

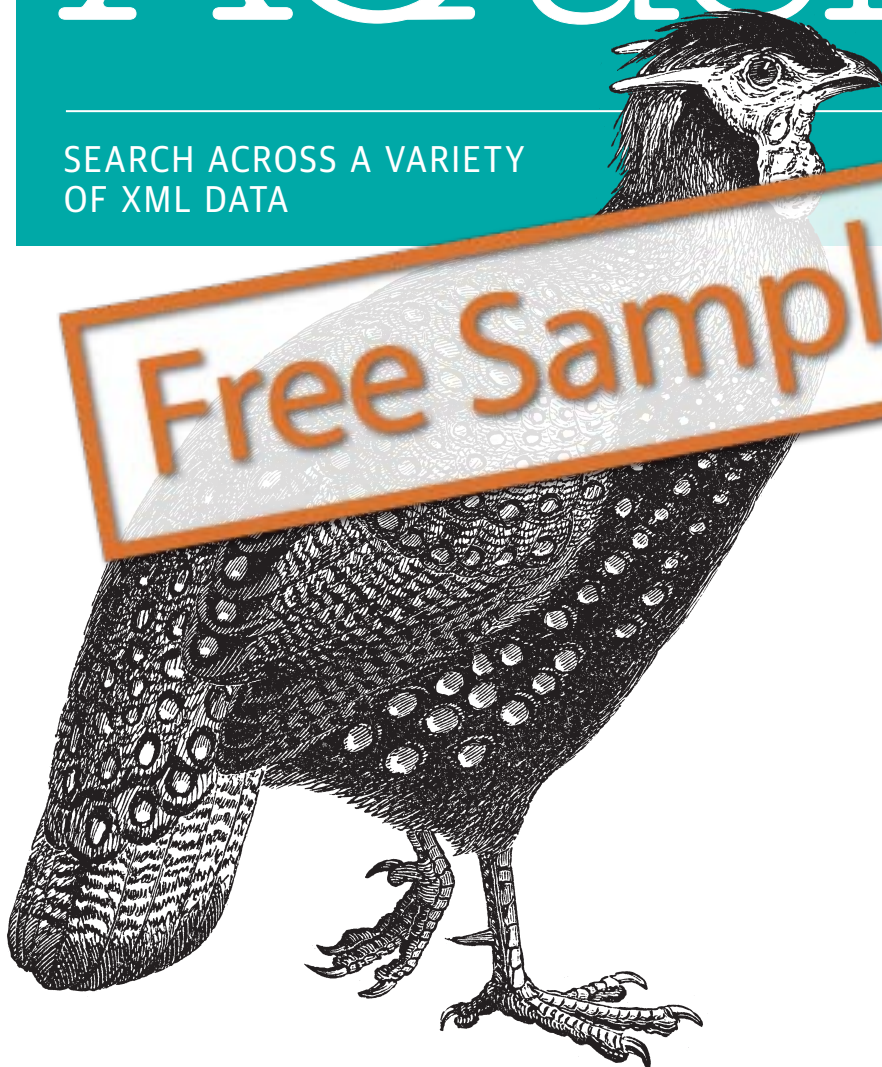
O'REILLY®

2nd Edition

XQuery

SEARCH ACROSS A VARIETY
OF XML DATA

Free Sampler



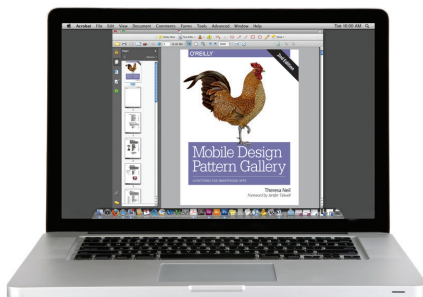
Priscilla Walmsley

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](http://AndroidMarketplace), 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

Copyright © 2016 Priscilla Walmsley. All rights reserved.

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.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *XQuery*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-91510-3

[LSI]

Table of Contents

Preface.....	xvii
1. Introduction to XQuery.....	1
What Is XQuery?	1
Capabilities of XQuery	2
Uses for XQuery	2
Processing Scenarios	3
Easing into XQuery	4
Path Expressions	5
FLWORS	7
Adding XML Elements and Attributes	8
Adding Elements	9
Adding Attributes	11
Functions	11
Joins	12
Aggregating and Grouping Values	12
2. XQuery Foundations.....	15
The Design and History of the XQuery Language	15
XQuery in Context	16
XQuery and XPath	16
XQuery Versus XSLT	16
XQuery Versus SQL	17
XQuery and XML Schema	17
Processing Queries	18
Input Documents	18
The Query	19
The Context	20

The Query Processor	20
The Results of the Query	21
The XQuery Data Model	21
Nodes	22
Atomic Values	26
Sequences	27
Types	28
Namespaces	28
3. Expressions: XQuery Building Blocks.....	31
Categories of Expressions	31
Keywords and Names	32
Whitespace in Queries	33
Literals	33
Variables	34
Function Calls	34
Comments	35
Precedence and Parentheses	35
Comparison Expressions	37
General Comparisons	37
Value Comparisons	38
Node Comparisons	40
Conditional (if-then-else) Expressions	41
Conditional Expressions and Effective Boolean Value	42
Nesting Conditional Expressions	43
Switch Expressions	43
Logical (and/or) Expressions	45
Precedence of Logical Expressions	45
Negating a Boolean Value	46
4. Navigating XML by Using Paths.....	47
Path Expressions	47
Path Expressions and Context	48
Steps	49
Axes	49
Node Tests	50
Abbreviated Syntax	53
Other Expressions as Steps	53
Predicates	54
Comparisons in Predicates	55
Using Positions in Predicates	56
Using Multiple Predicates	59

More Complex Predicates	59
A Closer Look at Context	60
Working with the Context Node	61
Accessing the Root	61
Dynamic Paths	62
The Simple Map Operator	63
5. Adding Elements and Attributes to Results.	65
Including Elements and Attributes from the Input Document	65
Direct Element Constructors	66
Containing Literal Characters	67
Containing Other Element Constructors	68
Containing Enclosed Expressions	68
Specifying Attributes Directly	71
Declaring Namespaces in Direct Constructors	72
Use Case: Modifying an Element from the Input Document	73
Direct Element Constructors and Whitespace	74
Computed Constructors	77
Computed Element Constructors	77
Computed Attribute Constructors	80
Use Case: Turning Content to Markup	80
6. Selecting and Joining Using FLWORS.	83
Selecting with Path Expressions	83
FLWOR Expressions	83
The for Clause	85
The let Clause	88
The where Clause	89
The return Clause	90
The Scope of Variables	91
Quantified Expressions	91
Binding Multiple Variables	93
Selecting Distinct Values	93
Joins	95
Three-Way Joins	96
Outer Joins	96
Joins and Types	98
7. Sorting and Grouping.	99
Sorting in XQuery	99
The order by Clause	99
The sort Function	103

Document Order	103
Document Order Comparisons	105
Reversing the Order	106
Indicating That Order Is Not Significant	106
Grouping	108
Grouping Using the group by Clause	109
Aggregating Values	112
Ignoring “Missing” Values	114
Counting “Missing” Values	115
Aggregating on Multiple Values	116
Constraining and Sorting on Aggregated Values	116
8. Functions.....	119
Built-in Versus User-Defined Functions	119
Calling Functions	119
Function Names	120
Function Signatures	121
Argument Lists	121
Sequence Types	123
Calling Functions with the Arrow Operator	124
User-Defined Functions	124
Why Define Your Own Functions?	124
Function Declarations	125
The Function Body	126
The Function Name	127
The Parameter List	127
Functions and Context	130
Recursive Functions	130
9. Advanced Queries.....	133
Working with Positions and Sequence Numbers	133
Adding Sequence Numbers to Results	133
Using the count Clause	135
Testing for the Last Item	137
Windowing	138
Using start and end Conditions	140
Windows Based on Position	141
Windows Based on Previous or Next Items	142
Sliding Windows	143
Copying Input Elements with Modifications	144
Adding Attributes to an Element	145
Removing Attributes from an Element	146

Removing Attributes from All Descendants	147
Removing Child Elements	147
Changing Names	148
Combining Results	150
Sequence Constructors	150
The union Expression	151
The intersect Expression	151
The except Expression	151
Using Intermediate XML Documents	152
Creating Lookup Tables	152
Reducing Complexity	153
10. Namespaces and XQuery.....	157
XML Namespaces	157
Namespace URIs	157
Declaring Namespaces	158
Default Namespace Declarations	159
Namespaces and Attributes	159
Namespace Declarations and Scope	160
Namespaces and XQuery	161
Namespaces in Queries	162
Predeclared Namespaces	162
Prolog Namespace Declarations	163
Namespace Declarations in Direct Element Constructors	166
Namespace Declarations in Computed Constructors	167
The Impact and Scope of Namespace Declarations	168
Controlling Namespace Declarations in Your Results	170
In-Scope Versus Statically Known Namespaces	171
Controlling the Copying of Namespace Declarations	174
URI-Qualified Names	177
11. A Closer Look at Types.....	179
The XQuery Type System	179
Advantages of a Strong Type System	179
Do You Need to Care About Types?	180
The Built-in Types	181
Atomic Types	181
List Types	183
Union Types	183
Types, Nodes, and Atomic Values	183
Nodes and Types	183
Atomic Values and Types	184

Type Checking in XQuery	184
The Static Analysis Phase	184
The Dynamic Evaluation Phase	185
Automatic Type Conversions	185
Subtype Substitution	185
Type Promotion	186
Casting of Untyped Values	186
Atomization	186
Effective Boolean Value	187
Function Conversion Rules	189
Sequence Types	190
Occurrence Indicators	191
Generic Sequence Types	192
Simple Type Names as Sequence Types	193
Element and Attribute Tests	193
Sequence Type Matching	194
The instance of Expression	194
Constructors and Casting	195
Constructors	195
The Cast Expression	196
The Castable Expression	197
Casting Rules	198
12. Prologs, Modules, and Variables.	201
Structure of a Query: Prolog and Body	201
Prolog Declarations	202
The Version Declaration	203
Assembling Queries from Multiple Modules	204
Library Modules	204
Importing a Library Module	205
Loading a Library Module Dynamically	207
Variable Declarations	208
Variable Declaration Syntax	208
The Scope of Variables	209
Variable Names	209
Initializing Expressions	210
External Variables	210
Private Functions and Variables	211
Declaring External Functions	211
13. Inputs and Outputs.	213
Types of Input and Output Documents	213

Accessing Input Documents	214
Accessing a Single Document with a Function	214
Accessing a Collection	215
Setting the Context Outside the Query	216
Using Variables	216
Setting the Context in the Prolog	217
Serializing Output	217
Serialization Methods	218
Serialization Parameters	220
Specifying Serialization Parameters by Using Option Declarations	224
Specifying Serialization Parameters by Using a Separate XML Document	225
Serialization Errors	226
Serializing to a String	226
14. Using Schemas with XQuery.....	227
What Is a Schema?	227
Why Use Schemas with Queries?	228
W3C XML Schema: A Brief Overview	230
Element and Attribute Declarations	230
Types	231
Namespaces and XML Schema	232
In-Scope Schema Definitions	233
Where Do In-Scope Schema Definitions Come From?	233
Schema Imports	234
Schema Validation and Type Assignment	236
The Validate Expression	236
Validation Mode	238
Assigning Type Annotations to Nodes	238
Nodes and Typed Values	239
Types and Newly Constructed Elements and Attributes	240
Sequence Types and Schemas	241
15. Static Typing.....	245
What Is Static Typing?	245
Obvious Static Type Errors	246
Static Typing and Schemas	246
Raising “False” Errors	247
Static Typing Expressions and Constructs	247
The Typeswitch Expression	248
The Treat Expression	250
Type Declarations	251
Type Declarations in FLWORS	251

Type Declarations in Quantified Expressions	252
Type Declarations in Global Variable Declarations	253
The zero-or-one, one-or-more, and exactly-one Functions	253
16. Writing Better Queries.....	255
Query Design Goals	255
Clarity	256
Improving the Layout	256
Choosing Names	257
Using Comments for Documentation	257
Modularity	259
Robustness	259
Handling Data Variations	259
Handling Missing Values	260
Error Handling	262
Avoiding Dynamic Errors	262
The error and trace Functions	263
Try/Catch Expressions	263
Performance	265
Avoid Reevaluating the Same or Similar Expressions	266
Avoid Unnecessary Sorting	266
Avoid Expensive Path Expressions	267
Use Predicates Instead of where Clauses	268
17. Working with Numbers.....	269
The Numeric Types	269
The xs:decimal Type	269
The xs:integer Type	269
The xs:float and xs:double Types	270
The xs:numeric Type	270
Constructing Numeric Values	270
The number Function	271
Numeric Type Promotion	271
Comparing Numeric Values	272
Arithmetic Operations	273
Arithmetic Operations on Multiple Values	274
Arithmetic Operations and Types	274
Precedence of Arithmetic Operators	274
Addition, Subtraction, and Multiplication	275
Division	275
Modulus (Remainder)	276
Functions on Numbers	277

Formatting Numbers	279
Formatting Integers	279
Formatting Decimal Numbers	280
The Decimal Format Declaration	280
18. Working with Strings.....	283
The <code>xs:string</code> Type	283
Constructing Strings	283
String Literals	284
The <code>xs:string</code> Constructor and the <code>string</code> Function	284
Comparing Strings	284
Comparing Entire Strings	285
Determining Whether a String Contains Another String	285
Matching a String to a Pattern	286
Substrings	287
Finding the Length of a String	288
Concatenating and Splitting Strings	289
Concatenating Strings	289
Splitting Strings Apart	290
Converting Between Codepoints and Strings	291
Manipulating Strings	291
Converting Between Uppercase and Lowercase	291
Replacing Individual Characters in Strings	292
Replacing Substrings That Match a Pattern	292
Whitespace and Strings	294
Normalizing Whitespace	294
Internationalization Considerations	295
Collations	295
Unicode Normalization	297
Determining the Language of an Element	297
19. Regular Expressions.....	299
The Structure of a Regular Expression	299
Atoms	299
Quantifiers	299
Parenthesized Sub-Expressions and Branches	300
Representing Individual Characters	301
Representing Any Character	303
Representing Groups of Characters	303
Multi-Character Escapes	304
Category Escapes	304
Block Escapes	305

Character Class Expressions	306
Single Characters and Ranges	306
Subtraction from a Range	307
Negative Character Class Expressions	307
Escaping Rules for Character Class Expressions	308
Reluctant Quantifiers	308
Anchors	309
Back-References	310
Using Flags	311
Using Sub-Expressions with Replacement Variables	312
20. Working with Dates, Times, and Durations.	315
The Date and Time Types	315
Constructing and Casting Dates and Times	316
Time Zones	317
Comparing Dates and Times	318
The Duration Types	319
The <code>xs:yearMonthDuration</code> and <code>xs:dayTimeDuration</code> Types	320
Comparing Durations	320
Extracting Components of Dates, Times, and Durations	321
Formatting Dates and Times	322
Using Arithmetic Operators on Dates, Times, and Durations	323
Subtracting Dates and Times	323
Adding and Subtracting Durations from Dates and Times	324
Adding and Subtracting Two Durations	325
Multiplying and Dividing Durations by Numbers	326
Dividing Durations by Durations	326
The Date Component Types	327
21. Working with Qualified Names, URIs, and IDs.	329
Working with Qualified Names	329
Retrieving Node Names	330
Constructing Qualified Names	332
Other Name-Related Functions	333
Working with URIs	334
Base and Relative URIs	334
Documents and URIs	336
Escaping URIs	338
Working with IDs	339
Joining IDs and IDREFs	340
Constructing ID Attributes	341
Generating Unique ID Values	341

22. Working with Other XML Constructs.....	343
XML Comments	343
XML Comments and the Data Model	343
Querying Comments	344
Comments and Sequence Types	344
Constructing Comments	345
Processing Instructions	346
Processing Instructions and the Data Model	346
Querying Processing Instructions	347
Processing Instructions and Sequence Types	347
Constructing Processing Instructions	348
Documents	349
Document Nodes and the Data Model	349
Document Nodes and Sequence Types	350
Constructing Document Nodes	350
Text Nodes	351
Text Nodes and the Data Model	351
Querying Text Nodes	352
Text Nodes and Sequence Types	353
Why Work with Text Nodes?	353
Constructing Text Nodes	355
XML Entity and Character References	355
CDATA Sections	357
23. Function Items and Higher-Order Functions.....	359
Why Higher-Order Functions?	359
Constructing Functions and Calling Them Dynamically	360
Named Function References	360
Using <code>function</code> -lookup to Obtain a Function	361
Inline Function Expressions	361
Partial Function Application	362
The Arrow Operator and Dynamic Function Calls	363
Syntax Recap	363
Functions and Sequence Types	364
Higher-Order Functions	364
Built-In Higher-Order Functions	365
Writing Your Own Higher-Order Functions	366
24. Maps, Arrays, and JSON.....	369
Maps	369
Constructing Maps	369
Looking Up Map Values	371

Querying Maps	375
Changing Maps	375
Iterating over Entries in a Map	376
Maps and Sequence Types	376
Arrays	378
Constructing Arrays	378
Arrays Versus Sequences	379
Arrays and Atomization	380
Looking Up Array Values	380
Querying Arrays	382
Changing Arrays	383
Arrays and Sequence Types	384
JSON	385
Parsing JSON	385
Serializing JSON	386
Converting Between JSON and XML	387
25. Implementation-Specific Features.....	391
Conformance	391
Version Support	392
New Features in XQuery 3.0	392
New Features in XQuery 3.1	393
Setting the Query Context	394
The Option Declaration	395
Extension Expressions	396
Annotations	397
26. XQuery for SQL Users.....	399
Relational Versus XML Data Models	399
Comparing SQL Syntax with XQuery Syntax	401
A Simple Query	401
Conditions and Operators	402
Functions	404
Selecting Distinct Values	405
Working with Multiple Tables and Subqueries	406
Grouping	408
Combining SQL and XQuery	408
Combining Structured and Semi-Structured Data	409
Flexible Data Structures	409
SQL/XML	411

27. XQuery for XSLT Users.....	413
XQuery and XPath	413
XQuery Versus XSLT	413
Shared Components	414
Equivalent Components	414
Differences	415
Using XQuery and XSLT Together	420
XQuery Backward Compatibility with XPath 1.0	421
Data Model	421
New Expressions	422
Path Expressions	422
Function Conversion Rules	423
Arithmetic and Comparison Expressions	423
Built-in Functions	424
28. Additional XQuery-Related Standards.....	425
XQuery Update Facility	425
Full-Text Search	426
XQueryX	428
RESTXQ	430
XQuery API for Java (XQJ)	432
A. Built-in Function Reference.....	435
B. Built-in Types.....	635
C. Error Summary.....	667
Index.....	705

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.

Want to read more?

You can [buy this book](#) at oreilly.com in print and ebook format.

Buy 2 books, get the 3rd FREE!

Use discount code OPC10

All orders over \$29.95 qualify for **free shipping** within the US.

It's also available at your favorite book retailer, including the iBookstore, the [Android Marketplace](#), and [Amazon.com](#).



O'REILLY®