Use XQuery from a Java environment

Search documents with XQuery from Java applications

Article by: Brett D. McLaughlin, Sr. (brett@newInstance.com), Author and Editor, O'Reilly Media, Inc.
April 29, 2008
View the original article

XML data format can be hard to search, but with the fairly recent introduction of the XQuery API, XML searches are now flexible and easy to perform. For Java™ programmers who work with XML documents using SAX, DOM, JDOM, JAXP, and more, the XQuery API for Java is a welcome addition to the programmer's toolkit. Now the power of XQuery is available to Java programmers without resorting to system calls or unwieldy APIs, all in a Sun-standardized package.

SQL databases, XML data, and queries, oh my!

While the world of programming—and particularly Java programming—seems to increase, the number of standardized choices is growing as well. In other words, more and more APIs blessed or approved by Sun are available. The result of this standardization is that an increasing number of developers are branching out beyond their core competencies, and learning new technologies.

High on the list of interesting and worthwhile tools and APIs to master are those that deal with data. No matter how cool or clever an application, it's ultimately only as useful as its ability to work with data. And, while the number of APIs constantly expands, the popular and commonly used number of data formats steadily decreases. While some data managers still use object-oriented database management systems, or XML-driven databases, relational databases (RDBMSs) have weathered that storm, and still seem the choice of most data managers. That leaves the Java developer with JDBC (for database connectivity) or perhaps JDO (Java Data Objects) to interact with SQL databases.

Data not in databases has also almost all standardized on XML as a data format. XML is robust, albeit verbose, and there are perhaps more APIs for working with XML than any other non-Java medium in the language. Whether it's parsing, data binding, or transforming, if your application can't deal with XML, then it's considered limited and perhaps even a bit behind the times.

These two seemingly unrelated facts—the propensity for data to live in SQL databases and the popularity of XML for all data outside of a database—has created some unique problems, though. SQL databases are easy to query; XML documents are not. Consumers expect to be able to search through data easily, and while this works nicely with data in databases, it's not so trivial with data in XML documents. Obviously, taking XML-formatted data and dumping it into a database just to make searching easier is the wrong approach. And that's where XQuery—and as a corollary, the XQuery API for Java (XQJ) comes in.

SQL databases versus all the others

When this article refers to queries and databases, it assumes a SQL database—also referred to as a relational database. However, there are some great applications for XML databases, and even object databases.

If you're interested in the world of XML databases, you can check out DB2® Express-C, available for free download . Most notably, XML databases make conversion between XML documents and relational data unnecessary. Also of relevance to this article is that XQuery actually becomes your query language for your database, since XML databases store data as XML data.

XQuery: A three-part technology

XQuery, at its simplest, is a language used to define searches of XML documents. Much as SQL gives specific meaning to SELECT and FROM—in a particular context—XQuery defines a meaning for forward slash (/0 and at-sign (@), as well as a host of other keywords and key-characters.

At its heart, though XQuery is made up of three components:

1. The XPath specification: a means to select zero, one, or multiple nodes in an XML document
2. Additional syntax to select a particular XML document, and to add selection criteria to the nodes returned from an XPath
3. An API—like XQJ, the XQuery for Java API—that evaluates XQuery expressions in a specific programming language

Frequently used acronyms

- API: application programming interface
- DOM: Document Object Model
- GUI: Graphical User Interface
- IDE: Integrated development environment
- JAXP: Java API for XML Processing
- SAX: Simple API for XML
- W3C: World Wide Web Consortium
- XML: Extensible Markup Language

To really master XQuery, you need a solid grasp of all three of these components. For the Java programmer, that obviously means learning XPath, learning the additional XPath syntactical constructs, and then wrapping all of this up in a Java-based API to issue XQuery expressions to an XML document.

The good news is that the XPath and XQuery syntax is fairly intuitive. If you've ever navigated a directory structure in a UNIX® shell, a Mac OS X terminal, or a DOS window, you're way ahead of the game. Add to that basic use of operators like less than (<), greater than (>), and equals (=), and you're over half of the way to being an XPath pro.

XPath Basics

XQuery actually depends almost entirely on another XML specification, the XPath spec. XPath isn't about functionality as much as defining a way to create *paths* that refer to parts of an XML document. For example, the XPath /play/act/scene translates to all of the scene elements nested within an act element, nested within the play root element.

XPaths are relative

Basic XPaths use element names and forward slashes. And by default, an XPath begins with your current location in an XML document. So if you use the DOM, for example, and navigate to a speech element, and then you issue the path speaker, that evaluates to any speaker elements within the speech element that is your current location. So an XPath evaluates relative to your location in a document.

Moving away from the current node

To move to the root of a document, precede your path with a forward slash. So /play looks for the root element called play, no matter where in a document you are. And you can select a parent element of the current element with ../. In fact, this should start to look a lot like a directory structure. So the path ../../personae/title would go two levels up from the current element, then look

for a personae element, and then a title element nested within that.

Selecting attributes

You can select a lot more than just elements. Consider an XPath like this: /cds/cd@title. This returns all attributes named "title" that are on cd elements, nested within a root element named cds.

Remember that the @ syntax doesn't return the value of an attribute, but the attribute itself. So @isbn would select all attributes named isbn, not the *value* of those attributes. Additionally, in XPath, the term attribute refers to the attribute's name and value.

Selecting text

Just as you can select elements and attributes, you can select the text within elements. If your XPath ends in an element name, like /cds/cd/title, then you're selecting elements—and those elements do *not* include the text within those elements. However, if you want the text within an element, use the text() syntax. So for all the textual titles of a set of CDs, you can use a path like /cds/cd/title/text(). So that path doesn't give you any elements; it gives you the text within the indicated elements.

Text is nested within elements

The common way to think about an element and the text within that element is to consider the element as "having text." You'll often see things like "the value of the title element is 'You Can't Count On Me.'" But that's incorrect; instead, text is nested within an element. You can think of text as belonging to an element, but certainly not the same way that an attribute has a textual value.

An even better way to think about it is to consider that text is truly just nested within an element. That the element is the parent of the text, rather than the value of an element being the text nested within it.

XPath selects sets of nodes

One of the keys to effectively using XPath is to realize that an XPath always evaluates to a *node set*. That set might have zero, one, or multiple nodes, but the result of an XPath is *always* a set. That flies in the face of what most people think when they write XPaths: that the path returns an element, or an attribute, or some text. However, that's not at all the case.

If you use the DOM, you've thought of nodes before. In the DOM, everything in an XML document—elements, attributes, text—is a node. An element is an element node; an attribute (and its value) is an attribute node; even text within an element is a text node. So a path like ../../personae/title, which ultimately results in selecting title elements, actually returns a node set. The set might have zero (there are no matching elements), one, or more nodes. In this case, all of those nodes will be elements named "title."

As your paths get more complex, and potentially select wider sets—perhaps including attributes and elements at the same time, or text and elements—those paths still ultimately are just choosing node sets. Keeping that in mind is key to using XQuery properly. By using XPath, you select a set of nodes; and with XQuery, you then typically select a subset of those nodes with a search criteria, or potentially joining multiple sets of nodes and then applying a search criteria. By keeping in mind that a set can have more than one type (elements, text, or attributes), you'll be able to write those paths a lot better, and ensure they evaluate to what you want.

Becoming more selective with XPath

You've seen how to select sets of nodes based on the names of those nodes (in the case of elements

and attributes) as well as the parents of those nodes (in the case of text, or when you select all child nodes of a given parent). On its own, that's pretty powerful; but XPath offers quite a bit more in terms of selectivity, all using what's called a *predicate*.

Predicate Basics and Syntax

A predicate is a partial expression that can be applied to an existing node set. Predicates are placed with square brackets, [and]. They're applied to the node set indicated by the path to the left of the predicate. For example, consider the path /cds/cd; this selects all cd elements within the root element named "cds." Then supposed you want the first CD; you can use a predicate to get that CD, like this: /cds/cd[1]. This returns the first of the nodes selected by the path /cds/cd.

Microsoft® Internet Explorer® (and some other browsers) got it wrong

The XPath spec indicates that the predicate [1] refers to the first node in a set; in other words, node sets in XPath are indexed starting with the number 1. However, most versions of Internet Explorer mistakenly implemented XPath with a zero-based index, where [1] refers to the *second* item in the array, and [0] refers to the first. You'll need to test this within your own browser, but the correct action is for the first item in a set to be indexed with [1], not [0].

Predicates can be applied to any node set

Remember, a predicate applies to the node set to the left of the predicate itself. However, that doesn't mean that a predicate can only appear at the end of a complete XPath. Consider that an XPath itself is really a collection of paths, each returning a node set, and later portions of the path refining or acting on that set. So /cds/cd/title is really three paths:

1. /cds, which returns the root element, named "cds" (a set which contains one element node)
2. cd (relative to the previous node set), which returns all cd elements nested within the previous node set
3. title (again relative to the previous node set), which returns all title elements nested within the previous node set

A predicate must be applied to a node set, but it can be applied to *any* node set. So here's a completely legal path: /cds[1]/cd[2]/title[1]. This selects the first of the node set selected by /cds; the second of the nodes selected by /cds[1]/cd; and then the first of the nodes selected by /cds[1]/cd[2]/title.

Note: *Parts of this path are essentially meaningless; for example, selecting a root element using / and then applying the [1] predicate will always return the first (and only) element in the set. The only time this predicate won't return an element is if the node set itself is empty, when the root element indicated is not named the same as a document's actual root element. But, for illustration purposes—and from a technical standpoint—there's nothing wrong with the XPath itself, or the predicates used.*

Going beyond numeric indices in a predicate

Of course, an API that only let you reference sites with numeric values would be pretty limited. You're forced in that model to always know exactly where in the set the items you want are. XPath offers a lot more than that, though. For starters, you can use the last() function in your predicates to select the last item in a set, no matter how many total items are in the set. So /cds/cd[last] selects the last cd in a document.

You can also select all the items that are less than a certain position, or greater than a certain position, using the position() function. The position() function returns the position in a set of a given node. For example, suppose you want the first 5 CDs; you could use the path /cds/cd[position()<6].

This selects all the nodes where position() evaluates to less than 6.

Selecting nodes based on data

Finally—in this short introduction to XPath, although not in a complete discussion of XPath—you can select nodes based on child elements of the nodes, or on attributes of the node. Just as each progressive portion of an XPath is based on the set of nodes determined by the preceding path, the predicate of a set is based on the set that you apply it to. Add to that a predicate's ability to use operators like less than (<) and greater than (>), and you can be quite selective based on data within selected nodes, not just the position of those nodes within an overall set.

So suppose you want all CDs that have an attribute named "style" with a value of "folk." You'd use an expression to first select all the CDs, and then compare the style attributes of those CDs to the value "folk." In XPath, that looks like /cds/cd[@style='folk']. This is pretty self-explanatory based on what you've seen so far. First, a set of nodes is selected through /cds/cd. Then, the attributes of each node named "style" are isolated using the partial predicate @style. Just as discussed earlier, preceding a name with @ indicates an attribute. And that attribute is assumed to be relative to the set of nodes already chosen (in this case, all the cd elements). Then the value of these attributes is compared to a string, "folk." The ones that have matching attributes are returned; everything else is tossed out of the result set.

The same can be done with nested elements of the set selected. Suppose you have a document structured like that in Listing 1.

Listing 1. Structure of a sample CD listing document

```
<cds>
<cd style="some-style">
<title>CD title</title>
<track-listing>
<track>Track title</track>
<!-- More track elements... -->
</track-listing>
</cd>
<!-- More CDs... -->
</cds>
```

Now suppose you want all CDs with 10 or more tracks. What you want to do is select all the CD elements, which involves this path you've seen a number of times: /cds/cd. Then, using a predicate, you want to get a count of a particular node set relative to the set you've chosen; in this case, you want to select all of the track element nodes, which are nested in a track-listing element, nested within the node set you're actually wanting to have returned. Finally, you want to apply a count of those nodes, which XPath lets you conveniently do with the count() function. Last, but not least, you need to compare that count with a number, here, 10. Put all this together, and you get a path and predicate like this: /cds/cd[count(track-listing/track) >= 10].

A note on the contradictions within XPath predicates

If you've read carefully, you might note some inconsistencies in how XPath treats element text and attributes here. Earlier, I talked about an attribute node being both the attribute and its value, treated as a single unit of information. However, in a predicate, an expression like @type refers not to the entire type attribute node, but to just the value of that attribute. That allows you to compare that value to other values (as in @type='reggae').

In the same way, you can reference the text in an element within a predicate like this:

/cds/cd[title='Revolver']. Here, the value of the type element, nested within cd, is compared to the value "Revolver." And, just as in the case of attribute nodes, some of the standard rules of thinking about elements are broken. In general, an element node is the parent of a text node; but here, the predicate essentially de-references the text within an element.

These slight aberrations in the rules aren't a problem, as long as you know they exist, and can switch between the two approaches to thinking about elements and attributes. You just need to always be clear when an attribute and its value are considered as one node, and when you reference the value of an attribute; similarly, know when an element contains other text nodes, versus when its text can be compared to another value.

Adding XQuery to the mix

XPath is incredibly powerful, but has some limitations. First and foremost, it's largely suited for static data. In other words, you construct an XPath query for a specific document, with a specific set of data that you're comparing elements, attributes, and text to using predicates and XPaths. Additionally, XPath doesn't have any control structures (like if/else statements) or the ability to perform processing beyond simply comparisons.

To be fair, these limitations aren't a big deal for most non-programmers. But take a Java (or C#, or Python) programmer, with the power of an entire programming language, and they'll quickly come up with ideas for searches in XML documents well beyond what XPath alone offers. That, of course, is where XQuery comes in.

Begin by selecting your document

One of XQuery's little-used, but most important features is its ability to indicate the document to apply an XPath to. So where, in XPath, you'd apply a path like /cds/cd[title='Revolver'] to a specific document, you can indicate the document in your actual XQuery using the doc() function. So if you wanted to search *catalog.xml*, you'd use the XQuery expression doc("catalog.xml")/cds/cd[title='Revolver'].

The power of that little function is the ability to write code that selects a document programmatically (perhaps based on user input), or to loop through a set of documents (maybe all the iTunes catalogs on a network), and apply the statement to each easily.

XQuery and FLWOR

Of course, XQuery offers a lot more than simple runtime document selection. It's most powerful when you harness what's often called its FLWOR capabilities. FLWOR is an acronym for "for, let, where, order by, return." These are all clauses you can use in your XQuery expressions to get more precise results.

For SQL veterans, you should already begin to feel a little more comfortable. WHERE and ORDER BY are both common parts of SQL queries. And for programmers, the term for should look familiar. Here's the brief rundown on what FLWOR clauses do:

It's not a flower

It's unclear why the powers-that-be who are in charge of acronyms chose FLWOR instead of FLOWR; the first is impossible to say, while the second would sound like "flower." The most obvious reasoning is that in many expressions, the ordering of clauses is for, then where, then order by, and then return. But that leaves out let, which can appear before or within an XQuery expression.

The acronym is FLWOR, because the group that standardizes XML specifications (the W3C) chose it, but you'd do well to not use FLOWR and look foolish in front of your other XML-speaking friends.

- **for**: You can use for to take a node set and iterate over it. In many ways, for is the assignment of a variable to the current value in a node set, so you can operate on that variable.
- **let**: You assign values to variable with let, although (as you'll soon see) you won't use let as often as you use the other FLWOR clauses.
- **where**: where lets you apply selection criteria to a node set, beyond what XPath offers. Of course, in many queries, you'll see that where isn't doing more than XPath; it's just moving the predicate on an XPath to a different place.
- **order by**: The order by clause doesn't change data, or filter it; it just applies an ordering to a result set, and allows you to sort values based on something other than the location used in the XPath itself.
- **return**: Using a return clause lets you operate on a node set, but then return (in results) something other than that node set. You might want to select a set, order and filter it, and then return only the child elements of the results; return is the key that makes that possible.

Let's take a little more in-depth look at each of these clauses.

Using for clauses

The for clause is used almost identically as it is in programming languages like Java and C#. Here's the format of the clause:

```
for $variable-name in XPath
...
```

The variable name can be any normal identifier, like x. Normally, variables are best named by their use (like firstName or title), but as this variable is essentially a loop counter, using single letters is also fine.

The XPath can be anything you want. /cds/cd is a perfect example. So you might have something like this:

```
for $cd in doc("catalog.xml")/cds/cd
...
```

That's it. Now the variable $cd will take on the value of each node returned by the XPath /cds/cd. Here, the ... represents the rest of the XQuery expression, which I'll get to next.

For programmers, this statement is really no different from this:

```
for (int i = 0; i <cdArray.length; i++) {
CD cd = cdArray[i];
// Process CD
}
```

Or, for lists:

```
for (Iterator i = cdList.iterator(); i.hasNext(); ) {
CD cd = (CD)i.next();
// Process each CD
}
```

Assigning variables using let

Putting aside the XQuery above for a moment, the let clause is used to assign variables. You've already seen in XQuery that a variable is defined by putting a dollar sign ($) in front of an identifier name. You most often will use variables in XQuery as shown above, though, where the variable is created implicitly by a for clause, rather than explicitly with a let clause.

However, suppose you want to use an explicit variable. You could do something like this:

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd
...
```

Here, the document to search within is assigned to a variable. Assignment in XQuery uses :=, which —unless you've used Pascal in your youth—might look a bit odd. Putting this in a more realistic context, though, you might have a function that iterates over a list of XML documents, and assigns to $docName each of those document names in turn. You can then select each cd element in *each* of the documents in that larger list, and process each in the same way.

Completing a query with the return clause

Let's jump out of the ordering imposed by FLWOR for a moment, in order to complete a query. So far, you've got:

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd
...
```

Next up is returning something. This query selects all the cd elements, but returning that isn't very useful—even though it's exactly the node set that's desired. Instead of returning those elements, suppose you want something more identifiable; perhaps the title of each CD, stored in the title element. That's where return comes in.

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd
return $cd/title/text()
```

Here, the selection of all cd elements is made. Then, each of those resulting nodes is assigned to $cd in turn. Finally, the return clause returns not that element itself, but a child element named "title," which contains the target data.

Be sure *not* to make a simple mistake and write your query like this:

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd
return /cds/cd/title/text()
```

This has three significant problems:

- First, it defeats the purpose of XQuery; you just return an XPath location, rather than do any querying.
- Second, you return data regardless of doc($docName), which indicates which document to select the CDs from.
- Finally, and most importantly, this will ignore any filtering or ordering you perform on the node set returned by the for clause.

You're about to build some of that next, so this will become more obviously important shortly. For now, though, always ensure you have the variable defined in your for clause reappearing in your return clause. That simple rule of thumb will help you ensure your queries behave like you expect.

Get selective with where

Using where allows you be more selective with XQuery. Using a where clause in XQuery works just as it does in SQL; you add the where clause to your selection in order to refine the result set. Here's a very simple example:

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd
where $cd@type = 'reggae'
return $cd/title/text()
```

This returns all the titles of the reggae CDs. There's little here to explain; the where clause is pretty straightforward. But you can apply more complex conditions with and:

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd
where $cd@type = 'reggae'
and count($cd/track-listing/track) > 10
return $cd/title/text()
```

This returns all the reggae CDs with more than 10 tracks.

One more important type of where clause usage is where you' essentially perform a join. Suppose you have an XML file set up something like Listing 2.

Listing 2. Expanded structure of a CD listing document

```
<cds>
<cd style="some-style">
<title>CD title</title>
<artist id="289" />
<track-listing>
<track>Track title</track>
<!-- More track elements... -->
</track-listing>
</cd>
<!-- More CDs... -->

<artists>
<artist id="289">
<firstName>Bob</firstName>
<lastName>Marley</lastName>
</artist>

<!-- More artist elements -->
</artists>
</cds>
```

This expands on the XML structure shown in Listing 1. It adds artist elements, identified by an id attribute, and related to each CD by at least one artist element nested within each cd element.

With XQuery, it's possible to perform what amounts to a join between CDs and their artists. Here's an example:

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd,
$artist in doc($docName)/cds/artists/artist
where $cd/artist/$id = $artist/$id
and $artist/lastName = 'Marley'
return $cd/title/text()
```

Two things go on here. For the first time, you see that a for clause can define more than one variable. Instead of just identifying CDs, this statement also defines $artist, so it's easy to work with the set of artist elements.

Then, the where clause essentially joins CDs with their artists with this line: where $cd/artist/$id =

$artist/$id. Remember that this matches each CD with each artist, producing the equivalent with a SQL join. Then, further selection is defined: $artist/lastName = 'Marley'. So this selects all the artists who have a last name of "Marley." But remember, there's also a join here, and the return clause returns the CD titles. So you get all the titles of the CDs who have an artist whose last name is "Marley."

This is where XQuery really comes into its own. You can perform complex SQL-like joins and selections, all applied to XML documents (many of which probably weren't structured with advanced searching in mind).

Ordering your node sets

If where works *mostly* like its SQL counterpart, then order by works *exactly* like its SQL counterpart. You can order your results by anything you can reference using XPath:

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd,
$artist in doc($docName)/cds/artists/artist
where $cd/artist/$id = $artist/$id
and $artist/lastName = 'Marley'
order by $cd/release/@date
return $cd/title/text()
```

Here, the returned CD titles are ordered by the date attribute on a child element of each CD, release. By default, sorting is done in ascending order; you can make that explicit if you want, though:

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd,
$artist in doc($docName)/cds/artists/artist
where $cd/artist/$id = $artist/$id
and $artist/lastName = 'Marley'
order by $cd/release/@date ascending
return $cd/title/text()
```

And of course you can sort in descending order. In this case, that would return the most recently released CDs first:

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd,
$artist in doc($docName)/cds/artists/artist
where $cd/artist/$id = $artist/$id
and $artist/lastName = 'Marley'
order by $cd/release/@date descending
return $cd/title/text()
```

Note: *If you don't use a schema or XQuery processor that knows that the date attribute is a date type, you can get errors with this statement. Worse, it might get ordered by treating the dates as textual data, and alphabetizing them. Almost all modern processors will handle dates, though, so this is rarely a concern.*

You can also include more than one criteria for sorting. For example, the current expression returns all CDs by any artist with a last name of "Marley" (not just Bob), and interleaves them when it sorts by release date. You could refine the expression to sort by the artist's name, and then release date:

```
let $docName := 'catalog.xml'
for $cd in doc($docName)/cds/cd,
$artist in doc($docName)/cds/artists/artist
where $cd/artist/$id = $artist/$id
```

```
and $artist/lastName = 'Marley'
order by $artist/firstName, $cd/release/@date
return $cd/title/text()
```

Note that sorting by $artist/lastName before $artist/firstName is redundant, since the results all have the same last name value.

Adding Java technology to the mix

It might seem like you've read a lot of introduction just to get to the part on using XQuery from a Java environment. However, most programmers who pick up XQJ (the acronym frequently used for Java's XQuery API) are at best passingly familiar with XPath and XQuery. Now that you've learned a bit more than the basics, you're ready to use these expressions from your Java programs.

The XQuery spec is vendor-neutral

The XQuery for Java API is under Sun's auspices, being developed as part of the Java Community Process, JSR 225. The spec itself involves several different vendors (from Sun and Nokia to BEA, Oracle, and Intel), as well as a few key individuals (like Jason Hunter, of servlets, JDOM, and XML fame). As such, it's a pretty good hammering out of ideas in an environment that isn't going to be tied down to one particular database vendor or XML product-maker.

...but XQJ implementations are not

Unfortunately, there's no Sun-standard implementation of XQJ yet. Most of the vendors on the expert group work for companies that provide an XQJ implementation with their product, and that means you'll deal with some vendor-specific issues. Of course, for long-time XML veterans, this is no different than the XML parser and XSL processor wars that characterized much of the early 2000s. In time, XQJ will standardize, and Sun will almost certainly release its own version of XQJ, or a wrapper API for XQJ implementations that functions like JAXP does for XML parsers and XSL processors.

Getting an XQJ implementation

The easiest way to begin with XQJ is to download a free trial from DataDirect. You have to fill out a rather annoying form, but then you're set for plenty of time—certainly long enough to work through this article. Visit the DataDirect XQuery download site. You'll also have to enter a database you want to access—even if you select the XML Documents Only option. Wade through these options, and you'll get an e-mail with instructions about where to download the JAR file, named datadirectxquery.jar.

Expanding the JARred JAR

The installation process is a bit confusing; first, you need to expand the downloaded datadirectxquery.jar. You can do this using the jar command. But before you do that, you should create a directory for the installation, and expand the JAR file into that directory. Then, you can run the jar command:

```
[bdm0509:~/Desktop] mkdir xqj
[bdm0509:~/Desktop] cd xqj
```

```
[bdm0509:~/Desktop/xqj] jar xvf ../datadirectxquery.jar
inflated: XQueryInstaller.jar
inflated: ddxqj.jar
inflated: ExtensionTool.jar
inflated: Readme.txt
inflated: 3rdPartySoftware.txt
inflated: Fixes.txt
inflated: installer.properties
```

Run the installer

Now, open up the new directory that you created and double-click on the XQueryInstaller.jar file. Assuming you installed Java on your system, this will launch the GUI installer.

You'll be prompted to select the trial or licensed version; select trial for now. Next, you'll need to select an installation directory. Make sure you have permissions to write files to the directory that you enter here. I selected /usr/local/java/xqj on my system, ensuring first that I could write to the /usr/local/java directory. The installation will create the last sub-directory for you—in this example, that's xqj —and then put the DataDirect XQuery files in that directory. Finally, let the installation run, and click Finish.

Once done, navigate to your new directory and check out its contents; they should look like the listing shown below:

```
[bdm0509:/usr/local/java] cd xqj
[bdm0509:/usr/local/java/xqj] ls
3rdPartySoftware.txt    examples                lib
Fixes.txt               help                    planExplain
Readme.txt              javadoc                 src
```

Add the XQJ JAR to your classpath

Now you need to update your classpath to include the XQuery JAR with the XQJ classes, stored in the lib directory and named ddxq.jar. This is not the original JAR you downloaded, but the one that was installed by the JAR you expanded above (yes, DataDirect might have made things less confusing by using a ZIP or .tar.gz file for their main download). You can set your classpath manually, or you can use a shell script, .profile file, or an IDE to add the JAR to your classpath. Just make sure that ddxq.jar is available on your classpath.

Set up database connectivity

DataDirect's download offers you database connectivity as well, allowing you to run XQueries against relational databases. When you filled out the download form, you were forced to choose a database to (potentially) connect with. DataDirect then customized the download so that database setup is available. That goes quite a bit beyond the scope of this article, but if you're interested in using your DataDirect libraries to connect to databases alongside making XQueries against XML documents saved on disk. A lot of other JAR files are in the lib directory of your installation, but you don't need any of those for simple file querying. If you later use DataDirect for database connectivity, then you'll want to explore these other JARs, too.

Running an XQuery from Java

Once you have XPath and XQuery firmly in hand, and an XQJ implementation on your classpath,

you're ready to write Java code and run your queries. There are two basic steps you'll follow in every program you write:

1. Create/Access an XQuery data source.
2. Execute your XQueries.

These are both fairly simple steps, and as long as you're not switching between different XQuery implementations, the first step stays the same in all your programs. In fact, you might want to bundle up the code you write to handle data source configuration and connection into a utility class (that's left for you as an exercise).

Working with XQuery data sources

If you've done much JDBC work, or written any n-tiered data-driven applications, you're probably familiar with the idea of a data source. In this context, a **data source** is a connection object that abstracts away the details of how that connection is made, and what the connection is connecting to. So a data source might represent a networked connection to a MySQL database, or a file-based connection to a static XML document. Once you have a data source, though, you can operate on it without regard to the connection semantics.

When you're just querying XML documents on a local disk (which is what this article focuses on), setting up a connection is pretty straightforward. Listing 3 shows a very basic Java program that creates a new data source, and gets ready for querying.

Data sources and databases

Data sources are the key to a vendor like DataDirect making database connectivity available alongside static XML document querying. Once the connection object is setup, queries can be run against it, and the vendor deals with executing that query against a static XML document, or a variety of different types of relational databases. You'll have to check out DataDirect's documentation, as well as that of your database vendor, to see if XQuerying your database is an option for you.

Listing 3. Setting up a data source for querying

```
package ibm.dw.xqj;

import com.ddtek.xquery3.XQConnection;
import com.ddtek.xquery3.XQException;
import com.ddtek.xquery3.xqj.DDXQDataSource;

public class XQueryTester {

  // Filename for XML document to query
  private String filename;

  // Data Source for querying
  private DDXQDataSource dataSource;

  // Connection for querying
  private XQConnection conn;

  public XQueryTester(String filename) {
    this.filename = filename;
  }

  public void init() throws XQException {
    dataSource = new DDXQDataSource();
    conn = dataSource.getConnection();
  }

  public static void main(String[] args) {
```

```
    if (args.length != 1) {
      System.err.println("Usage: java ibm.dw.xqj.XQueryTester [XML filename]");
      System.exit(-1);
    }

    try {
      String xmlFilename = args[0];
      XQueryTester tester = new XQueryTester(xmlFilename);
      tester.init();
    } catch (Exception e) {
      e.printStackTrace(System.err);
      System.err.println(e.getMessage());
    }
  }
}
```

This looks a lot longer and more complex than it really is. Much of that is because the test program is set up in a way that is pretty modular—with a constructor and init() method—so you can easily reuse this code for your own purposes without lots of changes.

This program gets a filename (from the command-line, which it passes into the class's constructor), and then executes the following code:

```
dataSource = new DDXQDataSource();
conn = dataSource.getConnection();
```

First, a new data source is created; the type of this object is com.ddtek.xquery3.xqj.DDXQDataSource. Use an empty constructor, and since you're not connecting to a database, you don't need any further configuration. Then, that data source is used to get a new com.ddtek.xquery3.XQConnection object. This object is the key to creating and executing new XQuery expressions; you haven't done that yet, but now your program is ready to take in a query string, and execute it.

Query a real XML document

You'll also need an XML file to actually query. In Download, you'll find a link to download a zipped, sample CD catalog library file; this is an XML document provided by the W3C for samples just like these. It's over 200 lines, so not printed here, but will give you a good document to query.

Listing 4 shows a small portion of the document to give you an idea of its structure.

Listing 4. A portion of cd_catalog.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<CATALOG>
       <CD>
                <TITLE>Empire Burlesque</TITLE>
                <ARTIST>Bob Dylan</ARTIST>
                <COUNTRY>USA</COUNTRY>
                <COMPANY>Columbia</COMPANY>
                <PRICE>10.90</PRICE>
                <YEAR>1985</YEAR>
        </CD>
        <CD>
                <TITLE>Hide your heart</TITLE>
                <ARTIST>Bonnie Tyler</ARTIST>
                <COUNTRY>UK</COUNTRY>
                <COMPANY>CBS Records</COMPANY>
                <PRICE>9.90</PRICE>
                <YEAR>1988</YEAR>
        </CD>
 <!-- more CD listings ... -->
```

```
</CATALOG>
```

Build the XQuery

Next, you need to create your actual query. This is easiest to do with a simple Java String. Create a new variable, and simply fill it with a string version of your query, as shown here (this code is all in the main() method) in the XQueryTester class:

```
try {
   String xmlFilename = args[0];
   XQueryTester tester = new XQueryTester(xmlFilename);
   tester.init();

   final String sep = System.getProperty("line.separator");
   String queryString =
     "      for $cd in doc($docName)/CATALOG/CD " +
     "    where $cd/YEAR > 1980 " +
     "      and $cd/COUNTRY = 'USA' " +
     " order by $cd/YEAR " +
     "   return " +
     "<cd><title>{$cd/TITLE/text()}</title>" +
     " <year>{$cd/YEAR/text()}</year></cd>";
   System.out.println(tester.query(queryString));
 } catch (Exception e) {
   e.printStackTrace(System.err);
   System.err.println(e.getMessage());
 }
```

The query itself selects all the CDs from 1981 on, recorded on a US label, orders them by year of release, and then returns XML that includes the title and year of release for each returned CD. There are a few things worth noting here:

- The docName variable represents the document to search. The code will need to handle populating that variable with the document supplied to the program through the command line.
- Instead of just returning a single value for each node in the result set, an XML string is returned. This XML could be dropped into another, larger XML document, displayed online, or passed into an XSL processor.
- The XML element names in the source document— CD, TITLE, YEAR, and so on.—are discarded in the result set, and new XML element names are used: cd, title, and year.

None of these on their own are difficult, but they are just additional ways that XQuery provides flexibility in selecting and returning data from XML.

One thing that is worth noting: when you return a string of XML as your result set, you should surround the variables in your return string within curly braces:

```
return <cd><title>{$cd/TITLE/text()}</title>" +
        " <year>{$cd/YEAR/text()}</year></cd>"
```

These curly braces let the XQuery processor know to treat the enclosed data as variables that are evaluated and replaced with a value, rather than literal text.

Declare an external variable

The XQuery uses a variable, $docName, to represent the document. But you need to declare that variable explicitly, and let the query know that something external—in this case, the Java program —will define that variable for use. In XQuery, you use the declare syntax for this purpose. Here's

the format of the declare syntax:

```
declare variable [variable-name] as [variable-type] external;
```

In this case, [variable-name] is $docName. [variable-type] should come from the XML Schema base data types. Most of the time, you'll use either xs:string for strings and xs:int for integers. There are several other types, but these are the most common.

So for the XQueryTester class, the query needs to be modified:

```
    try {
      String xmlFilename = args[0];
      XQueryTester tester = new XQueryTester(xmlFilename);
      tester.init();

      final String sep = System.getProperty("line.separator");
      String queryString =
        "declare variable $docName as xs:string external;" + sep +
        "      for $cd in doc($docName)/CATALOG/CD " +
        "    where $cd/YEAR > 1980 " +
        "      and $cd/COUNTRY = 'USA' " +
        " order by $cd/YEAR " +
        "   return " +
        "<cd><title>{$cd/TITLE/text()}</title>" +
        " <year>{$cd/YEAR/text()}</year></cd>";
      System.out.println(tester.query(queryString));
    } catch (Exception e) {
      e.printStackTrace(System.err);
      System.err.println(e.getMessage());
    }
```

Now, all that's left is to build a function for executing this query.

Run the XQuery

To run a query, you need to follow these steps:

1. Create an XQExpression object from your XQConnection.
2. Bind any variables to the query using the bindXXX() methods on your XQExpression object.
3. Execute the query, and store the results in an XQSequence object.

The only step that's tricky here is to bind variables. In this example, the variable docName needs to be associated with the filename passed in to the program. Since you're binding a string variable, you use bindString. That method takes three arguments:

1. a javax.xml.namespace.QName instance (this class is from the JAXP package), with the name of the variable in your XQuery
2. The value to bind to that variable
3. The atomic type that the variable type should match up to

If you put all of this together, you'll get a method like this:

```
public String query(String queryString) throws XQException {
  XQExpression expression = conn.createExpression();
  expression.bindString(new QName("docName"), filename,
    conn.createAtomicType(XQItemType.XQBASETYPE_STRING));
  XQSequence results = expression.executeQuery(queryString);
  return results.getSequenceAsString(new Properties());
}
```

The first line, XQExpression expression = conn.createExpression();, creates a new expression

object. Then, the filename of the document to query is bound to the docName variable: expression.bindString(new QName("docName"), filename, conn.createAtomicType(XQItemType.XQBASETYPE_STRING));. It's not worth worrying much about the mechanics of the QName object at this point; just give it the name of the variable in your XQuery (without the $ character). The next argument is the filename; the third is another fairly constant value, the type you want to ensure your value conforms to. Since in the query the variable was defined as xs:string, you want the type XQItemType.XQBASETYPE_STRING.

While some of this isn't intuitive, most of these are things you simply look up in the API docs. The other bindXXX() methods, like bindInt(), bindFloat(), and so forth, all work the same way.

Last, you execute the query, and get back a sequence of results. You can iterate through this result set, or even easier, convert it to a string and let your program work with the results as a long string of data. That's what's done in the query() method, as it doesn't require the calling program to know about the DataDirect XQuery API.

Listing 5 shows the completed XQueryTester code.

Listing 5. The completed XQueryTester class

```
package ibm.dw.xqj;

import javax.xml.namespace.QName;
import java.util.Properties;

import com.ddtek.xquery3.XQConnection;
import com.ddtek.xquery3.XQException;
import com.ddtek.xquery3.XQExpression;
import com.ddtek.xquery3.XQItemType;
import com.ddtek.xquery3.XQSequence;
import com.ddtek.xquery3.xqj.DDXQDataSource;

public class XQueryTester {

  // Filename for XML document to query
  private String filename;

  // Data Source for querying
  private DDXQDataSource dataSource;

  // Connection for querying
  private XQConnection conn;

  public XQueryTester(String filename) {
    this.filename = filename;
  }

  public void init() throws XQException {
    dataSource = new DDXQDataSource();
    conn = dataSource.getConnection();
  }

  public String query(String queryString) throws XQException {
    XQExpression expression = conn.createExpression();
    expression.bindString(new QName("docName"), filename,
      conn.createAtomicType(XQItemType.XQBASETYPE_STRING));
    XQSequence results = expression.executeQuery(queryString);
    return results.getSequenceAsString(new Properties());
  }

  public static void main(String[] args) {
    if (args.length != 1) {
```

```
      System.err.println("Usage: java ibm.dw.xqj.XQueryTester [XML filename]");
      System.exit(-1);
    }

    try {
      String xmlFilename = args[0];
      XQueryTester tester = new XQueryTester(xmlFilename);
      tester.init();

      final String sep = System.getProperty("line.separator");
      String queryString =
        "declare variable $docName as xs:string external;" + sep +
        "      for $cd in doc($docName)/CATALOG/CD " +
        "    where $cd/YEAR > 1980 " +
        "      and $cd/COUNTRY = 'USA' " +
        " order by $cd/YEAR " +
        "   return " +
        "<cd><title>{$cd/TITLE/text()}</title>" +
        " <year>{$cd/YEAR/text()}</year></cd>";
      System.out.println(tester.query(queryString));
    } catch (Exception e) {
      e.printStackTrace(System.err);
      System.err.println(e.getMessage());
    }
  }
}
```

Run your XQueryTester

Compile this program, and run it like this:

```
[bdm0509:~/Documents/developerworks/java_xquery]
   java ibm.dw.xqj.XQueryTester cd_catalog.xml
<cd><title>Greatest Hits</title><year>1982</year></cd><cd><title>
Empire Burlesque</title><year>1985</year></cd><cd><title>When a man loves a
woman
</title><year>1987</year></cd><cd><title>The dock of the bay</title><year>
1987</year></cd><cd><title>Unchain my heart</title><year>1987</year></cd>
<cd><title>Big Willie style</title><year>1997</year></cd><cd><title>
1999 Grammy Nominees</title><year>1999</year><cd>
```

Note: *I've inserted the line breaks into this to make it fit the online article format. The actual results have no line breaks, and are "crammed" together all on one line.*

Experiment!

The program is set up to handle queries and bind to your XML document. Now you can play with the query string, and really get a feel for both XQuery and running those queries from Java programs. Try to select all the CDs; then change the format of the returned results to formatted text. See if you can return all CDs, from the most current to the oldest; or all the CDs priced under ten dollars. Once you've a program like this, you can easily tweak your query, experiment, and even change the XML document that you feed to the program.

Conclusion

It's difficult to talk about XQJ without getting into the basics of XQuery; and you can't talk about XQuery without digging deeply into XPath. Connecting all these disparate parts often means that

doing just one part—the Java program to execute a query, or the location within a document that you start a query from—is pretty easy. And that's the best way to think about using XQuery from Java: as a collection of individually simple parts that can be combined into a cool, powerful program.

Because you'll have to get comfortable with several different technologies, you should keep two components the same, and play with the third. So keep your Java program the same, and play with different queries and different input documents. Then, with a basic query, experiment with using more input variables declared in your query. You might want to toy with the XPaths in your return statement, and then the starting point for your searches. As you get better at each component, you'll find your queries are more complex, and your Java coding becomes more robust.