Process XML using XQuery

Learn to more easily and naturally search your XML data with XQuery 1.0

Nicholas Chase

27 March 2007
(First published 24 September 2002)

For years developers have used SQL to retrieve data from structured sources such as relational databases. But what about unstructured and semi-structured sources, such as XML data? To be viable as a data source, XML needed a means to conveniently retrieve the data. XQuery provides this means, allowing developers to write a statement that both extracts data and (if necessary) structures the results as XML. This tutorial shows you how to use XQuery to retrieve information from an XML document stored in an XQuery-enabled database. It also explains the ways in which XPath changes with version 2.0, and what those changes mean for data management.

Before you start

This tutorial teaches you how to use XQuery to retrieve information from an XML document. It also explains the ways in which XPath changes with version 2.0, and what those changes mean for data management.

You should be familiar with XML. An understanding of XPath 1.0 is helpful, but not required.

What is this tutorial about?

As XML matures as a format for storing data, the importance of an SQL-like language to retrieve information from a non-structured format such as XML documents grows. XML Query, or XQuery, is a response to this requirement. It utilizes several different structures and ideas, including significant enhancements to XPath that were developed for XSLT 2.0.

With XQuery, you can select very precise sets of data from an XML document or documents, and output an XML structure in response. It also includes sophisticated type management and the ability to create queries that adapt to different situations.

This tutorial guides you through the process of creating and understanding different types of queries available with XQuery and the changes in XPath 2.0. It progresses as follows:

- **XQuery basics**: Provides an overview of the topic, with brief explanations of each topic.
- **Using XQuery with DB2**: Walks you through setting up an IBM® DB2® V9 database for XQuery queries of stored XML.
- **FLWOR statements**: Explains the use of FOR-LET-WHERE-ORDER-RETURN statements, which are used in a manner similar to SQL.
- **XPath expressions**: Examines the ways in which XPath expressions select specific nodes from an XML document.
- **Sequences**: Looks at sequences, which replace XPath 1.0's node-sets, and some of the operations and functions that can act on them.
- **Additional features:** Explains some of the more advanced constructions available in an XQuery expression.
- **XQueryX**: Explains the XML-structured form of XQuery that corresponds to the human-readable form.

Prerequisites

The output for all queries is shown in the tutorial, so it's not necessary to actually run them in order to understand the concepts discussed. If, however, you do want to run some of the queries, you can download and install IBM DB2 version 9, which includes PureXML™ capabilities that make it possible to embed XQuery in an SQL statement. The code in this tutorial was tested on DB2 Enterprise 9.

You can download a trial version of DB2 9: DB2 Enterprise 9 or DB2 Express-C 9, a no-charge version of the DB2 Express 9 data server.

XQuery basics

To start, let's explore just what XQuery actually is.

What is XQuery?

XML Query, or XQuery, is a powerful way to search XML data for specific information. Sharing common roots with XPath 2.0, some of XQuery -- such as traditional XPath expressions -- looks familiar, while other aspects, such as FLWOR and conditional statements, are completely new. To help eliminate interpretation problems, the XQuery working group created the "XML Query Use Cases" document (see Resources, which provides sample queries and their expected results. The examples in this tutorial are based on some of the sample data created for that document.

XQuery is a good example of the interrelation of XML specifications and W3C Recommendations. The XQuery Working Group, along with the XSL Working Group, is also responsible for XPath 2.0, which includes much of the power developed for XQuery. In fact, XQuery is part of an entire group of Recommendations approved together:

- **XQuery 1.0: An XML Query Language**: A user-focused document that explains the language and its syntax.
- **XML Path Language (XPath) 2.0**: Another user-focused document, this time explaining the syntax and concepts behind XML Path language.

- **XQuery 1.0 and XPath 2.0 Functions and Operators**: A list of all of the functions and operators available in XQuery and XPath, and what they do. This is a very long document.
- **XQuery 1.0 and XPath 2.0 Data Model (XDM)**: A formal definition of the data model on which all of these functions and operators act.
- **XSLT 2.0 and XQuery 1.0 Serialization**: Defines the way in which XQuery, data should be serialized.
- **XQuery 1.0 and XPath 2.0 Formal Semantics**: Explains in detail exactly what terms of the other specifications are intended to mean.
- **XML Syntax for XQuery 1.0 (XQueryX)**: Specifies how to represent an XQuery query using XML.

Fortunately, you do not need to read all of these documents to get started. XQuery 1.0: An XML Query Language details almost everything you'll need, though you might find yourself occasionally turning to XML Path Language (XPath) 2.0 or XQuery 1.0 and XPath 2.0 Functions and Operators for particularly complex problems. Several additional specifications, such as update capabilities, are still in the pipeline.

In addition to enhancements to XPath, XQuery allows you to create complex queries by nesting SQL-like clauses, and to create complex results by including XML constructors directly in the output. This section provides an overview of these aspects of XQuery, each of which is then covered in greater detail.

XPath 2.0 enhancements

Enhancements to XPath generally fall into four categories:

- **Sequences**: Where XPath 1.0 typically returned node-sets, XPath 2.0 returns sequences. As discussed in Sequences, sequences are similar to node-sets, but provide more capabilities. One of the most important differences between node-sets and sequences is the fact that node-sets, unlike sequences, are unordered. Every XPath 2.0 expression returns a sequence, even if it's a sequence of just one item.
- **Data types**: XPath 1.0 understood four data types: node-sets, strings, booleans, and numbers. XPath 2.0 includes support not only for XML Schema's built-in data types, but also for user-defined data types.
- **Enhanced function set**: XPath 2.0 includes functions for manipulating data, just as 1.0 did. It also includes fairly robust support for sequence comparisons, such as the ability to check node equality to a specified depth or to select the intersection or union of two sequences. It also includes a myriad of new computational abilities, such as date functions and other mathematics.
- **Multiple sources**: XPath 2.0 includes the ability to explicitly set the source of data for an expression, making it possible to combine data from multiple sources into a single query.

FLWOR statements

FLWOR is short for `FOR-LET-WHERE-ORDER-RETURN`, which describes the structure of a typical XQuery. In a FLWOR statement, data is bound to a variable that is then used by subsequent steps. Take, for example, the following statement in Listing 1.

Listing 1. A sample FLWOR statement

```
for
  $book in doc("http://www.bn.com/bib.xml")//book
let
  $title := $book/title
where
  $book/publisher = 'Addison-Wesley'
return
  <bookInfo>
        { $title }
  </bookInfo>
```

This returns a `bookInfo` element for every book element in the original document that has a publisher child of Addison-Wesley. Each `bookInfo` element has a title element child, as in Listing 2.

Listing 2. Sample results

```
<bookInfo>
  <title>TCP/IP Illustrated</title>
</bookInfo>
<bookInfo>
  <title>Advanced Programming in the UNIX environment</title>
</bookInfo>
```

The section FLWOR statements discusses this topic in detail.

Constructors

A constructor is a means to create XML elements and attributes, CDATA sections, comments, or processing instructions within the results of an XQuery expression. The XQuery Recommendation describes two types of constructors, which allow the direct creation of elements rather than the indirect creation enabled by an XSLT template element.

The first is a typical constructor that simply embeds results within an element or other construct, as illustrated in the following expression in Listing 3.

Listing 3. Using a constructor

```
<books>
{
    for $book in doc("http://www.bn.com/bib.xml")/bib/book
    return
        <itemInfo>
            { $book/title }
            { $book/price }
        </itemInfo>
}
</books>
```

The second is known as a computed constructor, because it enables the creation of an element with a name that is determined by the result of an expression. For example, you can duplicate the listing above using this syntax (see Listing 4).

Listing 4. The computed element syntax

```
element books
{
    for $book in doc("http://www.bn.com/bib.xml")/bib/book
    return
        element itemInfo {
            { $book/title }
            { $book/price }
        }
}
```

But you might also use this ability to create a document that names the itemInfo element after the publisher (see Listing 5).

Listing 5. Using a computed constructor

```
element books
{
    for $book in doc("http://www.bn.com/bib.xml")/bib/book
    return
        element {$book/publisher} {
            { $book/title }
            { $book/price }
        }
}
```

Sequences

One of the major advantages of sequences over node-sets is the fact that sequences are ordered. True, an XPath 1.0 expression could pick out a node based on document order, but XPath 2.0 understands the order of the sequence itself. For example, the following statement, from the use-cases document, outputs the text of any actions that took place between the first and second incision of the first procedure during an operation (see Listing 6).

Listing 6. Basing information on sequence order

```
<critical_sequence>
 {
  let $proc := doc("report1.xml")//section[section.title="Procedure"][1],
      $i1 :=  ($proc//incision)[1],
      $i2 :=  ($proc//incision)[2]
  for $n in $proc//node() except $i1//node()
  where $n >> $i1 and $n << $i2
  return $n
 }
</critical_sequence>
```

More than just determining the order in which nodes appear, sequences are also important in terms of how you combine them and how they interact. The section Sequences discusses sequences in detail.

Functions

XQuery provides the ability to define functions that can later be used within other expressions (see Listing 7).

Listing 7. Defining a local function

```
define function local:summary($emps as element (employee)*)
   as element(dept)*
{
   for $d in fn:distinct-values($emps/deptno)
   let $e := $emps[deptno = $d]
   return
      <dept>
         <deptno>{$d}</deptno>
         <headcount> {fn:count($e)} </headcount>
         <payroll> {fn:sum($e/salary)} </payroll>
      </dept>
}

local:summary(fn:doc("acme_corp.xml")//employee[location = "Denver"])
```

Note that this example uses two predefined namespace aliases, local: (`http://www.w3.org/2005/xquery-local-functions`) and fn: (`http://www.w3.org/2005/xpath-functions`).

This capability makes it possible to not only use the functions that are predefined by XPath 2.0, but also to define your own on an as-needed basis. Functions are discussed in more detail in Creating functions.

Multiple sources

One of the great advantages of XQuery is its ability to combine input from multiple sources by specifying explicitly the source for a particular expression. For example, the query in Listing 6 compares the information in one file, items.xml, with the information in another, bids.xml (see Listing 8).

Listing 8. Using multiple sources

```
<result>
  {
    for $i in  doc("items.xml")//item_tuple
    let $b :=  doc("bids.xml")//bid_tuple[itemno= $i/itemno]
    where contains($i/description, "Bicycle")
    return
        <item_tuple>
            { $i/itemno }
            { $i/description }
            <high_bid>{ max($b/bid) }</high_bid>
        </item_tuple>
    sortby(itemno)
  }
</result>
```

The result includes information from both files. More information on data sources is provided in Data sources.

Using XQuery with DB2

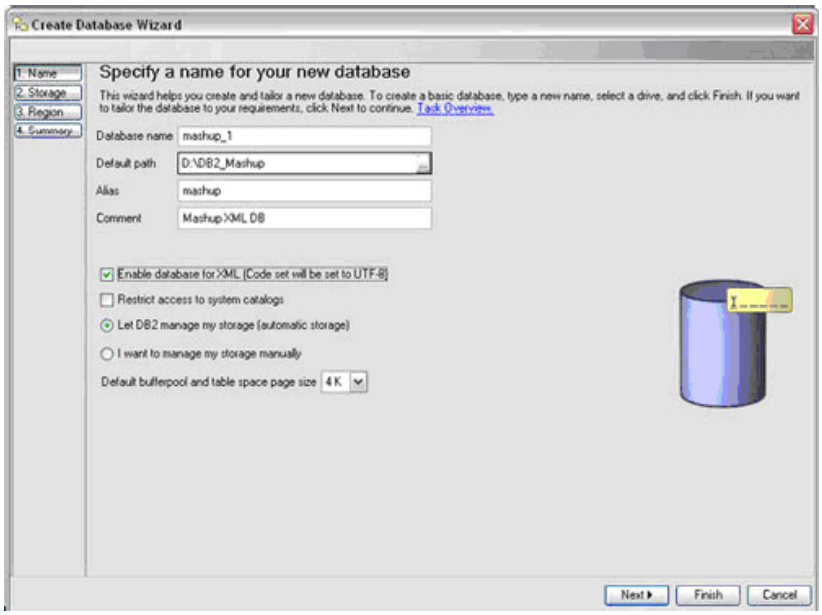
All this is interesting, but to move beyond the academic stage, you will need to have some sort of XQuery engine against which to run your queries. Fortunately, IBM DB2 version 9 includes PureXML, a native XML engine, which you can use to run your XQuery queries. Let's look at setting up DB2 for XML.

Create a DB2 database

To take advantage of the XML features of DB2, you need to configure the new database deliberately, so you will go through this creation process step by step.

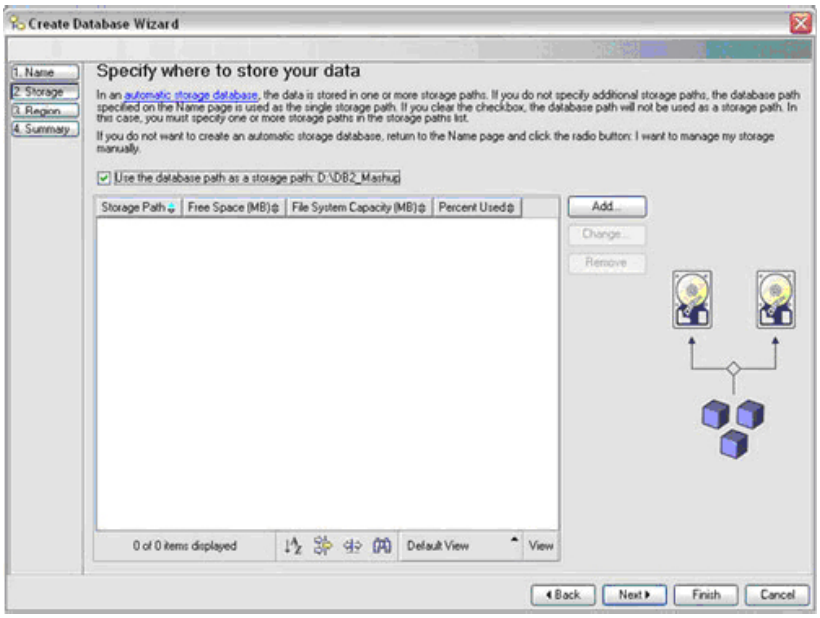
1. Download DB2 and install it according to the instructions. From there, you will need to create a database to store your XML data.
2. On your Microsoft® Windows® task bar, in the system tray, you should have an icon representing your instance of the DB2 server, which should be named DB2.
3. Right-click the icon and select DB2 Control Center to open the DB2 Control Center.
4. On the right hand side, you will see a branching menu with two options, All Systems and All Databases. Right-click All Databases and select **Create Database>Standard**. This action opens the database creation wizard, shown in Figure 1.

Figure 1. Database creation wizard, step 1



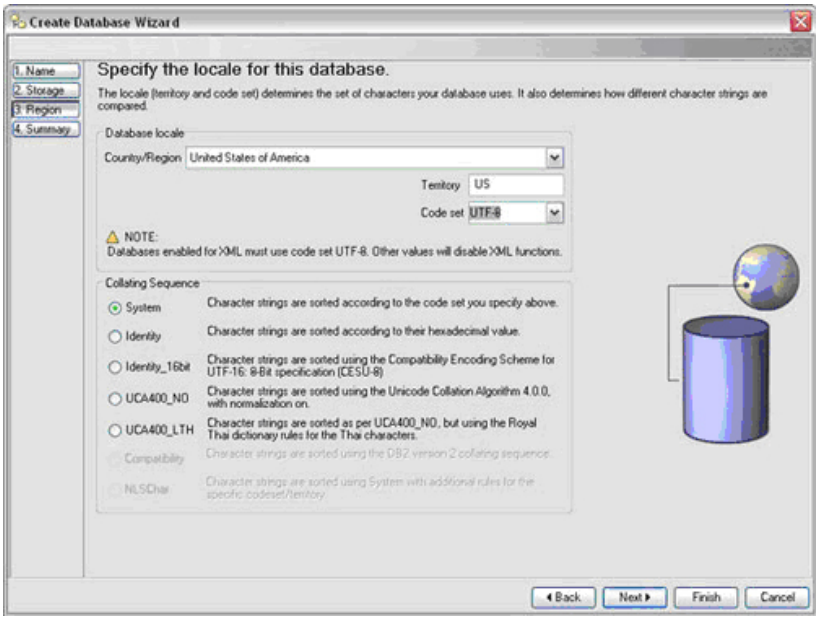
5. Here you are asked what you want to name the database and where you want to store it. For this tutorial, use mashup_1 for the name and alias of the database. You can add any comment name you like.
6. Next, select where you will store the database files. Databases can hold tremendous amounts of data. Select a location that can handle the volume of data for your DB2 applications. For this application on your development machine, you should only need a minimal mount of space. For now, select a location which has more than 10MB of space available and create a folder there. Type that folder path in the Default path field. You can also click Browse and select a folder that way.
7. Click to enable the Enable database for XML checkbox, which requires that you set code type to UTF-8. This is the key for your requirements.
8. Leave the remaining settings at the defaults and click Next (see Figure 2).

Figure 2. Database creation wizard, step 2



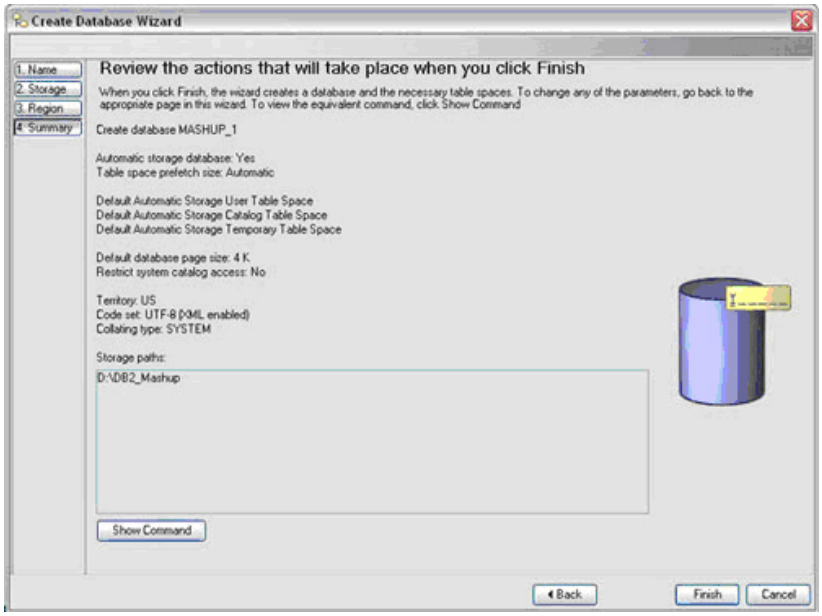
9. Again, the installer asks you where to store the data for the DB. Leave the Use the database path as a storage path checkbox enabled and click Next (see Figure 3).

Figure 3. Database creation wizard, step 3



10. Again you encounter a critical selection for your new database, the locale settings. For XML tables to function, they require UTF-8 encoding. You can select any country code, but you must select UTF-8 in the Code set drop-down list. Leave Collating Sequence at its default value and click Next (see Figure 4).

Figure 4. Database creation wizard, step 4



11. Finally, you will be rewarded with a summary screen and a Finish button. Click Finish and wait a moment as the system creates the initial files and starts the processes for your new database. Once the database is complete, you will find it under the All Databases tree. You can expand the mashup_1 item and see the components.

Creating the table

Now it's time to create a table to hold your data. The easiest way to do this is to execute queries in the DB2 Control Center.

In this tutorial, you are only interested in executing XQuery against specific XML, but one of the strengths of DB2 is the ability to combine XML and relational data. For example, you might have a table structured like this (see Listing 9).

Listing 9. A simple table combining XML and relational data

```
create table
create table monthlyReports (
     id INT generated always as identity,
     reportDate TIMESTAMP,
     reportData XML)
```

This would allow you to execute a query such as shown in Listing 10.

Listing 10. A simple query

```
select xmlquery('$sales/total' passing reportData as "sales") from monthlyReports
where reportDate < '1-1-2007'
```

This query would select a specific XML node from any record in the table with a `reportDate` less than January 1, 2007. In this case, however, you are going to concentrate on the XQuery part

of the equation, so you will create a table with only an XML column, and you will insert just one record (see Listing 11).

Listing 11. The table creation SQL

```
create table bookinfo (bibliography XML)
```

You can then insert the data into the table (see Listing 12).

Listing 12. Inserting the data

```
insert into bookinfo (bibliography) values
   (XMLPARSE(document cast ('<?xml version="1.0"?>
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price> 65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the UNIX environment</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author>
      <last>Abiteboul</last>
      <first>Serge</first>
    </author>
    <author>
      <last>Buneman</last>
      <first>Peter</first>
    </author>
    <author>
      <last>Suciu</last>
      <first>Dan</first>
    </author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price> 39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>
    <editor>
      <last>Gerbarg</last>
      <first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>' as clob) preserve whitespace))
```

(If you are executing this command from the command line, you will need to remove all of the line feeds.)

Now that you have an XQuery engine and some data,you can start looking at queries.

FLWOR statements

Now you can start to look at some actual XQuery queries.

Anatomy of a FLWOR statement

FLWOR statements are the closest thing that XQuery has to an SQL statement. With FLWOR statements, you can create very specific queries in a more natural way than XPath 1.0 statements did.

For example, consider this request: "Using bib.xml, provide `bookInfo` elements for the Addison-Wesley books, with the content of each consisting of the title element." Certainly, you could do it with XSLT, but it would be impossible to do using XPath 1.0 by itself. You could select the title elements, but you couldn't add them into the new `bookInfo` element.

Using the FLWOR statement, you can create the desired results (see Listing 13).

Listing 13. A simple FLWOR statement

```
for $book in doc("http://www.bn.com/bib.xml")//book
let $title := $book/title
where $book/publisher = 'Addison-Wesley'
return

<bookInfo>
        { $title }
</bookInfo>
```

Before moving on, I want to take a moment to discuss the slight difference between using XQuery for an individual file and using XQuery to look at data and the database. Notice here that I use the `doc()` function to reference an XML document through its URL. This function returns the nodes present in the document at that URL. The corresponding query for DB2 is shown in Listing 14.

Listing 14. A simple FLWOR statement in DB2

```
select xmlquery('for $book in $bib//book
                   let $title := $book/title
                   where $book/publisher = "Addison-Wesley"
                   return
                      <bookInfo>{ $title }</bookInfo>'
                                passing bibliography as "bib") from bookinfo
```

Following it through literally, this statement is translated as "For each book in the document, set a variable equal to the `title` element, and if the book publisher is Addison-Wesley, return a `bookInfo` element that contains the `title` element."

This statement shows the major aspects of a FLWOR statement. Notice first that variables are preceded with the dollar sign, $. These variables are bound to different node sequences, which can then be passed down through the statement. The *$book* variable, for example, is used by both the let and where clauses.

Curly braces, {}, distinguish information that should simply be output (such as the `bookInfo` start and end tags) from that which should be evaluated, such as the *$title* variable.

This section discusses each of these clauses and how they're used.

The source document

The examples in this section use the file bib.xml, one of the sample files provided with the W3C's use cases (see Listing 15).

Listing 15. The source document

```
<?xml version="1.0"?>
<!DOCTYPE bib SYSTEM "dtd/bib.dtd">
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price> 65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the UNIX environment</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author>
      <last>Abiteboul</last>
      <first>Serge</first>
    </author>
    <author>
      <last>Buneman</last>
      <first>Peter</first>
    </author>
    <author>
      <last>Suciu</last>
      <first>Dan</first>
    </author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price> 39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>
    <editor>
      <last>Gerbarg</last>
      <first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

The FOR clause

To start, look at the FOR clause in an XQuery statement.

Like the for each structure that appears in some languages, the FOR clause loops through each node in the sequence, binding that node to the variable in order to send it on to the next step. For example, the expression in Listing 16 produces the result in Listing 17.

Listing 16. The FOR clause

```
<results>
 {
  for $book in doc ("http://www.bn.com/bib.xml")/bib/book  return
    <bookInfo>
       { $book/title }
    </bookInfo>
 }
</results>
```

(If you run this statement in DB2, remember to change the format to: `select xmlquery('<results> { for $book in $q/bib/book return <bookInfo> { $book/title } </bookInfo> }</results>' passing bibliography as "q") from bookinfo !)`

Listing 17. The result

```
<results>
  <bookInfo>
    <title>TCP/IP Illustrated</title>
  </bookInfo>
  <bookInfo>
    <title>Advanced Programming in the UNIX environment</title>
  </bookInfo>
  <bookInfo>
    <title>Data on the Web</title>
  </bookInfo>
  <bookInfo>
    <title>The Economics of Technology and Content for Digital TV</title>
  </bookInfo>
</results>
```

Notice that there's a bookInfo element for each node in the sequence returned by /bib/book. The situation is different for the LET clause, shown in The LET clause, part 1.

The LET clause, part 1

Where the FOR clause loops through each node in a sequence, the LET clause binds a variable to an entire sequence of nodes. For example, the query in Listing 18 returns the result in Listing 19.

Listing 18. The LET clause

```
<results>
 {
  let $book := doc("http://www.bn.com/bib.xml")/bib/book  return
    <bookInfo>
     { $book/title }
    </bookInfo>
 }
</results>
```

Listing 19 shows the result.

Listing 19. The result

```
<results>
  <bookInfo>
    <title>TCP/IP Illustrated</title>
    <title>Advanced Programming in the UNIX environment</title>
    <title>Data on the Web</title>
    <title>The Economics of Technology and Content for Digital TV</title>
  </bookInfo>
</results>
```

In this case, the `LET` clause has bound the variable *$book* to the entire sequence of four nodes returned by the /bib/book, so there's only one set of data returned. Notice that even though *$book* refers to a sequence of nodes, you can still use the expression `$book/title` to return just the `title` elements.

The LET clause, part 2

The `LET` clause can also bind more than one variable. For example, the query in Listing 20 returns the result in Listing 21.

Listing 20. Binding multiple variables

```
<results>
 {
  let $t := doc("http://www.bn.com/bib.xml")/bib/book/title,
      $p := doc("http://www.bn.com/bib.xml")/bib/book/publisher
return
    <bookInfo>
      { $t }
      { $p }
    </bookInfo>
 }
</results>
```

Listing 21 shows the result.

Listing 21. The result

```
<results>
  <bookInfo>
    <title>TCP/IP Illustrated</title>
    <title>Advanced Programming in the UNIX environment</title>
    <title>Data on the Web</title>
    <title>The Economics of Technology and Content for Digital TV</title>
    <publisher>Addison-Wesley</publisher>
    <publisher>Addison-Wesley</publisher>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <publisher>Kluwer Academic Publishers</publisher>
  </bookInfo>
</results>
```

Notice that both variables were set, though they were output as a whole, because they were set as a whole. This kind of arrangement typically makes more sense when the `FOR` and `LET` clauses are combined, as they are in The `WHERE` clause.

The WHERE clause

The WHERE clause works by evaluating an expression each time the statement attempts to RETURN information. Consider the query in Listing 22.

Listing 22. Adding a WHERE clause

```
<results>
 {
  for $book in doc("http://www.bn.com/bib.xml")/bib/book
    let $t := $book/title,
        $p := $book/publisher
    where $book/@year > 1992
    return
      <bookInfo>
        { $t }
        { $p }
      </bookInfo>
 }
</results>
```

Each time the statement tries to return information, the WHERE clause checks its expression with the currently bound *$book*. If it's true, then the RETURN clause outputs. If not, it doesn't. So the above query returns the following in Listing 23.

Listing 23. The result

```
<results>
  <bookInfo>
    <title>TCP/IP Illustrated</title>
    <publisher>Addison-Wesley</publisher>
  </bookInfo>
  <bookInfo>
    <title>Data on the Web</title>
    <publisher>Morgan Kaufmann Publishers</publisher>
  </bookInfo>
  <bookInfo>
    <title>The Economics of Technology and Content for Digital TV</title>
    <publisher>Kluwer Academic Publishers</publisher>
  </bookInfo>
</results>
```

First, notice that "Advanced Programming in the UNIX environment" is missing because it wasn't published after 1992.

Second, notice that the structure of the results is different from the example in The LET clause, part 2. When the LET clause is executed, *$book* is bound to a single book element, so *$t* and *$p* get only a single value. This process is repeated for each book element in the original document.

Binding to attributes

While the previous example (see the WHERE clause) did what was wanted, it would have been nice to see the actual year attached to the results. Because the year is an attribute rather than an element, however, it bears separate examination.

In this example, I add the year to the output (see Listing 24).

Listing 24. Working with attributes

```
<results>
 {
  for $book in doc("http://www.bn.com/bib.xml")/bib/book
  let $t := $book/title,
      $p := $book/publisher,
      $y := $book/@year
  where $y > 1992
  return
    <bookInfo>{ $y }
      { $t }
      { $p }
    </bookInfo>
 }
</results>
```

The *$y* variable is output in the same way as the *$t* and *$p* variables. You might expect it to simply appear as a text node in the `bookInfo` element, but it doesn't. Just as the title and publisher elements appear as elements, the year attribute appears as an attribute (see Listing 25).

Listing 25. The results

```
<results>
  <bookInfo year="1994">
    <title>TCP/IP Illustrated</title>
    <publisher>Addison-Wesley</publisher>
  </bookInfo>
  <bookInfo year="2000">
    <title>Data on the Web</title>
    <publisher>Morgan Kaufmann Publishers</publisher>
  </bookInfo>
  <bookInfo year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>
    <publisher>Kluwer Academic Publishers</publisher>
  </bookInfo>
</results>
```

Because XQuery knows that *$y* is bound to an attribute, it is added to its parent element as an attribute.

Sorting

You can sort a sequence by any relevant information. For example, you can sort the query in Binding to attributes by year (see Listing 26).

Listing 26. Sorting the data

```
<results>
 {
  for $book in doc("http://www.bn.com/bib.xml")/bib/book
  let $t := $book/title,
      $p := $book/publisher,
      $y := $book/@year
  where $y > 1992
  order by $y
  return
    <bookInfo>{ $y }
      { $t }
      { $p }
    </bookInfo>
 }
</results>
```

Nesting expressions, part 1

You can nest FLWOR statements within each other. For example, a statement can return the results of a second statement, so the expression in Listing 27 returns the result in Listing 28.

Listing 27. Nesting statements

```
<authors>
{
   for $book in doc("http://www.bn.com/bib.xml")//book
   let $authors := $book/author
   return
          for $author in $authors
                          return $author
}
</authors>
```

Listing 28 shows the returned result.

Listing 28. The result

```
<authors>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <author>
    <last>Abiteboul</last>
    <first>Serge</first>
  </author>
  <author>
    <last>Buneman</last>
    <first>Peter</first>
  </author>
  <author>
    <last>Suciu</last>
    <first>Dan</first>
  </author>
</authors>
```

Here the second, bolded FLWOR statement returns a sequence, which is then returned by the overall statement.

Nesting expressions, part 2

You can actually nest clauses anywhere you want to produce a sequence. For example, the result in Nesting expressions, part 1 could have been achieved with the statement in Listing 29.

Listing 29. Nesting statements, redux

```
<authors>
{
   for $book in doc("http://www.bn.com/bib.xml")//book
   let $authors := ( for $author in $book/author return $author )
   return
      $authors
}
</authors>
```

Here, the bolded statement still produces a sequence, but that sequence is bound directly to the *$authors* variable.

Combining clauses

Despite the name, no rule dictates that a FLWOR statement must read `FOR-LET-WHERE-RETURN`. Because you can nest statements, `FOR` and `LET` clauses can appear in any order. Consider, for example, the following query in Listing 30.

Listing 30. Combining clauses

```
<books>
  {
    let $doc := doc("http://www.bn.com/bib.xml")//book
    for $book in $doc
    let $title := $book/title
    let $author := $book/author
    where $book/@year > 1994
    return
      <bookInfo>{ $book/@year }
        { $title }
        { $author }
      </bookInfo>
  }
</books>
```

First, the engine binds the sequence of all book elements to the *$doc* variable. Next, it binds *$book* to each of those nodes, in turn, as they exist within the sequence. Each time it binds *$book* to a new node, it binds *$title* to any `title` elements that are children of the current *$book*, and *$author* to any `author` elements that are children of the current *$book*. It then checks the current year attribute and returns a `bookInfo` element with the appropriate children and attribute if the where condition is true. The result is this sequence in Listing 31.

Listing 31. The result

```
<books>
  <bookInfo year="2000">
    <title>Data on the Web</title>
    <author>
```

```
      <last>Abiteboul</last>
      <first>Serge</first>
    </author>
    <author>
      <last>Buneman</last>
      <first>Peter</first>
    </author>
    <author>
      <last>Suciu</last>
      <first>Dan</first>
    </author>
  </bookInfo>
  <bookInfo year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>

  </bookInfo>
</books>
```

Creating joins

Just as in SQL, XQuery makes it possible to create a query that joins two sets of data. Also as in SQL, however, if not done carefully, this can result in a Cartesian join, in which every node in the first sequence is tupled with every node in the second set. For example, the query in Listing 32 returns the result in Listing 33.

Listing 32. Creating a join

```
<books>
  {
    let $doc := doc("http://www.bn.com/bib.xml")//book
    for $title in $doc//title, $author in $doc//author
    return
      <bookInfo>
        { $title }
        { $author }
      </bookInfo>
  }
</books>
```

Listing 33 shows the returned result.

Listing 33. The result

```
<books>
  <bookInfo>
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
  </bookInfo>
  <bookInfo>
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
  </bookInfo>
  <bookInfo>
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Abiteboul</last>
      <first>Serge</first>
```

```
    </author>
  </bookInfo>
  <bookInfo>
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Buneman</last>
      <first>Peter</first>
    </author>
  </bookInfo>
  <bookInfo>
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Suciu</last>
      <first>Dan</first>
    </author>
  </bookInfo>
  <bookInfo>
    <title>Advanced Programming in the UNIX environment</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
  </bookInfo>
  <bookInfo>
    <title>Advanced Programming in the UNIX environment</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
  </bookInfo>
  <!-- And so on.  Repetitive data snipped for brevity's sake. -->
</books>
```

This is clearly not what is desired. This is because it's semantically the same query as the one in Listing 34.

Listing 34. Restating the join

```
<books>
  {
    let $doc := doc("http://www.bn.com/bib.xml")//book
    for $title in $doc//title
    for $author in $doc//author
    return
      <bookInfo>
        { $title }
        { $author }
      </bookInfo>
  }
</books>
```

To prevent this kind of problem, you need a `WHERE` clause, just as in an SQL statement (see Listing 35).

Listing 35. Sorting things out with a `WHERE` clause

```
<books>
  {
    let $doc := doc("http://www.bn.com/bib.xml")//book
    for $title in $doc//title, $author in $doc//author
    where $title/parent::node() = $author/parent::node()
    return
      <bookInfo>
        { $title }
        { $author }
      </bookInfo>
  }
</books>
```

Now, results are only returned if *$title* and *$author* share the same parent node. (XPath expressions such as this one are covered more fully in XPath expressions.)

Data sources

In all of the examples so far, the source of the data being queried has been a document identified by the URI `http://www.bn.com/bib.xml`. The `doc()` function can take any URI as an argument, but how it's ultimately handled is up to the implementation. For example, a Web-based XQuery demo might implement `doc()` in such a way that rather than actually retrieving the text of a URI, it looks up the URI and maps it to a local file, thus preventing unauthorized documents from being queried.

XQuery (along with XPath 2.0) also defines the `collection()` function, which is similar to `doc()` in that it takes a URI as an argument. However, where `doc()` is assumed to return the root node of an actual document, `collection()` might return any collection of nodes, no matter how they were generated.

One way to think of this is to once again turn to DB2. When you execute the query in Listing 36, it returns the result in Listing 37.

Listing 36. A simple query

```
select xmlquery('<books> {
                  for $book in $bib//book
                    let $title := $book/title
                    where $book/publisher = "Addison-Wesley"
                    return
                        <bookInfo>{ $title }</bookInfo>
                } </books>'
                                passing bibliography as "bib") from bookinfo
```

Listing 37 displays the expected results.

Listing 37. Results of the simple query

```
<bookInfo><title>TCP/IP Illustrated</title></bookInfo>
<bookInfo><title>Advanced Programming in the UNIX
environment</title></bookInfo>
```

This is expected because you only added one record to the bookinfo table, so you expect the variable *$bib* to refer to the XML document in that record.

All of this is very reasonable, however there's nothing to stop you from adding a second record, or third, or 58th. In this case, *$bib* actually represents a collection of nodes from all of these records. If you added a second identical record, the results would actually be (see Listing 38).

Listing 38. Results from more than one record

```
<books>
   <bookInfo><title>TCP/IP Illustrated</title></bookInfo>
   <bookInfo><title>Advanced Programming in the UNIX
environment</title></bookInfo>
   <bookInfo><title>TCP/IP Illustrated</title></bookInfo>
   <bookInfo><title>Advanced Programming in the UNIX
environment</title></bookInfo>
</books>
```

Notice that even though you deal with two different records, the result is combined into a single results because you deal with a single collection of nodes.

XPath expressions

XQuery is dependent upon the ability to select specific nodes, which involves the the use of XPath. Let's take a look at how that works.

The basic XPath expression

The heart of XQuery is the ability to select a particular sequence of nodes from a document, and that happens through an XPath expression. You are probably familiar with at least the basic type of XPath expression, such as: `/bib/book/@year`.

This expression starts at the document root and selects all `bib` children, then all `book` children of the `bib` children, and then all `year` attributes of the `book` children. This is known as the abbreviated form. While it is certainly useful, understanding the unabbreviated form that underlies it provides power that far exceeds what you can do with a simple statement such as this.

Axes and node tests

The XPath expression example in The basic XPath expression is actually an abbreviated form of `/child::bib/child::book/attribute::year`.

In each location step, such as `child::bib`, the resulting sequence narrows. It starts with the document root and all of its children, then narrows that set to just the children of the root node. Next, the expression narrows the set to only the children that satisfy the name test of `bib`. Now, the set includes just the `bib` children.

Next that sequence is narrowed further. Only the children of those `bib` elements are selected, and then all that aren't named book are eliminated. That leaves just the `book` grandchildren of the document root.

Finally, only the attributes of those `book` children are selected. Any that don't match the name test year are discarded, leaving only the year attribute of each book grandchild of the document root.

The `child:: axis` is one of 13 axes defined by XPath. The complete set includes:

- `child`
- `descendant`
- `attribute`
- `self`
- `descendant-or-self`
- `following-sibling`
- `following`
- `namespace`
- `parent`
- `ancestor`
- `preceding-sibling`
- `preceding`
- `ancestor-or-self`

The axis is followed by the node test, which tests either for a name, as in the above example, or for a node type. For example, to retrieve the text of the `book` element rather than the attribute, you could use: `/child::bib/child::book/child::text()`.

The node kind tests are `text(), comment(), processing-instruction()`, and `node()`, which returns a node of any type.

So, in Creating joins, when you used the expression: `$title/parent::node()` you selected all nodes that were the parent of the element bound to `$title` -- which is, of course, a single node; you weren't narrowing it further, as `node()` doesn't discriminate. This left the `parent` element for the `title` element, `book`.

Narrowing the sequence: predicates

A location step actually has three potential parts. Besides the axis and node test, you can add a qualifier that narrows the set even further. In XPath 1.0, qualifiers were limited to predicates. A predicate acts like a filter so you can create an expression against which to test each node in the sequence. The sequence keeps nodes that pass and removes nodes that fail.

For example, the expression: `//book[@year > 1992]` is the abbreviated form of `/descendant-or-self::book[attribute::year > 1992]`.

First, you gather all of the descendants of the document root, as well as the root itself. This includes all non-attribute nodes in the document. (Remember, an element is the parent of an attribute, but an attribute is not the child of the element.) That sequence is narrowed to include only those elements that are named book. From there, you test the `year` attribute of each `book` element. If the attribute is greater than 1992, the node is kept. If the attribute is equal to or less than 1992, the node is discarded.

Predicates can consist of fairly complex expressions, including XPath functions and Boolean expressions such as and and or.

Data types

In addition to the four data types already available in XPath 1.0, XPath 2.0 adds support for XML Schema built-in types such as date, duration, and QName. You can cast data values to these types as necessary using constructors, which is particularly useful for date values. For example, an expression might retrieve values based on two dates (see Listing 39).

Listing 39. Selecting data based on dates

```
<result>
  {
    for $i in doc("/XQuery/docs/R/items.xml")//item_tuple
    where xs:dateTime($i/start_date) <= xs:dateTime("2006-12-01T12:00:00")
      and xs:dateTime($i/end_date) >= xs:dateTime("2006-12-31T12:00:00")
      and contains ($i/description, "Bicycle")
    return
        <item_tuple>
            { $i/itemno }
            { $i/description }
        </item_tuple>
  }
</result>
```

The list of new types and related functions is much too long to include here. For a complete look, see the XQuery 1.0 and XPath 2.0 Functions and Operators document (see Resources).

XPath 2.0 can also detect user-defined types.

Sequences

One of the major differences between XPath 1.0 in XPath 2.0 is the addition of sequences, so it's important to understand what they are and what they bring to the party.

What is a sequence?

A *sequence* is a group of objects that exist in a particular order. These objects can be nodes or simple values such as strings or numbers. In XPath 2.0, every expression returns a sequence, even if it's a sequence of just one item.

Sequences are inherently shallow. If one sequence is added as an item to another sequence, its items become part of the overall sequence. For example, if the variable *$fruits* was bound to the sequence: (`"apples", "bananas", "oranges"`) then the sequence: (`"steak", "bananas", $fruits, "potatoes"`) would actually be equal to: (`"steak", "bananas", "apples", "bananas", "oranges", "potatoes"`.)

Note that duplicates are allowed in sequences. Also, where XPath 1.0 node-sets were unordered, sequences are ordered, so position can be ascertained relative to the sequence.

Position

In XPath, you can determine position exactly, or by using functions such as `first()` and `last()`. For example, the following query outputs the second book in the document by W. Stevens (see Listing 40).

Listing 40. Searching by position

```
<books>
  {
    let $doc := doc("http://www.bn.com/bib.xml")
    let $booksByStevens :=
              ($doc/bib/book[author/last='Stevens'])[position() = 2]
    return
        <bookInfo>
            { $booksByStevens/title }
        </bookInfo>
  }
</books>
```

The expression: `$doc/bib/book[author/last='Stevens']` returns a sequence of all the `book` elements that have an `author` child with a last child of Stevens. The predicate: `position() = 2` eliminates all nodes that don't have a position of 2 within the sequence, leaving only the second book (see Listing 41).

Listing 41. The results

```
<books>
  <bookInfo>
    <title>Advanced Programming in the UNIX environment</title>
  </bookInfo>
</books>
```

In this case, the expression determines which node is in a particular position. You can also use the `index-of()` function to determine the position of a particular node within a sequence.

Equality

When dealing with sequences, there are three types of equality:

- *Node comparisons* look at whether two sequences actually contain the same nodes.
- *Value comparisons* look at the actual values within the sequences to see if they're equal.
- *General comparisons* apply to individual values within an arbitrarily long sequence.

Value comparisons

The most familiar comparison is that of values. For example, you can compare the last name of the authors of the first two books, which is Stevens in both cases (see Listing 42).

Listing 42. Comparing values

```
let $doc := $bib
let $first := $doc//book[1]/author/last
let $second := $doc//book[2]/author/last
     return
         <results>
           {if ($first eq $second) then <true /> else () }
         </results>
```

This, of course, would, come up as true, because both variables referred to an element with the same value. Value comparisons use the operators `eq` (equal), `ne` (not equal), `lt` (less than), `le` (less than or equal to), `gt` (greater than) and `ge`, (greater than or equal to).

The general comparison gives similar results.

General comparisons

General comparisons use the familiar equals sign (=), not equals (!=), less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=). For an example see Listing 43.

Listing 43. Comparing values

```
let $doc := $bib
let $first := $doc//book[1]/author/last
let $second := $doc//book[2]/author/last
     return
         <results>
           {if ($first = $second) then <true /> else () }
         </results>
```

In a general comparison, the expression is considered to be true of if at least one of the operands satisfies the operator. For an example see Listing 44.

Listing 44. Comparing values

```
let $doc := $bib
let $first := $doc//book/author/last
let $second := $doc//book[2]/author/last
     return
         <results>
           {if ($first = "Stevens") then <true /> else () }
         </results>
```

This statement will also show the `true` element, because at least one of the nodes in the *$first* sequence has a value of Stevens.

Node comparisons

Node comparisons are different, in that they operate on the nodes themselves. For an example see Listing 45.

Listing 45. Comparing nodes

```
let $doc := $bib
let $first := $doc//book[1]/author/last
let $second := $doc//book[2]/author/last
     return
         <results>
           {if ($first is $second) then <true /> else () }
         </results>
```

This statement will not show the `true` element because the first node is not the same as the second node, even though they have the same value. Node comparisons use the following operators: `is`, which determines whether the two nodes are actually the same node, `<<`, which determines whether the first node comes before the second one, and `>>`, which checks to see whether the first node comes after the second node.

Aggregate functions

The aggregate functions, `min()`, `max()`, `count()`, `sum()`, and `avg()`, act on a sequence of nodes and return a sequence that consists of a single value. For example, the query in Listing 46 returns the result in Listing 47.

Listing 46. Aggregate functions

```
let $doc := doc("http://www.bn.com/bib.xml")
    return
        <results>
            { avg($doc//price) }
        </results>
```

Listing 47 shows the returned results.

Listing 47. The result

```
<results>
  75.45
</results>
```

These are not, however, the only functions that act on a sequence. Functions such as `distinct-values()`, and `insert-before()` either get information from the sequence or act upon it. For example, the query in Listing 48 removes the extra reference to W.Stevens, and returns the result in Listing 49.

Listing 48. The distinct-values() function

```
let $doc := doc("http://www.bn.com/bib.xml")
for $author in  distinct-values($doc//book/author)
    return
        <results>
            { $author }
        </results>
```

Listing 49 shows the returned result.

Listing 49. The result

```
<results>Stevens W.</results><results>Abiteboul
Serge</results><results>Buneman Peter</results>
<results>Suciu Dan</results>
```

Wondering why you are not seeing the full structure of the `author` elements? Remember, you're looping through the `distinct-values()`, so it's the values you get, and not the elements.

Set functions

XPath 2.0 also enables set-type operations such as intersections and unions on sequences. The query in Listing 50 produces the result in Listing 51.

Listing 50. Set functions

```
let $doc := doc("http://www.bn.com/bib.xml")
```

```
let $allBooks := $doc//book

let $authoredBooks := $doc//book[author]
let $editedBooks := $doc//book[editor]
let $dansBooks := $doc//book[author/first = 'Dan']

let $except := $allBooks except $dansBooks
let $intersection := $authoredBooks intersect $dansBooks
let $union := $authoredBooks union $editedBooks
return
   <books>
       <authoredBooks>
          { $authoredBooks/title }
       </authoredBooks>
       <dansBooks>
          { $dansBooks/title }
       </dansBooks>
       <intersection>
          { $intersection/title }
       </intersection>
       <except>
          { $except/title }
       </except>
       <union>
          { $union/title }
       </union>
   </books>
```

Listing 51 shows the returned result.

Listing 51. The result

```
<books>
  <authoredBooks>
    <title>TCP/IP Illustrated</title>
    <title>Advanced Programming in the UNIX environment</title>
    <title>Data on the Web</title>
  </authoredBooks>
  <dansBooks>
    <title>Data on the Web</title>
  </dansBooks>
  <intersection>
    <title>Data on the Web</title>
  </intersection>
  <except>
    <title>TCP/IP Illustrated</title>
    <title>Advanced Programming in the UNIX environment</title>
    <title>The Economics of Technology and Content for Digital TV</title>
  </except>
  <union>
    <title>TCP/IP Illustrated</title>
    <title>Advanced Programming in the UNIX environment</title>
    <title>Data on the Web</title>
    <title>The Economics of Technology and Content for Digital TV</title>
  </union>
</books>
```

Quantified expressions

One of the more interesting additions to XPath 2.0 is the notion of *quantified expressions*. In a quantified expression, you can test to see whether all, or even some of the nodes in a particular sequence fulfill a particular requirement. For example, you can specify that you only wanted to output a group of books if at least some of them were from a particular publisher (see Listing 52).

Listing 52. Selecting based on a quantified expression

```
if (some $book in doc("bib.xml")//book satisfies ($book/publisher ="Addison-Wesley"))
  <bib>  {
      for $b in doc("bib.xml")//book
         where count($b/author) > 0
         return
            <book>
              { $b/title }
            </book>
  }</bib>
else ()
```

This capability greatly simplifies tasks that have been virtually impossible in XPath 1.0.

Additional features

XQuery includes some fairly robust programming-like capabilities. This section takes a moment to look at two of them:

- `if-then` statements
- The ability to create a function

`If-then` statements

An `if-then-else` statement works just as it does in traditional programming languages. Consider, for example, the following use-case that outputs the first two authors found, then the `et-al` element if there are more than two authors for a particular book (see Listing 53).

Listing 53. Using an if-then-else statement

```
<bib>
  {
    for $b in doc("www.bn.com/bib.xml")//book
    where count($b/author) > 0
    return
        <book>
            { $b/title }
            {
                for $a in $b/author[position()<=2]
                return $a
            }
            {
                if (count($b/author) > 2)
                then <et-al/>
                else ()
            }
        </book>
  }
</bib>
```

The query returns Listing 54.

Listing 54. The result

```
Listing 50.
```

```
<bib>
    <book>
        <title>TCP/IP Illustrated</title>
        <author>
            <last>Stevens</last>
            <first>W.</first>
        </author>
    </book>
    <book>
        <title>Advanced Programming in the UNIX environment</title>
        <author>
            <last>Stevens</last>
            <first>W.</first>
        </author>
    </book>
    <book>
        <title>Data on the Web</title>
        <author>
            <last>Abiteboul</last>
            <first>Serge</first>
        </author>
        <author>
            <last>Buneman</last>
            <first>Peter</first>
        </author>
        <et-al/>
    </book>
</bib>
```

Creating functions

In addition to the predefined functions that are part of XPath, XQuery enables the creation of new functions. For example, you can create a function that provides an author count (see Listing 55).

Listing 55. Creating a new function

```
declare function local:author_summary($b as element(book)) as element(bookcount)
{
    <bookCount>
        { $b/title }
        <numberOfAuthors>{
          fn:count($b/author)
       }</numberOfAuthors>
    </bookCount>
}

<counts>
  {
    for $book in fn:doc("http://www.bn.com/bib.xml")//book
    return local:author_summary($book)
  }
</counts>
```

As mentioned in the earlier example of a user-defined function, this example uses two predefined namespace aliases, local: and fn:. The former refers to the functions you declare, the latter to built-in XQuery functions. Once you declare the function, you can use it just like a built-in function.

In this case, you declare the function, and then use it within the FLWOR statement. For each *$book* node, the function is executed and returns an element, which is then returned by the FLWOR statement. The resulting output is shown in Listing 56.

Listing 56. The result

```
<counts>
  <bookCount>
    <title>TCP/IP Illustrated</title>
    <numberOfAuthors>1</numberOfAuthors>
  </bookCount>
  <bookCount>
    <title>Advanced Programming in the UNIX environment</title>
    <numberOfAuthors>1</numberOfAuthors>
  </bookCount>
  <bookCount>
    <title>Data on the Web</title>
    <numberOfAuthors>3</numberOfAuthors>
  </bookCount>
  <bookCount>
    <title>The Economics of Technology and Content for Digital TV</title>
    <numberOfAuthors>0</numberOfAuthors>
  </bookCount>
</counts>
```

User-defined functions can also be used within XPath expressions.

In this case, you're both taking in and outputting a single element. You also have the option to specify multiple elements, or even optional elements. The notation should be familiar to those who have been working with XML long enough to remember DTD's; an asterisk (*) for zero or more nodes, a plus (+) for one or more nodes, and a question mark (?) for zero or one node.

XQueryX

So far, I've focused on the human-readable form of XQuery. But there's one more way to look at XQuery.

XQueryX

One permutation of XQuery that I haven't discussed is the ability to represent an XQuery as an XML document. One of the goals of the XQuery project was to make sure that these queries can be easily read by an application. This will allow developers to create applications that could easily build, edit, or manage XQuery queries.

This part of the project is called XqueryX, and consists of a schema to map the XQuery syntax into XML. For example, consider this simple query (see Listing 57).

Listing 57. Sample query

```
<books>
{
  for $b in doc("bib.xml")//book
  return
       $b/title
}
</books>
```

Mapping it into XQueryX creates this document (see Listing 58).

Listing 58. XQueryX version of Listing 57

```
<?xml version="1.0"?>
```

```
<xqx:module xmlns:xqx="http://www.w3.org/2005/XQueryX"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.w3.org/2005/XQueryX
                                http://www.w3.org/2005/XQueryX/xqueryx.xsd">
  <xqx:mainModule>
    <xqx:queryBody>
      <xqx:elementConstructor>
        <xqx:tagName>books</xqx:tagName>
        <xqx:elementContent>
          <xqx:flworExpr>
            <xqx:forClause>
              <xqx:forClauseItem>
                <xqx:typedVariableBinding>
                  <xqx:varName>b</xqx:varName>
                </xqx:typedVariableBinding>
                <xqx:forExpr>
                  <xqx:pathExpr>
                    <xqx:stepExpr>
                      <xqx:filterExpr>
                        <xqx:functionCallExpr>
                          <xqx:functionName>doc</xqx:functionName>
                          <xqx:arguments>
                            <xqx:stringConstantExpr>
                              <xqx:value>bib.xml</xqx:value>
                            </xqx:stringConstantExpr>
                          </xqx:arguments>
                        </xqx:functionCallExpr>
                      </xqx:filterExpr>
                    </xqx:stepExpr>
                    <xqx:stepExpr>
                      <xqx:xpathAxis>descendant-or-self</xqx:xpathAxis>
                      <xqx:nameTest>book</xqx:nameTest>
                    </xqx:stepExpr>
                  </xqx:pathExpr>
                </xqx:forExpr>
              </xqx:forClauseItem>
            </xqx:forClause>
            <xqx:returnClause>
              <xqx:pathExpr>
                <xqx:stepExpr>
                  <xqx:filterExpr>
                    <xqx:varRef>
                      <xqx:name>b</xqx:name>
                    </xqx:varRef>
                  </xqx:filterExpr>
                </xqx:stepExpr>
                <xqx:stepExpr>
                  <xqx:xpathAxis>child</xqx:xpathAxis>
                  <xqx:nameTest>title</xqx:nameTest>
                </xqx:stepExpr>
              </xqx:pathExpr>
            </xqx:returnClause>
          </xqx:flworExpr>
        </xqx:elementContent>
      </xqx:elementConstructor>
    </xqx:queryBody>
  </xqx:mainModule>
```

Now, this looks overwhelming -- it wasn't meant to be easy for humans to read -- but if you follow it through step-by-step, it does make sense. For example, the portion of the query that consists of `doc("bib.xml")//book` is represented by Listing 59.

Listing 59. A small portion of the expression

```
...
<xqx:pathExpr>
  <xqx:stepExpr>
    <xqx:filterExpr>
      <xqx:functionCallExpr>
        <xqx:functionName>doc</xqx:functionName>
        <xqx:arguments>
          <xqx:stringConstantExpr>
            <xqx:value>bib.xml</xqx:value>
          </xqx:stringConstantExpr>
        </xqx:arguments>
      </xqx:functionCallExpr>
    </xqx:filterExpr>
  </xqx:stepExpr>
  <xqx:stepExpr>
    <xqx:xpathAxis>descendant-or-self</xqx:xpathAxis>
    <xqx:nameTest>book</xqx:nameTest>
  </xqx:stepExpr>
</xqx:pathExpr>
...
```

If you follow this through, you deal with an XPath expression (`pathExpr`) with two steps (`stepExpr`), `doc("bib.xml")` and `//book`. The first part is a filter (`filterExpr`), which consists of a function (`functionCallExpr`), which has a name (`functionName`) and argument (`arguments`). You can similarly follow the second half of the expression through.

XQueryX provides a schema that enables you to re-create any XQuery query as an XML document, as well as stylesheets for transforming that XML back into XQuery.

Summary

XQuery is a more convenient way to access XML information than traditional XPath 1.0 expressions. It includes much greater type checking, data manipulation functions, and a fundamental change from node-sets to sequences. Perhaps the greatest change is the addition of programming-like capabilities that more closely mirror the types of tasks that will be necessary as XML makes further inroads into data storage.

These additions include FLWOR statements, which provide functionality similar to SQL statements, and the ability to create user-defined functions.

Other topics covered in this tutorial include:

- Using DB2's XQuery implementation
- FLWOR expressions and their various permutations
- Changes to XPath with version 2.0
- Sequences
- Advanced programming capabilities built into XQuery