Beginning DB2: From Novice to Professional

Copyright © 2008 by Grant Allen

ISBN-13: 978-1-59059-942-6

ISBN-10: 1-59059-942-X

ISBN-13 (electronic): 978-1-4302-0548-7

ISBN-10 (electronic): 1-4302-0548-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Gennick
Technical Reviewer: Fred Sobotka
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Nancy Sixsmith, ConText Editorial Services, Inc.
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Diana Van Winkle
Proofreader: Linda Seifert
Indexer: Becky Hornyak
Artist: Diana Van Winkle
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 11

Using XML with DB2

Do you remember where you were when Armstrong landed on the moon? What about when the Berlin Wall fell? How about the day you first heard about XML—that it would relieve world hunger, bring global peace, and solve Fermat's last theorem, all before breakfast? Those heady days of the late 1990s were full of dot-com mania, with a liberal dose of XML to grease the wheels of … well … just about anything!

Of course, the great promises never quite materialized, but somehow XML hung on and became the de facto tool for many kinds of information interchange. At about the same time, many people in the database industry predicted the death of relational databases because XML was "the way of the future." Of course, a few wise old heads noted that XML was exactly like the hierarchical databases of 20 and 30 years prior, and they'd quickly ceded the data processing world to the likes of DB2.

For the next 10 years, XML and relational databases did a sort of dance around each other, with vendors—including IBM—slowly moving closer to dealing with the XML as first-class kind of data. With the release of DB2 9, IBM has brought XML into the heart of the database, with its pureXML technology that enables simultaneous storage and processing of relational and XML data with seamless interplay between the two.

There are some great tools supporting XML in the DB2 pureXML features. Some of the related technologies around XML are not as well known as XML or the SQL features in the database. So I'll spend some time covering XQuery and XPath to make sure you're best equipped to work with pureXML, too!

Exploring XML in the Sample Database

Before we dive into the depths of working with the XML and DB2 pureXML technology, it is worth taking a quick tour of the `SAMPLE` database to see how IBM has used its new technology to provide you a fast track to learning about these great new features. You can also see how tools such as the Control Center and Command Editor natively support XML and allow you to work with it easily.

Make sure that you are logged in as instance owner (`db2admin` under Windows, `db2inst1` under Linux if you've followed the defaults) and fire up the Command Editor from within the Control Center. Connect to the `SAMPLE` database:

```
connect to sample

   Database Connection Information
```

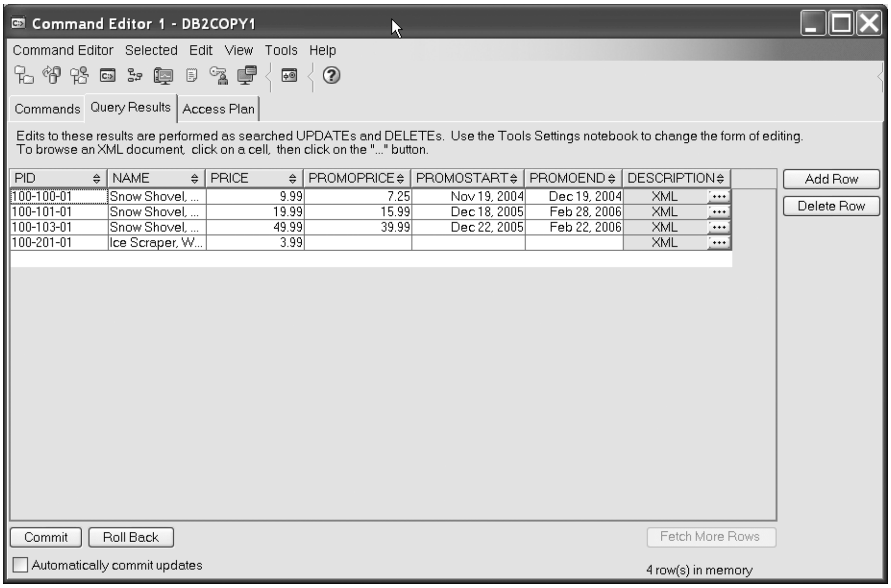
```
 Database server        = DB2/NT 9.5.0
 SQL authorization ID   = DB2ADMIN
 Local database alias   = SAMPLE

A JDBC connection to the target has succeeded.
```

The SAMPLE database includes three tables—CUSTOMER, PRODUCT, and SUPPLIERS—that include XML columns as well as working data that was populated when you created the SAMPLE database. If you query one of these tables using a normal SQL query, you'll see how the Command Editor provides some slick tools for working with your XML data. Try selecting the contents of the PRODUCT table:

```
select * from product
```

The output will appear as shown in Figure 11-1, but you can't see any data in the DESCRIPTION column. Instead, you see the word XML and an ellipsis button (the one with the three dots). That's not because the products lack data for the description; the data is in XML form.


Command Editor 1 - DB2COPY1
Command Editor Selected Edit View Tools Help
Commands
Query Results
Access Plan
Add Row
Delete Row
Commit
Roll Back
Fetch More Rows
Automatically commit updates
4 row(s) in memory

Edits to these results are performed as searched UPDATEs and DELETEs. Use the Tools Settings notebook to change the form of editing.
To browse an XML document, click on a cell, then click on the "..." button.

PID	NAME	PRICE	PROMOPRICE	PROMOSTART	PROMOEND	DESCRIPTION
100-100-01	Snow Shovel, ...	9.99	7.25	Nov 19, 2004	Dec 19, 2004	XML ...
100-101-01	Snow Shovel, ...	19.99	15.99	Dec 18, 2005	Feb 28, 2006	XML ...
100-103-01	Snow Shovel, ...	49.99	39.99	Dec 22, 2005	Feb 22, 2006	XML ...
100-201-01	Ice Scraper, W...	3.99				XML ...

Figure 11-1. *Command Editor results, including XML data*

The Command Editor knows that XML has complex presentation requirements and has a built-in Document Viewer that understands how to traverse data in XML form and present it in a meaningful way. Remember that unlike SQL, XML has concepts such as hierarchy and order, and this is where the Document Viewer really helps. Click the ellipsis button for the first row in the results, and you'll see the Document Viewer spring to life, as shown in Figure 11-2.

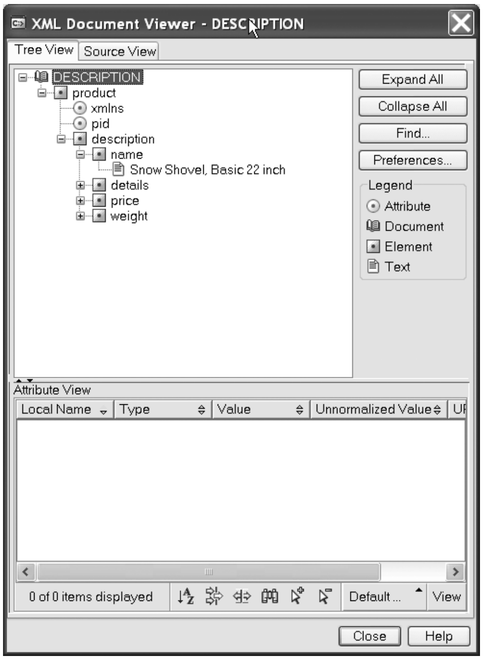


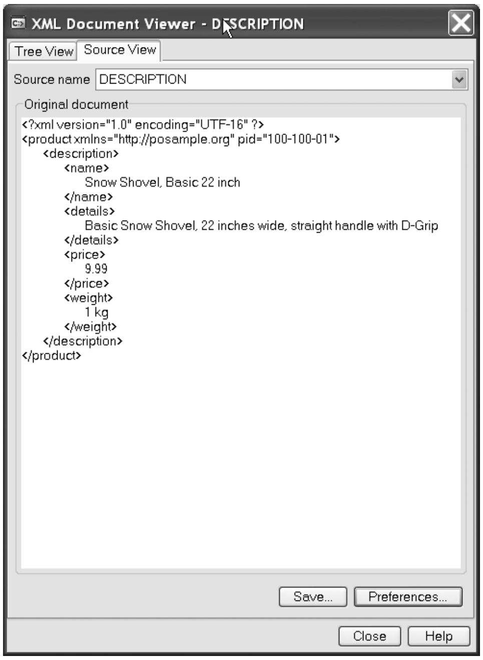
Figure 11-2. *Displaying XML data columns in the Document Viewer*

By default, the Document Viewer shows you the pictorial version of the XML tree, allowing you to see the structure of the data first. In this example, you see that a product's description has a <product> element at the root of the XML, with `pid` and `xmlns` (namespace) attributes. It then has a child element of <description> that in turn has child elements <name>, <details>, <price>, and <weight>. The values for an element aren't shown by default, but simply clicking on the plus sign next to an element name will expose the data for this row, as I did for the <name> element in Figure 11-2.

The Document Viewer also allows you to see the raw XML by choosing the Source view. Click the Source View tab, and the view will change to that shown in Figure 11-3.

If your XML text is displayed in a single line that scrolls off the screen, click the Preferences button and choose the Format Text option. This procedure invokes the parsing capabilities of the Document Viewer and it formats the elements in the much more user-friendly way you see in Figure 11-3.

You'll be pleased to know that the Document Viewer for XML is built in to the Control Center. If you launch the Control Center and drill down to the `CUSTOMER` table in the `SAMPLE` database, you can right-click it and choose Open. You'll see the same output for XML columns here—the code XML and an ellipsis button inviting you to explore the XML.



```
<?xml version="1.0" encoding="UTF-16" ?>
<product xmlns="http://posample.org" pid="100-100-01">
    <description>
        <name>
            Snow Shovel, Basic 22 inch
        </name>
        <details>
            Basic Snow Shovel, 22 inches wide, straight handle with D-Grip
        </details>
        <price>
            9.99
        </price>
        <weight>
            1 kg
        </weight>
    </description>
</product>
```

Figure 11-3. *Viewing raw XML data in the Document Viewer*

■**Note** If you installed the original DB2 9 release, code-named Viper, your Document Viewer will require an extra step to process XML. You'll see a Fetch XML button on any result screen (from either the Command Editor or the Control Center); you have to click it before the XML placeholder and ellipsis button become visible. If you automatically see them and not the Fetch button, you're already benefiting from DB2 9.5 and the great new Viper 2 features!

The Document Viewer is a very handy tool for investigating and exploring individual XML results, but you're certain to want to work with XML data in bulk at some stage. I think you'll agree that having to navigate through dialog boxes for every row of a large set of results might quickly become tedious. As you expect, DB2 offers powerful pureXML features you can use from your favorite query tool and within your application code to work with XML en masse!

Querying Your XML Data

With the introduction of XML into relational databases, there was a need to adapt the query language to enable the hierarchical nature of XML to be exploited. While some attempt was made to add features to SQL, a complementary approach was also developed to enable data manipulation with an XML focus.

The DB2 pureXML support offers multiple ways to work with your data, meaning that you can leverage any existing knowledge you already have. If you're new to the whole idea of mixing XML in your relational database, you have choices about a variety of tools to help you.

There are two main query dialects supported by pureXML in DB2: the XQuery and Xpath dialects.

XML IN THE SAMPLE DATABASE

IBM refines the data in the `SAMPLE` DB2 database with each release. One change that has occurred in the DB2 9.5 release is the inclusion of an explicit XML namespace in all the sample XML data that wasn't present in the 9.1 release. In lay terms, this means the XML is peppered with an additional attribute in the top level of a given schema. It is the namespace attribute, `xmlns`, which appears like this in the data (in this example, a <customerinfo> element from the info column of the customer table):

```
<customerinfo xmlns="http://posample.org" Cid="1000">
```

In these circumstances, all XQuery and XPath queries need to be prefaced with a namespace declaration. If you are using a `SAMPLE` database created with DB2 9.5, every XQuery and XPath example needs to start with the following additional statement, instead of just the keyword `xquery`:

```
XQUERY declare default element namespace "http://posample.org";
```

Because this statement expands every query to a multistatement one, you'll need to set an alternative statement terminator. I recommend using the pipe symbol (|) because it won't conflict with anything else and using the semicolon to separate each individual statement in the unit of work; for example:

```
XQUERY declare default element namespace "http://posample.org";
db2-fn:xmlcolumn('CUSTOMER.INFO')/customerinfo |
```

Using XQuery for XML

XQuery is a language designed to work intimately with the XML in your database. It shares some similar concepts with SQL, but also has important differences to deal with the unique nature of XML data.

I'll start with a discussion of some of the unique capabilities of XQuery. First and foremost, XML has structure, and XQuery has methods of working with this structure. In particular, XML documents have a hierarchy, so one element can be contained within another. XML also has implied order—the sequence in which the elements, data, and so forth appear in the document is part of the metadata about that document. In other words, order is important. This is a distinct difference from relational data and SQL.

To deal with these particular characteristics, XQuery uses a simple slash character (`/`) to allow you to navigate through XML hierarchies and structures. It also provides predicates similar to those used in SQL. Most importantly, DB2 wraps these basic constructs in two pureXML functions: `db2-fn:xmlcolumn` and `db2-fn:sqlquery`. Each function can form the core of an XQuery statement, along with a set of clauses that are colloquially known as the "F-L-W-O-R" commands.

XQuery builds around the foundation functions using clauses similar to those in SQL, but they are slightly different in name and function, as shown in Table 11-1.

Table 11-1. *The F-L-W-O-R Clauses of XQuery*

Clause	Purpose
for	Introduces the alias by which data will be manipulated
let	Introduces working values and variables for data manipulation
where	Introduces additional predicates (similar to the SQL where clause)
order by	Provides ordering rules (similar to the SQL order by clause)
return	Indicates aliases and values to return to the caller (the results)

Understanding XQuery is always easier with some examples, so let's build up your knowledge using the `CUSTOMER` table you've already explored.

First, the most SQL-like technique. You can use XQuery code to invoke an SQL statement that targets XML data. There's no reason you need to approach XML data retrieval this way for simple data retrieval, but there are more complex uses you'll explore later that rely on the ability to do this. Using the keyword `xquery`, you introduce to DB2 that you're about to invoke the pureXML features. The `db2-fn:sqlquery` function allows you to pass a SQL statement to be issued. These results are formatted for ease of reading and flow on the page. The following statement invokes an SQL statement from XQuery, and that SQL statement returns the INFO column, which is the column in the `CUSTOMER` table containing XML data:

```
xquery declare default element namespace "http://posample.org";
db2-fn:sqlquery("select info from fuzzy.customer")|

INFO
----------------------------------------------------------------------------------
<customerinfo xmlns="http://posample.org" Cid="1000">
 <name>Kathy Smith</name>
 <addr country="Canada">
  <street>5 Rosewood</street>
  <city>Toronto</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>M6W 1E6</pcode-zip>
</addr>
<phone type="work">416-555-1358</phone>
</customerinfo>
<customerinfo xmlns="http://posample.org" Cid="1001">
 <name>Kathy Smith</name>
 <addr country="Canada">
  <street>25 EastCreek</street>
  <city>Markham</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>N9C 3T6</pcode-zip>
 </addr>
 <phone type="work">905-555-7258</phone>
</customerinfo>
...
```

The beauty of the db2-fn:sqlquery function is its capability to leverage what you already know. The only caveat, and the key difference between just issuing your SQL directly, is that db2-fn:sqlquery must return only XML data. In its simplest form, it's just like a wrapper around a select statement. To use the equivalent db2-fn:xmlcolumn function, you need to introduce the first two of the F-L-W-O-R clauses: FOR and RETURN. The general syntax looks like this:

```
xquery namespace-clause
for alias in db2-fn:xmlcolumn('table-and-xml-column-name')/optional-predicates
return alias
```

An alias is simply a placeholder variable that allows you to conveniently refer to the XML data as it is refined down to those items you want to see. An alias starts with a dollar sign ($) and can then contain almost any name you like. The table and column name is specific in typical DB2 dotted notation. They are not case sensitive, in line with normal references to tables and columns in regular SQL. The optional XQuery predicates allow you to specify matching criteria, wildcards, elements, attributes, and so forth. Because XML is case sensitive, this part of an XQuery is also case sensitive. This case sensitivity isn't just a factor of DB2 pureXML; you'll encounter it whenever you work with XML.

You can build an equivalent statement to the first db2-fn:sqlquery function as follows:

```
xquery declare default element namespace "http://posample.org";
for $x in db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO')
return $x |
```

Remember to use the schema name appropriate for your database, as you're unlikely to be using FUZZY as your schema. If you created the SAMPLE database as the instance owner, you'll likely be using DB2INST1 under Linux or DB2ADMIN under Windows if you followed the default. Because you were interested in all the data in the INFO column, you didn't need to specify any predicates to refine the result. When you execute this statement, the results look very familiar:

```
INFO
-----------------------------------------------------------------------------------
<customerinfo xmlns="http://posample.org" Cid="1000">
 <name>Kathy Smith</name>
 <addr country="Canada">
  <street>5 Rosewood</street>
  <city>Toronto</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>M6W 1E6</pcode-zip>
</addr>
<phone type="work">416-555-1358</phone>
</customerinfo>
<customerinfo xmlns="http://posample.org" Cid="1001">
```

```
 <name>Kathy Smith</name>
 <addr country="Canada">
  <street>25 EastCreek</street>
  <city>Markham</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>N9C 3T6</pcode-zip>
 </addr>
 <phone type="work">905-555-7258</phone>
</customerinfo>
...
```

No surprises here; you're seeing the same data you browsed through in the Document Viewer. Naturally, you're not always interested in all the data, and XQuery supports the use of predicates, conditions, and similar constructs to refine your queries. The equivalent of selecting columns in SQL is to use the / character and the names of the element you want to see. So if you're interested only in the names of your customers, extend the XQuery as follows:

```
Xquery declare default element namespace "http://posample.org";
for $x in db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO')/customerinfo/name
return $x |
```

Here, you're instructing the XQuery processor to walk through the document and pick out the <name> child elements of the <customerinfo> top-level elements. The results look like this (with a little touchup of the formatting):

```
1
-----------------------------------------------------------------------------------
<name>Kathy Smith</name>
<name>Kathy Smith</name>
<name>Jim Noodle</name>
<name>Robert Shoemaker</name>
<name>Matt Foreman</name>
<name>Larry Menard</name>

  6 record(s) selected.
```

You might immediately think that the logical extension is to select multiple columns. That's where the analogy with SQL stops—or takes a little detour because you're not dealing with tabular data, but with a tree. To fetch elements that are "peers," or from different branches of the XML document, use the let clause of the F-L-W-O-R syntax:

```
Xquery declare default element namespace "http://posample.org";
for $x in db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO')/customerinfo
let $cname := xs:string($x/name)
let $cprovince := xs:string($x/addr/prov-state)
return ($cname, $cprovince) |
```

The xs:string notation is similar to data type casting in regular SQL—you're indicating that the data to be retrieved should be treated as a string. The let clause can leverage the alias that you declare for the matching elements in the for clause, saving you from having to type that full function, table, and element path every time you want to refer to your matched XML. The results are the names and provinces of your customers:

```
1
----------------------------------------------------------------------------------
Kathy Smith
Ontario
Kathy Smith
Ontario
Jim Noodle
Ontario
Robert Shoemaker
Ontario
Matt Foreman
Ontario
Larry Menard
Ontario

  12 record(s) selected.
```

Importantly, the results are not treated as a table of name/province pairs. Instead, the matching elements are returned as the XML is navigated, so first a name will be returned, then its child province(s), then the next city name, and so on. Depending on the complexity of the XML schema, this could mean you get distinctly hierarchical data instead of tabular data. This can take a little getting used to, but after awhile, you'll be picturing trees of data in your mind.

I'll now introduce another of the F-L-W-O-R clauses. The where clause enables you to add predicates and conditions to filter the data being returned. The normal range of options is available—from equality, greater than and less than, to Boolean operators. You can construct a simple where clause for the XQuery to exclude one of your customers. Let's leave Jim Noodle out of the picture for the time being (no offense, Jim):

```
Xquery declare default element namespace "http://posample.org";
for $x in db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO'')/customerinfo
where $x/name != 'Jim Noodle'
return $x/name |
1
----------------------------------------------------------------------------------
<name>Kathy Smith</name>
<name>Kathy Smith</name>
<name>Robert Shoemaker</name>
<name>Matt Foreman</name>
<name>Larry Menard</name>

  5 record(s) selected.
```

You asked the XQuery to seek out all <customerinfo> elements and compare the <name> child element with the text 'Jim Noodle' by using the handy alias $x. You returned the child element <name> from our alias. Quite straightforward, and the logic translates across from SQL quite nicely.

Ordering your data is something that's bound to crop up as a requirement, and the order by clause of the F-L-W-O-R syntax is reminiscent of SQL and fits with the XQuery techniques you already learned. I'll also introduce another of the special functions of XQuery.

```
xquery declare default element namespace "http://posample.org";
for $x in db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO')/customerinfo
where $x/name != 'Jim Noodle'
order by $x/name/text()
return $x/name |
1
----------------------------------------------------------------------------------
<name>Kathy Smith</name>
<name>Kathy Smith</name>
<name>Larry Menard</name>
<name>Matt Foreman</name>
<name>Robert Shoemaker</name>

  5 record(s) selected.
```

The `text()` function returns the actual text of a given element. In this case, you wanted to use that text as the basis for ordering your results. You can go to town on the F-L-W-O-R clauses, building more elaborate and complex commands as your comfort of XQuery grows. For these purposes, I covered the basics of every clause and how to combine them together to query your XML data.

Using XPath Queries for XML

The easiest way to think of XPath queries, in comparison with XQuery style queries, is to strip away the F-L-W-O-R constructs and just think of the bare predicates that invoke the `xmlcolumn` function or other pureXML procedures.

You might be confused because XPath queries are also flagged with the keyword `xquery` when passing the query text to the Command Editor or the Command Line Processor (CLP). To illustrate, here's the equivalent search for customer info that you modeled with the preceding XQuery dialect.

```
xquery declare default element namespace "http://posample.org";
db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO') |
```

The results look remarkably similar to those from the earlier XQuery example:

```
INFO
----------------------------------------------------------------------------------
<customerinfo xmlns="http://posample.org" Cid="1000">
<name>
 Kathy Smith
</name>
<addr country="Canada">
 <street>
  5 Rosewood
 </street>
 <city>
  Toronto
 </city>
 <prov-state>
  Ontario
 </prov-state>
 <pcode-zip>
  M6W 1E6
 </pcode-zip>
</addr>
<phone type="work">
 416-555-1358
</phone>
</customerinfo>
...
```

The key difference between XQuery and XPath is the added flexibility XQuery offers with its FOR, LET, WHERE, ORDER BY and RETURN options. But XPath is powerful in its own right and can perform many of the same tricks.

XPath expressions can access any element or attribute in an XML document stored in DB2. To illustrate traversing to lower-level elements, look at the postal codes for those customers already in the `SAMPLE` database. The approach should be familiar: just append the element names, separated by forward-slash characters, to the XPath statement. The results are formatted to appear a little more readable and concise:

```
xquery declare default element namespace "http://posample.org";
db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO ')/customerinfo/addr/pcode-zip |

INFO
------------------------------------------------------------------------------------
<pcode-zip>M6W 1E6</pcode-zip>
<pcode-zip>N9C 3T6</pcode-zip>
<pcode-zip>N9C 3T6</pcode-zip>
<pcode-zip>N8X 7F8</pcode-zip>
<pcode-zip>M3Z 5H9</pcode-zip>
<pcode-zip>M4C 5K8</pcode-zip>

6 record(s) selected.
```

Voila! A set of postal codes as requested.

Using XPath Predicates

You can construct XPath queries based on the data values using [...] predicates, just as you can with XQuery. Let’s find the city for the customer with `Cid`, an attribute of the <customerinfo> element, of 1004. To do this, you use the @ symbol to lead the attribute name.

```
xquery declare default element namespace "http://posample.org";
db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO')/customerinfo[@Cid=1004]/name |
1
-----------------------------------------------------------------------------------
<name xmlns="http://posample.org">
Matt Foreman
</name>

1 record(s) selected.
```

And you're not restricted to single values. XPath supports the normal Boolean combination for attributes and other criteria. So you can query for two Cid values like this:

```
xquery declare default element namespace "http://posample.org";
db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO')/customerinfo[@Cid=1004 or @Cid=1005]/name |

1
-----------------------------------------------------------------------------------
<name xmlns="http://posample.org">
Matt Foreman
</name>
<name xmlns="http://posample.org">
Larry Menard
</name>

  2 record(s) selected.
```

XML documents have implied order, unlike data in normal relational storage. Your customer, Matt Foreman, actually has two phone numbers listed. You can use a positional predicate to pick the second one. In this query, the first predicate, [@Cid=1004], selects Mr. Foreman by Cid reference, and the second predicate, [2], chooses the second <phone> child element:

```
Xquery declare default element namespace "http://posample.org";
db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO')/customerinfo[@Cid=1004]/phone[2] |

1
-----------------------------------------------------------------------------------
<phone type="home">
416-555-3376
</phone>

  1 record(s) selected.
```

Using XPath Wildcards

XPath supports the asterisk (*) as a universal wildcard for any element. It also supports the function text() to return the actual text of an element. So you can retrieve the text of all the elements below <addr> as follows:

```
Xquery declare default element namespace "http://posample.org";
db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO')/customerinfo/addr/*/text()
```

```
1
----------------------------------------------------------------------------------
5 Rosewood
Toronto
Ontario
M6W 1E6
25 EastCreek
Markham
Ontario
N9C 3T6
...
```

Finally, XPath supports // as the "myself or any of my descendants" placeholder. The customer data illustrates a useful application of this feature. Customers have their own phone numbers in the <phone> element. Customers can also have assistants, who in turn might also have their own phone numbers. Using the // wildcard lets you retrieve all phone numbers at any arbitrary level at or below <customerinfo>. Again, I formatted these results to save a few trees:

```
xquery declare default element namespace "http://posample.org";
db2-fn:xmlcolumn('FUZZY.CUSTOMER.INFO')/customerinfo//phone |

1
----------------------------------------------------------------------------------
<phone type="work">416-555-1358</phone>
<phone type="work">905-555-7258</phone>
<phone type="work">905-555-7258</phone>
<phone type="work">905-555-7258</phone>
<phone type="home">416-555-2937</phone>
...
<phone type="home">416-555-6121</phone>

  11 record(s) selected.
```

More pureXML Features for Querying Data

Naturally, there are more pureXML features than just the `db2-fn:xmlcolumn` function. One of the more useful features is the ability to parse your XML data and present it as a virtual relational table using the `XMLTable` function. Here's an example query to show how this function works:

```
select x.name, x.city
from customer,
xmltable(XMLNAMESPACES(DEFAULT 'http://posample.org'),
 '$cust/customerinfo' passing info as "cust"
 columns
  "NAME" varchar(50) PATH 'name',
  "CITY" varchar(50) PATH 'addr/city'
) as x
```

The XMLTable function takes as a starting point a directive for the XML namespace using the XMLNAMESPACES function. It then nominates a starting element at a particular level of an XML schema, which is provided by an alias to the base table in which the XML data resides. So the from clause is essentially saying you want to construct a virtual table based on the <customerinfo> element that's referenced from the INFO column of the customer table using the alias "cust". You then state what you want the virtual table to look like using normal SQL column names and data types, and use the PATH option to link them to the source elements in the XML data.

Finally, the XMLTable function is a table-typed function, so it returns a table that needs an alias—you used X for this purpose. The select clause is then just like any other, simply listing the columns you'd like to see. The data looks like this:

```
X.NAME           X.CITY
------------------------
Kathy Smith       Toronto
Kathy Smith      Markham
Jim Noodle       Markham
Robert Shoemaker Aurora
Matt Foreman     Toronto
Larry Menard     Toronto
```

So when you need to work with your XML data as if it were in tabular form, XMLTable is the function to call on.

Changing XML Data

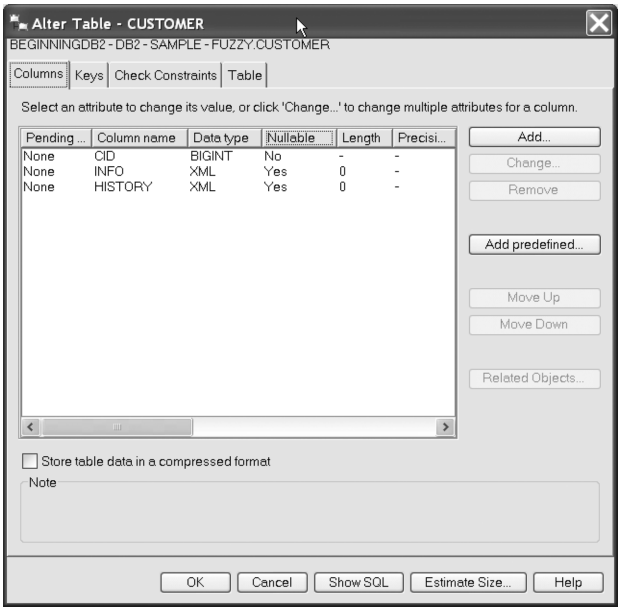
Now that you've conquered the world of XQuery and XPath, you're ready to round out the complementary statements that allow you to add, change, and remove your XML data. Each of them can be related to equivalent action in SQL, and you'll see how the two languages blend to tackle some of these tasks.

Inserting XML Data

There are several approaches to inserting XML into databases such as DB2, with choices revolving around whether you have to compromise your XML to store it in relational structures or whether you're lucky enough to have features such as pureXML that let you have the best of both worlds. The "shredding" of XML is where the information is ripped out of the elements that form XML documents to be placed into one or more relational tables.

Thankfully, pureXML allows true XML storage, so while it supports shredding or decomposition, you will probably find the native XML handling capabilities of greater use and more intuitive after you adopt the XML mindset.

Regular insert statements can be used to enter XML data into tables defined with XML fields. Figure 11-4 shows the structure of the customer table with which you've been practicing XQuery commands.



Select an attribute to change its value, or click 'Change...' to change multiple attributes for a column.

Pending ...	Column name	Data type	Nullable	Length	Precisi...
None	CID	BIGINT	No	-	-
None	INFO	XML	Yes	0	-
None	HISTORY	XML	Yes	0	-

☐ Store table data in a compressed format

Figure 11-4. *XML columns in the table*

Of the three fields, `Cid` is a mandatory integer, and the Info and History fields are of type XML. The `customer` table in the `SAMPLE` database is configured by default to be permissive about the adding of data to the XML fields. You'll examine the more complex options for XML insert shortly, but right now you can actually go ahead and use a simple SQL insert statement to add data, including XML, to this table:

```
insert into customer values (1006, '<my_xml>Hello</my_xml>','<more_xml/>')
DB20000I  The SQL command completed successfully.
```

That's just like dealing with simple text data. (And let's face it, that's *exactly* what XML is—text!) DB2's pureXML features are at work here, though, because even though I haven't yet invoked the advanced features of XML, DB2 is still ensuring that the data inserted into an XML typed field is well formed. If you try to insert data that fails the XML well-formed test, DB2 rejects it:

```
insert into customer values (1006, '<my_xml>Hello</wrong_tag>','<more_xml/>')
SQL16129N  XML document expected end of tag "my_xml".  SQLSTATE=2200M
```

The XML purists among you will know that there are two tests for determining the correctness of XML. The well-formed test, as you've seen, ensures tag matching and closure. The second test is validation, in which XML is compared with a definition for a particular type of XML document. In the early years of XML, these were Data Type Definitions, or DTDs. Later as XML matured, the XML schema standard emerged, allowing the definition of an XML document to itself be described in XML. DB2's pureXML supports validation of XML against both DTDs and XML schema.

To invoke validation, the insert statement is extended using the `xmlvalidate` function. I abbreviated this example to spare you several pages of raw XML data and to keep the structure understandable:

```
Insert into customer (cid, info)
values (1006,
xmlvalidate(source-xml-data according to xmlschema id "http://www.myschema.com"))
```

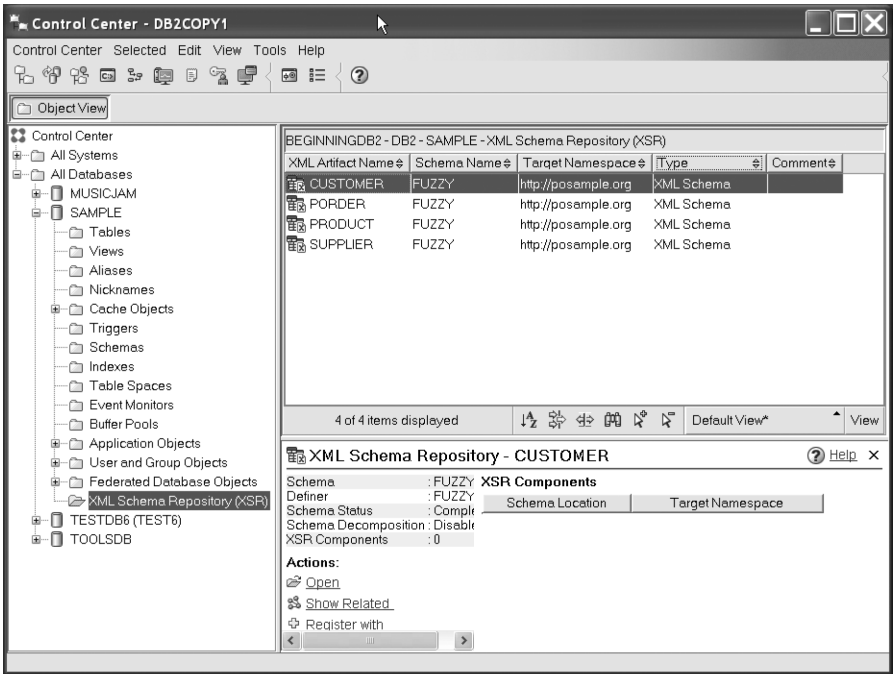
In this example, the validation is invoked using a publicly available schema on the Internet. DB2's pureXML capabilities extend to registering schemata (and DTDs) within DB2, so validation can complete locally. The `xmlvalidate` function is modified to call a schema ID in this case:

```
Insert into customer (cid, info)
values (1006, xmlvalidate(source-xml-data according to xmlschema id "CUSTOMER"))
```

Where do these schemata reside? I'm glad you asked.

XML Schema Registration in DB2

Each DB2 database includes support for XML schema objects, in the XML Schema Repository, or XSR. In the Control Center, under the object types for your `SAMPLE` database, you'll see the XSR listed as the last folder, as shown in Figure 11-5.



BEGINNINGDB2 - DB2 - SAMPLE - XML Schema Repository (XSR)

XML Artifact Name	Schema Name	Target Namespace	Type	Comment
CUSTOMER	FUZZY	http://posample.org	XML Schema	
PORDER	FUZZY	http://posample.org	XML Schema	
PRODUCT	FUZZY	http://posample.org	XML Schema	
SUPPLIER	FUZZY	http://posample.org	XML Schema	

XML Schema Repository - CUSTOMER

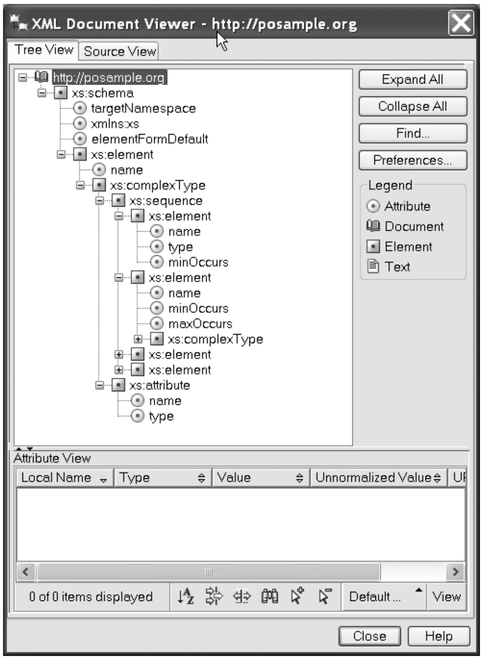
Schema	: FUZZY
Definer	: FUZZY
Schema Status	: Comple
Schema Decomposition	: Disable
XSR Components	: 0

XSR Components

Schema Location	Target Namespace

Figure 11-5. *The XSR in the Control Center*

Open the XSR folder; you'll see XML schemata registered. The CUSTOMER schema is the one in which you're interested, so right-click it and choose View Document. Figure 11-6 shows that your old friend the XML Document Viewer is called in to service to display the schema.



```
http://posample.org
  xs:schema
    targetNamespace
    xmlns:xs
    elementFormDefault
    xs:element
      name
      xs:complexType
        xs:sequence
          xs:element
            name
            type
            minOccurs
          xs:element
            name
            minOccurs
            maxOccurs
            xs:complexType
          xs:element
          xs:element
        xs:attribute
          name
          type
```

Figure 11-6. *Viewing an XML schema in the Document Viewer*

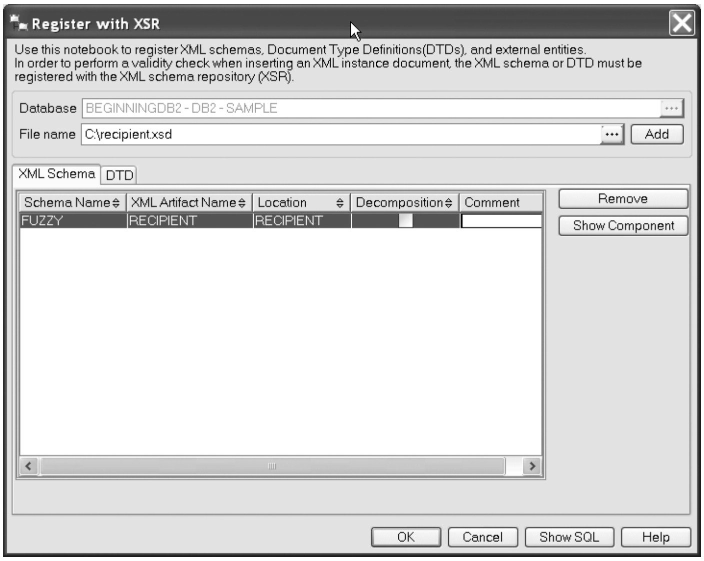
Registering a new XML schema is a quick process, and you'll use the CUSTOMER schema as a shortcut. Switch to the Source View tab of the Document Viewer and click the Save button, which prompts you to save the text file of the CUSTOMER schema. Call the file `recipient.xsd`. Once the file is saved, click Close to close the Document Viewer. Open the file `recipient.xsd` using your favorite text editor (Notepad, KWrite, and so on). The third line of the file reads as follows:

```
<xs:element name="customerinfo">
```

Change it to read as follows:

```
<xs:element name="recipientinfo">
```

Congratulations, you just defined a new XML schema. Okay, in real life you might put more effort into it, but because the discussion is about DB2, not XML schema design, it works for your purposes. Return to the Control Center; from the XSR Folder, right-click and choose the Register With XSR option. The dialog box shown in Figure 11-7 appears.



Use this notebook to register XML schemas, Document Type Definitions(DTDs), and external entities.
In order to perform a validity check when inserting an XML instance document, the XML schema or DTD must be registered with the XML schema repository (XSR).

Database: BEGINNINGDB2 - DB2 - SAMPLE
File name: C:\recipient.xsd

Schema Name	XML Artifact Name	Location	Decomposition	Comment
FUZZY	RECIPIENT	RECIPIENT		

Figure 11-7. *Registering a new XML Schema with the XSR*

Click the ellipsis button and browse to find the `recipient.xsd` file you created. Make sure that you click the Add button to add this file's schema to the one you'll register. You then need to enter values for XML Artifact Name and Location. Enter **RECIPIENT** for both. Now click the OK button in the dialog box, and the registration process will commence.

You might be presented with an error (not a warning) that the schema doesn't map any attributes to a table, but the schema will actually be loaded into the XSR, and you can further refine it and its relationship to your tables and XML fields at a later point. That process takes you a little beyond the scope of this book, so I'll stop the coverage of schema registration there.

Updating XML Data

Changing XML data structures is one of the areas in which developing XML data-manipulation standards have been a little lacking, but DB2 is leading the way with pushing the standard forward. To be brief, to update a column with XML data type, until recently you had to replace the whole XML document for that row and column; you couldn't just alter the data within one element.

The good news is that there are simple ways to update your XML data. The first is an old-fashioned SQL update, just like this:

```
update customer set info = '<my_xml>Hello</my_xml>' where cid = 1006
```

Again, DB2 performs a test for well-formed XML, so there are pureXML features helping you even with this straightforward technique. The second approach is to use the pureXML `XMLPARSE` function to take a given string and convert it to the XML data type for insertion. Think of this as an explicit conversion instead of the implicit one used by the standard SQL update statement:

```
Update customer
set info = xmlparse (document '<my_xml>Hello</my_xml>' preserve whitespace)
where cid = 1006
```

Forgive the simple structure of the XML here, but it allows you to concentrate on the syntax and semantics instead of on pages of XML text.

DB2 9.5 is the first database to support the new in-place update standard, allowing you to make a change to the elements of an XML column without the need to totally replace it. This saves a huge overhead in resources such as logging, extracting, and reinserting the same data. The general syntax is as follows:

```
update table-name
set column-name = xmlquery
( '[transform] copy $new-XML-placeholder := $XML-column-placeholder
   modify do replace value of $new-XML-placeholder/element/... with value
   return $placeholder-for-new-XML ')
[where normal-SQL-criteria];
```

While that looks somewhat more complex than a regular SQL update statement, the benefit comes from the capabilities of the modify option you see used in the xmlquery invocation. This modify clause does the hard work of making the updates in place, taking its instruction from the English-looking syntax options that follow it, such as “replace value of ... with ...” That should be quite understandable.

An example will help clarify how the command works. Update your previously updated data using this new method:

```
update customer
set info = xmlquery('transform copy $mynewxml := $INFO
 modify do replace value of $mynewxml/my_xml with "Hello Again"
 return $mynewxml')
where cid = 1006
```

Deleting XML Data

There are no special standards for deletion based on XML properties—element text, attributes values, and so on. You can base your delete statements on one of the traditional relational columns of your table; for example:

```
Delete from customer where cid = 1006
```

You can also use other XML functions, such as `XMLEXISTS`, to evaluate XML-based criteria to help you determine data to be deleted:

```
delete from customer
where xmlexists('$doc/my_xml/text()="Hello" ' PASSING info AS "doc")
```

Your criteria might well be much more complicated, but the principle is the same. Identify what you want to delete by using the SQL or XQuery capabilities that best help you target it.

Summary

You now have a firm grounding in handling XML data within your DB2 database, including the capabilities of the pureXML functions and the utility of XQuery and XPath queries. Because XML will be more prevalent in all forms of data and database management in the future, this knowledge should be useful in your dealings with DB2 and beyond.