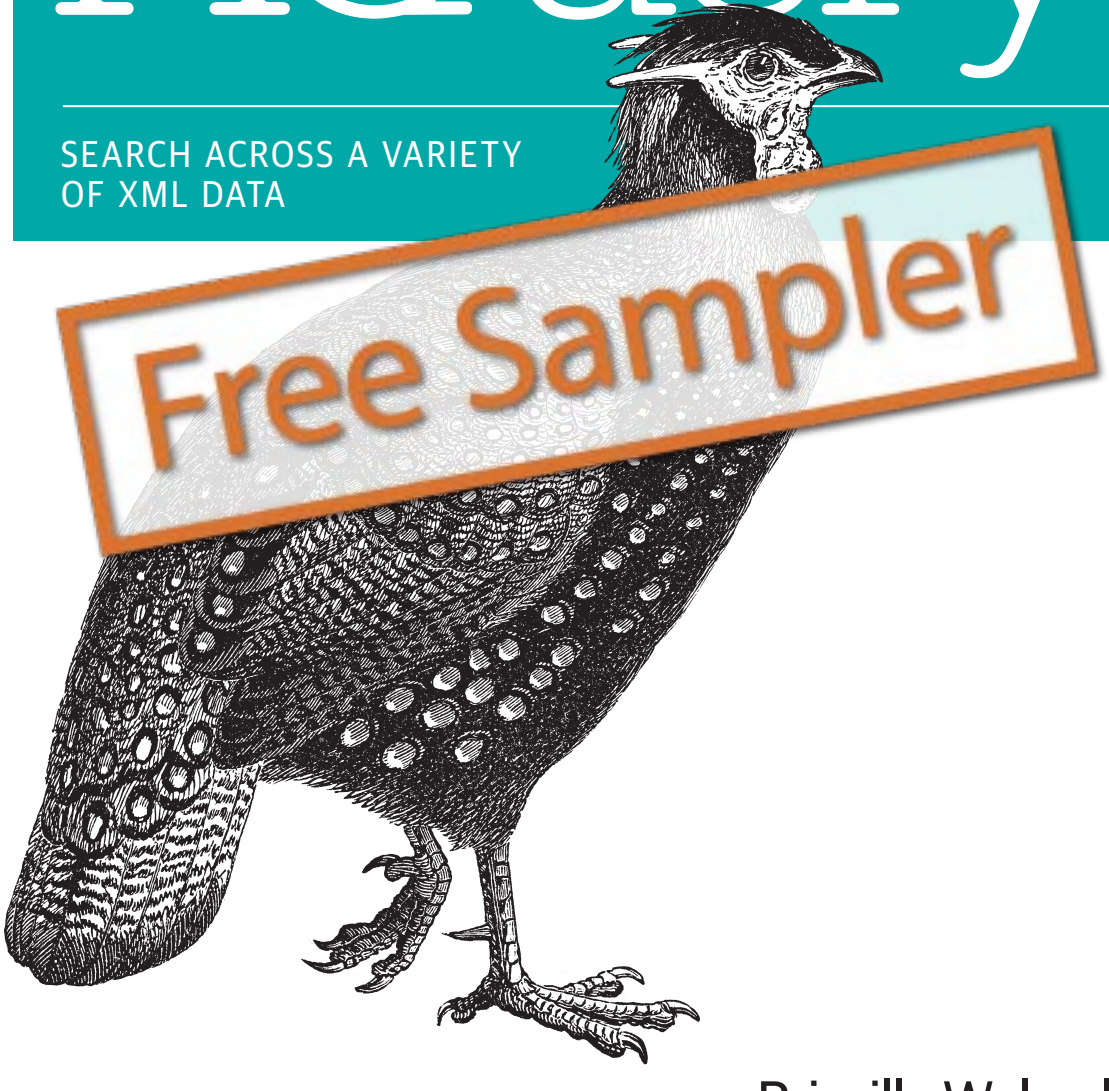
O'REILLY®

2nd Edition

XQuery

SEARCH ACROSS A VARIETY OF XML DATA



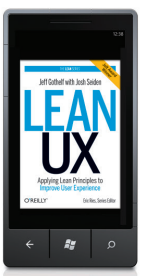
Priscilla Walmsley

O'Reilly ebooks.

Your bookshelf on your devices.

PDF

Mobi



ePub

DAISY

When you buy an ebook through oreilly.com you get lifetime access to the book, and whenever possible we provide it to you in four DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at ebooks.oreilly.com

You can also purchase O'Reilly ebooks through the iBookstore, the Android Marketplace, and Amazon.com.

O'REILLY®

SECOND EDITION

XQuery

Search Across a Variety of XML Data

Priscilla Walmsley

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

XQuery
by Priscilla Walmsley

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meg Foley
Production Editor: Shiny Kalapurakkel
Copyeditor: Nan Reinhardt
Proofreader: Sonia Saruba
Indexer: Priscilla Walmsley
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2007: First Edition
December 2015: Second Edition

Revision History for the Second Edition
2015-11-30: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491915103* for release details.

978-1-491-91510-3

[LSI]

Table of Contents

CHAPTER 1

Introduction to XQuery

This chapter provides background on the purpose and capabilities of XQuery. It also gives a quick introduction to the features of XQuery that are covered in more detail later in the book. It is designed to provide a basic familiarity with the most commonly used kinds of expressions, without getting too bogged down in the details.

What Is XQuery?

The use of XML has exploded in recent years. An enormous amount of information is now stored in XML, both in XML databases and in documents on a filesystem. This includes highly structured data such as sales figures, semi-structured data such as product catalogs and yellow pages, and relatively unstructured data such as letters and books. Even more information is passed between systems as transitory XML documents.

All of this data is used for a variety of purposes. For example, sales figures may be useful for compiling financial statements that may be published on the Web, reporting results to the tax authorities, calculating bonuses for salespeople, or creating internal reports for planning. For each of these uses, we are interested in different elements of the data and expect it to be formatted and transformed according to our needs.

XQuery is a query language designed by the W3C to address these needs. It allows you to select the XML data elements of interest, reorganize and possibly transform them, and return the results in a structure of your choosing.

Capabilities of XQuery

XQuery has a rich set of features that allow many different types of operations on XML data and documents, including:

- Selecting information based on specific criteria
- Filtering out unwanted information
- Searching for information within a document or set of documents
- Joining data from multiple documents or collections of documents
- Sorting, grouping, and aggregating data
- Transforming and restructuring XML data into another XML vocabulary or structure
- Performing arithmetic calculations on numbers and dates
- Manipulating strings to reformat text

As you can see, XQuery can be used not just to extract sections of XML documents, but also to manipulate and transform the results for output. In fact, XQuery is a Turing-complete functional programming language, which means you can also use it for general-purpose programming and application development, not just for querying data.

Uses for XQuery

There are as many reasons to query XML as there are reasons to use XML. Some examples of common uses for the XQuery language are:

- Finding textual documents in a native XML database and presenting styled results
- Generating reports on data stored in a database for presentation on the Web as HTML
- Extracting information from a relational database for use in a web service
- Pulling data from databases or packaged software and transforming it for application integration
- Combining content from traditionally non-XML sources to implement content management and delivery
- Ad hoc querying of standalone XML documents for the purposes of testing or research
- Building entire complex web applications

Processing Scenarios

XQuery's sweet spot is querying bodies of XML content that encompass many XML documents, often stored in databases. For this reason, it is sometimes called the "SQL of XML." Some of the earliest XQuery implementations were in native XML database products. The term "native XML database" generally refers to a database that is designed for XML content from the ground up, as opposed to a traditionally relational database. Rather than being oriented around tables and columns, its data model is based on hierarchical documents and collections of documents.

Native XML databases are most often used for narrative content and other data that is less predictable than what you would typically store in a relational database. Many of these products are now known by the broader term *NoSQL database* and provide support for not just XML but also JSON and other data formats. Examples of these database products that support XQuery are eXist, MarkLogic Server, BaseX, Zorba, and EMC Documentum xDB. Of these, all but MarkLogic Server and EMC Documentum xDB are open source. These products provide the traditional capabilities of databases, such as data storage, indexing, querying, loading, extracting, backup, and recovery. Most of them also provide some added value in addition to their database capabilities. For example, they might provide advanced full-text searching functionality, document conversion services, or end-user interfaces.

Major relational database products, including Oracle (via its XML DB), IBM DB2 (via pureXML), and Microsoft SQL Server, also have support for XML and various versions of XQuery. Early implementations of XML in relational databases involved storing XML in table columns as blobs or character strings and providing query access to those columns. However, these vendors are increasingly blurring the line between native XML databases and relational databases with new features that allow you to store XML natively.

Other XQuery processors are not embedded in a database product, but work independently. They might be used on physical XML documents stored as files on a file system or on the Web. They might also operate on XML data that is passed in memory from some other process. The most notable product in this category is Saxon, which has both open source and commercial versions. Altova's RaptorXML also provides support for standalone XQuery queries.

XML editors provide support for editing and running XQuery queries and displaying the results. Some, like Altova's XMLSpy, have their own embedded XQuery implementations. Others, like oXygen XML Editor, allow you to run queries using one or more separate XQuery processors. If you are new to XQuery, a free trial license to a product like oXygen or XMLSpy is a good way to get started running queries.

Easing into XQuery

The rest of this chapter takes you through a set of example queries, each of which builds on the previous one. Three XML documents are used repeatedly as input documents to the query examples throughout the book. They will be used so frequently that it may be worth printing them from the companion web site at *http://www.datypic.com/books/xquery/chapter01.html* so that you can view them alongside the examples.

These three examples are quite simplistic, but they are useful for educational purposes because they are easy to learn and remember while looking at query examples. In reality, most XQuery queries will be executed against much more complex documents, and often against multiple documents as a collection. However, in order to keep the examples reasonably concise and clear, this book will work with smaller documents that have a representative mix of XML characteristics.

The *catalog.xml* document is a product catalog containing general information about products (Example 1-1).

Example 1-1. Product catalog input document (catalog.xml)

```
<catalog>
  <product dept="WMN">
    <number>557</number>
    <name language="en">Fleece Pullover</name>
    <colorChoices>navy black</colorChoices>
  </product>
  <product dept="ACC">
    <number>563</number>
    <name language="en">Floppy Sun Hat</name>
  </product>
  <product dept="ACC">
    <number>443</number>
    <name language="en">Deluxe Travel Bag</name>
  </product>
  <product dept="MEN">
    <number>784</number>
    <name language="en">Cotton Dress Shirt</name>
    <colorChoices>white gray</colorChoices>
    <desc>Our <i>favorite</i> shirt!</desc>
  </product>
</catalog>
```

The *prices.xml* document contains prices for most of the products, based on an effective date (Example 1-2).

Example 1-2. Price information input document (prices.xml)

```
<prices>
  <priceList effDate="2015-11-15">
    <prod num="557">
      <price currency="USD">29.99</price>
      <discount type="CLR">10.00</discount>
    </prod>
    <prod num="563">
      <price currency="USD">69.99</price>
    </prod>
    <prod num="443">
      <price currency="USD">39.99</price>
      <discount type="CLR">3.99</discount>
    </prod>
  </priceList>
</prices>
```

The *order.xml* document is a simple order containing a list of products ordered (referenced by a number that matches the number used in *catalog.xml*), along with quantities and colors (Example 1-3).

Example 1-3. Order input document (order.xml)

```
<order num="00299432" date="2015-09-15" cust="0221A">
  <item dept="WMN" num="557" quantity="1" color="navy"/>
  <item dept="ACC" num="563" quantity="1"/>
  <item dept="ACC" num="443" quantity="2"/>
  <item dept="MEN" num="784" quantity="1" color="white"/>
  <item dept="MEN" num="784" quantity="1" color="gray"/>
  <item dept="WMN" num="557" quantity="1" color="black"/>
</order>
```

Path Expressions

The most straightforward kind of query simply selects elements or attributes from an input document. This type of query is known as a path expression. For example, the path expression:

```
doc("catalog.xml")/catalog/product
```

will select all the `product` elements from the *catalog.xml* document.

Path expressions are used to traverse an XML tree to select elements and attributes of interest. They are similar to paths used for filenames in many operating systems. They consist of a series of steps, separated by slashes, that traverse the elements and attributes in the XML documents. In this example, there are three steps:

1. `doc("catalog.xml")` calls an XQuery function named `doc`, passing it the name of the file to open
2. `catalog` selects the `catalog` element, the outermost element of the document
3. `product` selects all the `product` children of `catalog`

The result of the query will be the four `product` elements, exactly as they appear (with the same attributes and contents) in the input document. Example 1-4 shows the complete result.

Example 1-4. Four `product` elements selected from the catalog

```
<product dept="WMN">
  <number>557</number>
  <name language="en">Fleece Pullover</name>
  <colorChoices>navy black</colorChoices>
</product>
<product dept="ACC">
  <number>563</number>
  <name language="en">Floppy Sun Hat</name>
</product>
<product dept="ACC">
  <number>443</number>
  <name language="en">Deluxe Travel Bag</name>
</product>
<product dept="MEN">
  <number>784</number>
  <name language="en">Cotton Dress Shirt</name>
  <colorChoices>white gray</colorChoices>
  <desc>Our <i>favorite</i> shirt!</desc>
</product>
```

The asterisk (*) can be used as a wildcard to indicate any element name. For example, the path expression:

```
doc("catalog.xml")/*/product
```

will return any `product` children of the outermost element, regardless of the outermost element's name. Alternatively, you can use a double slash (//) to return `product` elements that appear anywhere in the catalog document, as in:

```
doc("catalog.xml")//product
```

In addition to traversing the XML document, a path expression can contain predicates that filter out elements or attributes that do not meet a particular criterion. Predicates are indicated by square brackets. For example, the path expression:

```
doc("catalog.xml")/catalog/product[@dept = "ACC"]
```

contains a predicate. It selects only those `product` elements whose `dept` attribute value is `ACC`. The `@` sign is used to indicate that `dept` is an attribute as opposed to a child element.

When a predicate contains a number, it serves as an index. For example:

```
doc("catalog.xml")/catalog/product[2]
```

will return the second `product` element in the catalog.

Path expressions are convenient because of their compact, easy-to-remember syntax. However, they have a limitation: they can only return elements and attributes as they appear in input documents. Any elements selected in a path expression appear in the results with the same names, the same attributes and contents, and in the same order as in the input document. When you select the `product` elements, you get them with all of their children and with their `dept` attributes. Path expressions are covered in detail in Chapter 4.

FLWORs

The basic structure of many (but not all) queries is the FLWOR expression. FLWOR (pronounced "flower") stands for "for, let, where, order by, return," the most common keywords used in the expression.

FLWORs, unlike path expressions, allow you to manipulate, transform, and sort your results. Example 1-5 shows a simple FLWOR that returns the names of all products in the ACC department.

Example 1-5. Simple FLWOR

Query

```
for $prod in doc("catalog.xml")/catalog/product
where $prod/@dept = "ACC"
order by $prod/name
return $prod/name
```

Results

```
<name language="en">Deluxe Travel Bag</name>
<name language="en">Floppy Sun Hat</name>
```

As you can see, the FLWOR is made up of several parts:

`for`
: This clause sets up an iteration through the `product` elements, and the rest of the FLWOR is evaluated once for each of the four products. Each time, a variable

named `$prod` is bound to a different `product` element. Dollar signs are used to indicate variable names in XQuery.

`where`
: This clause selects only products in the ACC department. This has the same effect as a predicate (`[@dept = "ACC"]`) in a path expression.

`order by`
: This clause sorts the results by product name, something that is not possible with path expressions.

`return`
: This clause indicates that the `product` element's `name` children should be returned in the query result.

The `let` clause (the L in FLWOR) is used to bind the value of a variable. Unlike a `for` clause, it does not set up an iteration. Example 1-6 shows a FLWOR that returns the same result as Example 1-5. The second line is a `let` clause that binds the `product` element's `name` child to a variable called `$name`. The `$name` variable is then referenced later in the FLWOR, in both the `order by` clause and the `return` clause.

Example 1-6. Adding a `let` clause

```
for $prod in doc("catalog.xml")/catalog/product
let $name := $prod/name
where $prod/@dept = "ACC"
order by $name
return $name
```

The `let` clause serves as a programmatic convenience that avoids repeating the same expression multiple times. With some implementations, it may improve performance because the expression is evaluated only once instead of each time it is needed.

This chapter has provided only very basic examples of FLWORs. In fact, FLWORs can become quite complex. Multiple `for` clauses are permitted, which set up iterations within iterations. Additional clauses such as `group by`, `count`, and `window` are available. In addition, complex expressions can be used in any of the clauses. FLWORs are discussed in detail in Chapter 6. Even more advanced examples of FLWORs are provided in Chapter 9.

Adding XML Elements and Attributes

Sometimes you want to reorganize or transform the elements in the input documents into differently named or structured elements. XML constructors can be used to create elements and attributes that appear in the query results.

Adding Elements

Suppose you want to wrap the results of your query in a different XML vocabulary, for example, XHTML. You can do this using a familiar XML-like syntax. To wrap the `name` elements in a `ul` element, for instance, you can use the query shown in Example 1-7. The `ul` element represents an unordered list in HTML.

Example 1-7. Wrapping results in a new element

Query

```
<ul>{
  for $prod in doc("catalog.xml")/catalog/product
  where $prod/@dept='ACC'
  order by $prod/name
  return $prod/name
}</ul>
```

Results

```
<ul>
  <name language="en">Deluxe Travel Bag</name>
  <name language="en">Floppy Sun Hat</name>
</ul>
```

This example is the same as Example 1-5, with the addition of the first and last lines. In the query, the `ul` start tag and end tag, and everything in between, is known as an element constructor. The curly braces around the content of the `ul` element signify that it is an expression (known as an enclosed expression) that is to be evaluated. In this case, the enclosed expression returns two elements, which become children of `ul`.

Any content in an element constructor that is not inside curly braces appears in the results as is. For example:

```
<h1>There are {count(doc("catalog.xml")//product)} products.</h1>
```

will return the result:

```
<h1>There are 4 products.</h1>
```

The content outside the curly braces, namely the strings `"There are "` and `" products."`, appear literally in the results, as textual content of the `h1` element.

The element constructor does not need to be the outermost expression in the query. You can include element constructors at various places in your query. For example, if you want to wrap each resulting `name` element in its own `li` element, you could use the query shown in Example 1-8. An `li` element represents a list item in HTML.

Example 1-8. Element constructor in FLWOR `return` clause

Query

```
<ul>{
  for $prod in doc("catalog.xml")/catalog/product
  where $prod/@dept='ACC'
  order by $prod/name
  return <li>{$prod/name}</li>
}</ul>
```

Results

```
<ul>
  <li><name language="en">Deluxe Travel Bag</name></li>
  <li><name language="en">Floppy Sun Hat</name></li>
</ul>
```

Here, the `li` element constructor appears in the `return` clause of a FLWOR. Since the `return` clause is evaluated once for each iteration of the `for` clause, two `li` elements appear in the results, each with a `name` element as its child.

However, suppose you don't want to include the `name` elements at all, just their contents. You can do this by calling a built-in function called `data`, which extracts the contents of an element. This is shown in Example 1-9.

Example 1-9. Using the `data` function

Query

```
<ul>{
  for $prod in doc("catalog.xml")/catalog/product
  where $prod/@dept='ACC'
  order by $prod/name
  return <li>{data($prod/name)}</li>
}</ul>
```

Results

```
<ul>
  <li>Deluxe Travel Bag</li>
  <li>Floppy Sun Hat</li>
</ul>
```

Now no `name` elements appear in the results. In fact, no elements at all from the input document appear.

Adding Attributes

You can also add your own attributes to results using an XML-like syntax. Example 1-10 adds attributes to the `ul` and `li` elements.

Example 1-10. Adding attributes to results

Query

```
<ul type="square">{
  for $prod in doc("catalog.xml")/catalog/product
  where $prod/@dept='ACC'
  order by $prod/name
  return <li class="{$prod/@dept}">{data($prod/name)}</li>
}</ul>
```

Results

```
<ul type="square">
  <li class="ACC">Deluxe Travel Bag</li>
  <li class="ACC">Floppy Sun Hat</li>
</ul>
```

As you can see, attribute values, like element content, can either be literal text or enclosed expressions. The `ul` element constructor has an attribute `type` that is included as is in the results, while the `li` element constructor has an attribute `class` whose value is an enclosed expression delimited by curly braces. In attribute values, unlike element content, you don't need to use the `data` function to extract the value: it happens automatically.

The constructors shown in these examples are known as direct constructors, because they use an XML-like syntax. You can also construct elements and attributes with dynamically determined names, using computed constructors. Chapter 5 provides detailed coverage of XML constructors.

Functions

Almost 200 functions are built into XQuery, covering a broad range of functionality. Functions can be used to manipulate strings and dates, perform mathematical calculations, combine sequences of elements, and perform many other useful jobs. You can also define your own functions, either in the query itself, or in an external library.

Both built-in and user-defined functions can be called from almost any place in a query. For instance, Example 1-9 calls the `doc` function in a `for` clause, and the `data` function in an enclosed expression. Chapter 8 explains how to call functions and also

describes how to write your own user-defined functions. Appendix A lists all the built-in functions and explains each of them in detail.

Joins

One of the major benefits of FLWORs is that they can easily join data from multiple sources. For example, suppose you want to join information from your product catalog (*catalog.xml*) and your order (*order.xml*). You want a list of all the items in the order, along with their number, name, and quantity.

The name comes from the product catalog, and the quantity comes from the order. The product number appears in both input documents, so it is used to join the two sources. Example 1-11 shows a FLWOR that performs this join.

Example 1-11. Joining multiple input documents

Query

```
for $item in doc("order.xml")//item
let $name := doc("catalog.xml")//product[number = $item/@num]/name
return <item num="{$item/@num}"
             name="{$name}"
             quan="{$item/@quantity}"/>
```

Results

```
<item num="557" name="Fleece Pullover" quan="1"/>
<item num="563" name="Floppy Sun Hat" quan="1"/>
<item num="443" name="Deluxe Travel Bag" quan="2"/>
<item num="784" name="Cotton Dress Shirt" quan="1"/>
<item num="784" name="Cotton Dress Shirt" quan="1"/>
<item num="557" name="Fleece Pullover" quan="1"/>
```

The `for` clause sets up an iteration through each `item` from the order. For each item, the `let` clause goes to the product catalog and gets the name of the product. It does this by finding the `product` element whose `number` child equals the item's `num` attribute, and selecting its `name` child. Because the FLWOR iterated six times, the results contain one new `item` element for each of the six `item` elements in the order document. Joins are covered in Chapter 6.

Aggregating and Grouping Values

One common use for XQuery is to summarize and group XML data. It is sometimes useful to find the sum, average, or maximum of a sequence of values, grouped by a particular value. For example, suppose you want to know the number of items contained in an order, grouped by department. The query shown in Example 1-12 accomplishes this. It uses a `group by` clause to group the items by department, and

the `sum` function to calculate the totals of the `quantity` attribute values for the items in each department.

Example 1-12. Aggregating values

Query

```
xquery version "3.0";
for $i in doc("order.xml")//item
let $d := $i/@dept
group by $d
order by $d
return <department name="{$d}" totQuantity="{sum($i/@quantity)}"/>
```

Results

```
<department name="ACC" totQuantity="3"/>
<department name="MEN" totQuantity="2"/>
<department name="WMN" totQuantity="2"/>
```

Chapter 7 covers sorting, grouping, and aggregating values in detail. The version declaration on the first line of this example is used to show that use of the `group by` clause requires at least version 3.0 of XQuery.