
```

In XQuery fundamentals you learned about text nodes. Let's apply that knowledge here. Imagine that you don't want to obtain XML fragments from your query, but instead want just a text representation of qualifying XML element values. To do this, you can invoke the `text()` function in your `return` clause:

Listing 17. Two queries to retrieve text representation of client fax data

```
xquery
for $y in db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client/fax
return $y/text()

(or)

xquery
db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client/fax/text()
```

The output of these queries is:

Listing 18. Sample output from previous queries

```
4087776666
4085555555
```

The results of the previous queries in this section are relatively simple because both involve the fax element, which is based on a primitive data type. Of course, elements may be based on complex types, containing sub-elements (or nested hierarchies). The Address element of our client contact information is an example of this. Consider what the following XQuery will return:

Listing 19. XQueries to retrieve complex XML type

```
xquery
for $y in db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client/Address
return $y

(or)

xquery
db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client/Address
```

If you guessed a sequence of XML fragments containing Address elements and all their sub-elements, you're right. Here's an excerpt from the output:

Listing 20. Partial output from previous query

```
<Address>
  <street>5401 Julio Ave.</street>
  <city>San Jose</city>
  <state>CA</state>
  <zip>95116</zip>
</Address>
<Address>
  <street>1204 Meridian Ave.</street>
  <apt>4A</apt>
  <city>San Jose</city>
  <state>CA</state>
  <zip>95124</zip>
</Address>
<Address>
   <street>9407 Los Gatos Blvd.</street>
   <city>Los Gatos</city>
   <state>CA</state>
   <zip>95032</zip>
</Address>

. . .
```

Section 5. Queries that filter on XML element values

Users often want to specify XML-based filtering criteria in their XQueries. And that's easy to do. In this section, you'll see how to make some of the previous XQuery examples more selective.

Specifying a single filtering predicate

Let's start by exploring how to return the mailing addresses of all customers who live in zip code 95116. You can incorporate a `where` clause into your XQuery to filter results based on the value of the zip element in the sample XML documents stored in DB2. To add a `where` clause to the FLWOR expression in Listing 19 to obtain only the addresses of interest:

Listing 21. FLWOR expression with a new "where" clause

```
xquery
for $y in db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client/Address
where $y/zip="95116"
return $y
```

The added `where` clause is fairly simple. The `for` clause binds the variable $y to each address in turn. The `where` clause contains a small path expression that navigates from each address to its nested zip element. The `where` clause is true (and the address is retained) only if the value of this zip element is equal to 95116.

Because only one client lives in the target zip code, the result returned is:

Listing 22. Output from previous query

```
<Address>
  <street>5401 Julio Ave.</street>
  <city>San Jose</city>
  <state>CA</state>
  <zip>95116</zip>
</Address>
```

The same result could be obtained by adding a predicate to the path expression, as follows:

Listing 23. Path expression with additional filtering predicate

```
xquery
db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client/Address[zip="95116"]
```

Specifying multiple filtering predicates

Of course, you can filter on zip code values and return elements unrelated to street addresses. You can also filter on multiple XML element values in a single query. The following query returns email information for customers who live anywhere in the city of San Jose or in zip code 95032 (which is a zip code in Los Gatos, California).

Listing 24. Filtering on multiple XML element values with a FLWOR expression

```
xquery
for $y in db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client
where $y/Address/zip="95032" or $y/Address/city="San Jose"
return $y/email
```

This example changes the `for` clause so that it binds variable $y to Client elements rather than to Address elements. This lets you filter the Client elements by one part of the subtree (Address) and return another part of the subtree (email). The path expressions in the `where` clause and `return` clause must be written relative to the element that is bound to the variable (in this case, $y).

The same query could be expressed somewhat more concisely as a path expression:

Listing 25. Filtering on multiple XML element values with a path expression

```

xquery
db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client[Address/zip="95032"
or Address/city="San Jose"]/email;
```

Given the sample data in the clients table, the output of either of these two previous queries is:

Listing 26. Query output

```

<email>
        love2shop@yahoo.com
</email>
<email>
        beatlesfan36@hotmail.com
</email>
<email>
        lennonfan36@hotmail.com
</email>
```

A practical example of how XQuery and SQL differ

If you inspect the query output in Listing 26, you may find that the returned results differ in two significant ways from what an SQL programmer might expect:

- No XML data is returned for qualifying customers who didn't provide their email addresses.
  Given our sample data, four customers meet our query selection criteria (they live in San Jose or zip code 95032), yet this fact isn't reflected in the query results. Why? Email elements are absent for two of these customers' records. Since XQuery doesn't use nulls, this "missing" information isn't reflected in the results.

- The output doesn't indicate which email addresses were derived from the same XML document.
  A close look at the sample data indicates that the final two email

addresses shown in Listing 26 are contained in one XML document (that is, they belong to one customer). This isn't obvious from the output.

Both situations can be desirable under some circumstances and undesirable under others. For example, if you want to send an email to every qualifying account on record, then iterating through a list of customer email addresses in XML format is easy to do in an application. However, if you want to mail only one notice to every customer -- including those who only provided their street addresses -- then the XQueries previously shown aren't sufficient.

There are multiple ways you can rewrite the previous queries so the returned results represent missing information in some fashion, and indicate when multiple email addresses were derived from the same customer record (that is, the same XML document). You'll learn how to do that a little later in this tutorial. However, if you just want to retrieve a list containing one email address per qualifying customer, you could modify the previous queries slightly:

Listing 27. Returning the first email element of qualifying clients

```
xquery
for $y in db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client
where $y/Address/zip="95032" or $y/Address/city="San Jose"
return $y/email[1]

(or)

xquery
db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client[Address/zip="95032"
or Address/city="San Jose"]/email[1];
```

Both of these queries instruct DB2 to return the first email element it finds within each qualifying XML document (customer contact record). If it doesn't find an email address for a qualifying customer, it won't return anything for that customer. Thus, both of these queries produce this output:

Listing 28. Query output

```
<email>
        love2shop@yahoo.com
</email>
<email>
        beatlesfan36@hotmail.com
</email>
```

Section 6. XML data transformations

A powerful aspect of XQuery is its ability to transform XML data from one form of XML into another. In this section, you'll learn how easy this is to do.

Converting XML to HTML

In Web based applications, it's common to convert all or part of XML documents into HTML for easy display. With XQuery, this process is straightforward. Consider the following query, which retrieves the addresses of clients, sorts the results by zip code, and converts the output into XML elements that are part of an unordered HTML list:

Listing 29. Querying DB2 XML data and returning results as HTML

```
xquery
<ul> {
for $y in db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client/Address
order by $y/zip
return <li>{$y}</li>
} </ul>
```

The query begins simply enough with the `xquery` keyword to indicate to the DB2 parser that XQuery is being used as the primary language. The second line causes the HTML markup for an unordered list (<ul>) to be included in the results. It also introduces a curly bracket, the first of two sets used in this query. Curly brackets instruct DB2 to evaluate and process the enclosed expression rather than treat it as a literal string.

The third line iterates over client addresses, binding the variable $y to each Address element in turn. The fourth line includes an `order by` clause, specifying that results must be returned in ascending order (the default order) based on customer zip codes (the zip sub-element of each Address bound to $y). The `return` clause indicates that the Address elements are to be surrounded by HTML list item tags before return. And the final line concludes the query and completes the HTML unordered list tag.

A portion of the output resulting from this query is:

Listing 30. HTML output of previous query

```
<ul>
  <li>
     <Address>
         <street>9407 Los Gatos Blvd.</street>
         <city>Los Gatos</city>
         <state>CA</state>
         <zip>95032</zip>
     </Address>
  </li>
  <li>
     <Address>
         <street>4209 El Camino Real</street>
         <city>Mountain View</city>
         <state>CA</state>
        <zip>95033</zip>
     </Address>
  </li>
. . .
</ul>
```

Using transformations to indicate missing or repeating elements in XML documents

Let's consider a topic we raised earlier: how to write an XQuery that will indicate missing values in the returned results, and will indicate when a single XML document (such as a single customer record) contains repeating elements, such as multiple email addresses. One way to do so involves wrapping the returned output in a new XML element, as shown in the following query:

Listing 31. Indicating missing values and repeating elements in an XQuery result

```
xquery
for $y in db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client
where $y/Address[zip="95032"] or $y/Address[city="San Jose"]
return <emailList> {$y/email} </emailList>
```

Running this query causes a sequence of emailList elements to be returned, one per qualifying customer record. Each emailList element will contain email data. If DB2 finds a single email address in a customer's record, it will return that element and its value. If it finds multiple email addresses, it will return all email elements and their values. Finally, if it finds no email address, it will return an empty emailList element.

Given our sample data, the output of this query is:

Listing 32. Output of previous query

```
<emailList>
   <email>love2shop@yahoo.com</email>
</emailList>
<emailList/>
<emailList/>
<emailList>
   <email>beatlesfan36@hotmail.com</email>
   <email>lennonfan36@hotmail.com<emailList>
</emailList>
```

Section 7. Conditional logic

You can incorporate conditional logic into your XQueries by using a few simple keywords.

Imagine that you'd like to contact each of your clients. You'd prefer to contact them by email, but if you don't have an email address, then you want to phone them at home. If you don't have a home phone number either, you want to send them a letter by postal mail. Thus, you need to query your DB2 clients table and assemble a contact list that may contain a single email address, home phone number, or postal mailing address for each of your clients.

This task is easy to accomplish if you incorporate conditional logic into your XQuery. Here's one way you can get the necessary information:

Listing 33. XQuery with a three-part conditional expression

```
xquery
for $y in db2-fn:xmlcolumn('CLIENTS.CONTACTINFO')/Client
return (
  if ($y/email) then $y/email[1]
  else if ($y/phone/home) then <homePhone>{$y/phone/home/text()}</homePhone>
  else $y/Address)
```

Let's look at lines 4 through 6 of this query. As you can see, they are part of the `return` clause, and they determine the query's output.

Line 4 tests if at least one email element is present in the document; if so, it specifies that the first email element should be returned. If no email elements are present, Line 5 is executed. It instructs DB2 to try to locate a home element beneath the phone element. If a home phone number is included in the document, its text node is extracted and returned as part of a new "homePhone" element. Finally, if neither an email address nor a home phone number is present in the client's profile (XML document), DB2 returns the full Address element. Because all records in the client table include a postal mailing address, the logic of this query ensures that DB2 will return one means of contacting each client.

The output from this query is:

Listing 34. Query output

```
<email>
        love2shop@yahoo.com
</email>

<Address>
        <street>
                1204 Meridian Ave.
        </street>
        <apt>
                4A
        </apt>
        <city>
                San Jose
        </city>
        <state>
                CA
        </state>
        <zip>
                95124
        </zip>
```

```
</Address>

<homePhone>
        4083332222
</homePhone>

<email>
        golfer12@yahoo.com
</email>

<email>
        beatlesfan36@hotmail.com
</email>

<homePhone>
        6503310091
</homePhone>
```

Section 8. "Hybrid" queries

You've seen how to write XQueries that retrieve XML document fragments, create new forms of XML output, and return different output based on conditions specified in queries themselves. These are a few simple ways to query XML data stored in DB2.

To be sure, there's more to learn about XQuery than this tutorial discusses. But there's one broad topic that we can't neglect: how to write queries that include both SQL and XQuery expressions. Doing so can be useful if you need to write queries that filter data based on XML and non-XML column values.

Because this tutorial focuses on XQuery and its use as a top-level language, we'll first explore how to embed SQL in XQueries. However, it's important to note that you can also do the opposite -- you can embed XQueries in SQL. You'll also see a brief example of how to do that at the end of this tutorial. However, for more information about SQL/XML and how to embed XQueries in SQL, see Resources.

Embedding SQL in XQueries

To embed SQL in an XQuery, you use the `db2-fn:sqlquery` function in place of the `db2-fn:xmlcolumn` function. The `db2-fn:sqlquery` function executes an SQL query and returns only the selected data. The SQL query passed to `db2-fn:sqlquery` must only return XML data; this enables XQuery to further process the result of the SQL query.

Let's use an example that incorporates many concepts we've already covered. Imagine that you want a list of all email addresses for Gold clients who live in San Jose. If a client has multiple email addresses, you want all of them to be included in the output as part of a single record. Finally, if a qualifying Gold client didn't provide an email address, you want to retrieve their postal mailing address. Here's one way to write such a query:

Listing 35. Embedding SQL within an XQuery that includes conditional logic

```
xquery
for $y in
db2-fn:sqlquery('select contactinfo from clients where status=''Gold'' ')/Client
where $y/Address/city="San Jose"
return (
     if ($y/email) then <emailList>{$y/email}</emailList>
     else $y/Address
)
```

Let's focus on the third line, which embeds an SQL statement. The `SELECT` statement contains a query predicate based on the `status` column, comparing the value of this VARCHAR column to the string Gold. In SQL, such strings are surrounded by single quotes. Although the example may appear to use double quotes, it actually uses two single quotes before and after the comparison value ("Gold"). The "extra" single quotes are escape characters. If you use double quotes around your string-based query predicate instead of pairs of single quotes, you'll get a syntax error.

The output of this query is:

Listing 36. Query output

```
<emailList>
        <email>
                love2shop@yahoo.com
        </email>

<Address>
        <street>
                1204 Meridian Ave.
        </street>
        <apt> 4A
        </apt>
        <city>
                San Jose
        </city>
        <state>
                CA
        </state>
        <zip>
                95124
        </zip>
</Address>
```

Embedding XQueries in SQL

It's important to understand that you can also embed XQueries in SQL. Indeed, DB2 9 features support for standard SQL/XML functions that are frequently used to formulate hybrid queries in which SQL is the outermost (or top-level) language. While it's beyond the scope of this tutorial to discuss DB2's SQL/XML support in detail, at least two functions are particularly worth noting:

`XMLExists`

Lets you navigate to an element (or other type of node) within your XML document and test for a specific condition. When specified as part of the SQL `WHERE` clause, `XMLExists` restricts the returned results to only those rows that contain an XML document that satisfies your specified condition (where the specified condition evaluates to "true"). As you might suspect, you pass an XQuery expression as input to the `XMLExists` function to navigate to the node of interest in your document.

`XMLQuery`

Lets you project XML into the SQL result set returned from your SQL/XML query. It's commonly used to retrieve one or more elements from XML documents. Again, as you might imagine, the `XMLQuery` function takes an XQuery expression as input.

Consider the following query, which uses SQL as the top-level language and includes calls to the `XMLQuery` and `XMLExists` functions:

Listing 37. Embedding XQuery path expressions in SQL to project and restrict XML data

```
select name, status,
xmlquery('$c/Client/phone/home' passing contactinfo as "c")
from clients
where xmlexists('$y/Client/Address[zip="95032"]' passing contactinfo as "y")
```

This query returns a result set with columns for the client's name, status, and home phone number. The first two columns contain SQL VARCHAR data, and the third column contains an XML element extracted from qualifying XML documents. Let's consider lines 2 and 4 of this query more closely.

Line 2 invokes `XMLQuery` as part of the `SELECT` clause, indicating that the result returned by this function should be included as a column in the SQL result set. As input to this function, we specified an XQuery path expression that causes DB2 to navigate to the home sub-element of the phone element directly beneath the Client element. The path expression references a variable ($c), which we declared to refer to the contactinfo column.

Line 4 invokes `XMLExists` within an SQL `WHERE` clause, indicating that DB2 should restrict results based on an XML predicate. This predicate is specified in the path expression -- it indicates that we're only interested in clients who live in a specific zip code. Once again, you'll note that the path expression provided as input to this SQL/XML function defines a variable ($y, in this case) that identifies where DB2 will find the XML documents of interest -- in the contactinfo column.

Given our sample data, the output of this query is a single-row result set with the following values:

Listing 38. Query output

```
Lisa Hansen          Silver    <home>4083332222</home>
```

For more information about DB2's SQL/XML support, see Resources.

Section 9. Summary

This tutorial introduced you to DB2's support of XQuery, and explained several basic language concepts. You learned how to write and execute simple XQueries against XML data stored in DB2.

To keep an eye on this series, bookmark the series page, DB2 9 Fundamentals exam 730 prep tutorial series.

Resources

Learn

- Check out the other parts of the DB2 9 Fundamentals exam 730 prep tutorial series.
- Certification exam site. Click the exam number to see more information about Exams 730 and 731.
- DB2 9 overview. Find information about the new data server that includes patented pureXML technology.
- DB2 XML evaluation guide: A step-by-step introduction to the XML storage and query capabilities of DB2 9 (developerWorks, Jun 2006): Take this tutorial to learn more about the DB2 Viper data server on Windows platforms using the XML storage and searching (SQL/XML, XQuery) capabilities available to support next-generation applications.
- Querying XML Data in DB2 with XQuery (developerWorks, Apr 2006): Learn basic XQuery expressions for querying DB2 XML data XQuery.
- Native XML Support in DB2 Universal Database Read this article to learn more about architecture and implementation of DB2's XML support.
- Get off to a fast start with DB2 Viper (developerWorks, Mar 2006): Learn how to insert, import, and validate XML data in DB2 9 (formerly known as DB2 Viper).
- Query DB2 XML Data with SQL (developerWorks, Mar 2006): Learn how to use several popular SQL/XML expressions to query DB2 XML data XQuery.
- Integration of SQL and XQuery in IBM DB2: Read this article from the IBM Systems Journal to learn how DB2 9 enables users to query XML data using SQL and XQuery.
- pureXML in DB2 9: Which way to query your XML data? (developerWorks, Jun 2006): Learn when to SQL/XML or XQuery to query XML data.
- DB2 XML wiki: Download books and peruse available papers on DB2 XML.
- DB2 9: Learn more about DB2's XML support.
- Take free tutorials on various XML technologies.
- Visit the developerWorks DBA Central zone to read articles and tutorials and connect to other resources to expand your database administration skills.
- Check out the developerWorks DB2 basics series, a group of articles geared toward beginning users.

Get products and technologies

- A trial version of DB2 9 is available for free download.
- Download DB2 Express-C, a no-charge version of DB2 Express Edition for the community that offers the same core data features as DB2 Express Edition and

provides a solid base to build and deploy applications.

Discuss

- Participate in the discussion forum for this content.

About the author

C. M. Saracco
C. M. Saracco is a senior software engineer at the IBM Silicon Valley Laboratory. She has written two books on database management topics (one co-authored with Charles J. Bontempo) and taught seminars in North America, South America, and Europe.