

THE EXPERT'S VOICE® IN SQL SERVER

Pro SQL Server 2008 XML

*The essential guide to managing and programming
with XML in a SQL Server environment.*

Michael Coles

*Foreword by Michael Rys,
Principal Program Manager, Microsoft*

Apress®

Pro SQL Server 2008 XML



Michael Coles

Pro SQL Server 2008 XML

Copyright © 2008 by Michael Coles

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-59059-983-9

ISBN-10: 1-59059-983-7

ISBN-13 (electronic): 978-1-4302-0630-9

ISBN-10 (electronic): 1-4302-0630-6

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

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Altova® and XMLSpy® are trademarks or registered trademarks of Altova GmbH, and are registered in numerous countries.

Lead Editor: Jonathan Gennick

Technical Reviewer: Fabio Claudio Ferracchiati

Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Kylie Johnston

Copy Editor: Kim Benbow

Associate Production Director: Kari Brooks-Copony

Production Editor: Liz Berry

Compositor/Artist: Kinetic Publishing Services, LLC

Proofreader: April Eddy

Indexer: Becky Hornyak

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

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

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

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

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

For Devoné and Rebecca

Contents at a Glance

Foreword	xv
About the Author	xvii
About the Technical Reviewer	xix
Acknowledgments	xxi
Introduction	xxiii
CHAPTER 1 Enter XML	1
CHAPTER 2 FOR XML and Legacy XML Support	17
CHAPTER 3 The xml Data Type	61
CHAPTER 4 XML Schema Collections	83
CHAPTER 5 XQuery	115
CHAPTER 6 XQuery Functions and Operators and XML DML	153
CHAPTER 7 Indexing XML	177
CHAPTER 8 XSLT and the SQLCLR	193
CHAPTER 9 HTTP SOAP Endpoints	231
CHAPTER 10 .NET XML Support	245
CHAPTER 11 Spatial Data and GML	275
CHAPTER 12 SQLXML	295
CHAPTER 13 LINQ to XML	319
CHAPTER 14 XML Support Tools	341
APPENDIX A W3C and Other References	357
APPENDIX B SQL Server XQuery Data Types	361
APPENDIX C XML Schema Reference	365
APPENDIX D XQuery/XPath/XML DML Quick Reference	375
APPENDIX E XSLT 1.0 and XPath 1.0 Reference	381
APPENDIX F Glossary	389
APPENDIX G Selected T-SQL and .NET Code Listings	401
INDEX	447

Contents

Foreword	xv
About the Author	xvii
About the Technical Reviewer	xix
Acknowledgments	xxi
Introduction	xxiii
■ CHAPTER 1 Enter XML	1
Looking Back at SQL Server XML	1
What Is XML?	2
Defining XML Data	4
XML Requirements	6
Well-Formed and Valid XML	8
Considering Other Formats	8
When to Use XML	11
What's New in SQL Server 2008 XML	13
The xml Data Type	14
XML Schema Collections	14
XML Indexes	14
FOR XML	15
XQuery and XML DML Support	15
HTTP SOAP Endpoints	15
Summary	16
■ CHAPTER 2 FOR XML and Legacy XML Support	17
Using the FOR XML Clause	17
PATH Mode	19
RAW Mode	23
AUTO Mode	29
EXPLICIT Mode	34
Using XPath Node Tests	37
Adding Namespaces to FOR XML	45
Creating Complex FOR XML Queries	47
OPENXML Rowset Provider	54

OPENROWSET XML Loading	58
Summary	59
CHAPTER 3 The xml Data Type	61
Creating xml Instances	61
Casting and Converting	63
Using xml Parameters and Return Types	65
Creating Well-Formed and Valid XML	65
XML Schema Collections	66
DTDs	68
Using XML Type Methods	71
Using the query() Method	71
Using the value() Method	73
Using the exist() Method	75
Using the nodes() Method	76
Using the modify() Method	79
Summary	81
CHAPTER 4 XML Schema Collections	83
Introducing XML Schema	83
Documenting with Annotations	85
Using Declaration Components	86
Creating Complex Elements	87
Defining Model Groups	90
Adding Attributes	93
Constraining Occurrences	95
Extending XML Schemas with Wildcards	102
Typing XML	108
Summary	114
CHAPTER 5 XQuery	115
Introducing the XQuery Language	115
Creating XQuery Queries	119
Defining the XQuery Prolog	121
Building Path Expressions	123
Limiting Results with Predicates	126
Using Quantified Expressions	132

Using FLWOR Expressions	133
Constructing XML with XQuery	136
Using the SQL Server xml Methods	139
Querying with query()	139
Retrieving Scalar Values with value()	141
Checking for Node Existence with exist()	142
Shredding XML with nodes()	142
Manipulating XML with modify()	145
Conditional Evaluation with if...then...else	146
Maximizing XQuery Performance	147
Use the value() Method	147
Avoid Reverse Axis Steps	148
Avoid // and Wildcards in the Middle	148
Use Subqueries	149
Avoid Predicates in the Middle	150
Summary	151

■ CHAPTER 6 XQuery Functions and Operators and XML DML 153

Using Operators	154
Calculating with Math Operators	154
Using Comparison Operators	154
Constructing Sequences with the Comma Operator	155
Using XQuery Type Expressions	156
Casting XQuery Values	156
Checking the Instance Type	156
Using XQuery Functions	158
Using Data Accessor Functions	158
Using String Functions	159
Using the Boolean Function	160
Using Numeric Functions	161
Using Aggregate Functions	162
Using Sequence Functions	163
Using Node Functions	165
Using Context Functions	166
Using Constructor Functions	168
Using QName Functions	169
Using SQL Server XQuery Extension Functions	170

Modifying XML with XML DML	171
Inserting Nodes with insert	171
Deleting Nodes with delete	174
Updating Nodes with replace value of	175
Summary	176
 CHAPTER 7 Indexing XML	177
Creating a Primary XML Index	177
Creating Secondary XML Indexes	181
Creating PATH Secondary XML Indexes	182
Creating VALUE Secondary XML Indexes	183
Creating PROPERTY Secondary XML Indexes	185
Setting XML Index Options	187
Full-Text Indexing XML	189
Summary	192
 CHAPTER 8 XSLT and the SQLCLR	193
Transforming XML	193
Accessing XSLT Through .NET	194
Performing a Simple Transformation	199
Elements of XSLT Stylesheets	203
Performing a Back-End Transformation	208
Advanced XSL Transformations	219
The Multitemplate Stylesheet	221
Recursion in the Stylesheet	226
Summary	228
 CHAPTER 9 HTTP SOAP Endpoints	231
Creating Endpoints	232
Consuming Endpoints	240
Summary	244
 CHAPTER 10 .NET XML Support	245
XML Validation	245
Accessing XML on the Web	253
REST Services	256

.NET XML Classes	261
System.Xml Namespace	261
SqlXml Data Type	266
SqlCommand Options	269
Additional .NET XML Support	272
Summary	274
 CHAPTER 11 Spatial Data and GML	275
Spatial Data	275
Populating Spatial Data	276
GML	280
Geometric Objects	280
Elements of GML	286
Summary	293
 CHAPTER 12 SQLXML	295
Querying	295
Updategrams	298
Inserts	303
Updates	305
Deletes	306
Executing Updategrams with SqlXmlCommand	307
Diffgrams	310
Bulk Loading	310
Querying SQLXML with XPath	314
Summary	317
 CHAPTER 13 LINQ to XML	319
Functional Construction	319
Loading XML from Other Sources	321
Loading XML with the XmlReader	321
Querying with LINQ to SQL	323
Loading XML from the File System	326
Loading XML from a String	327
Loading XML via HTTP	329
Querying XML	330
Transforming XML	337
Summary	339

CHAPTER 14	XML Support Tools	341
	Bulk Copy Program	341
	XML for Analysis	343
	SQL Server Integration Services	345
	XML Query Plans	347
	Database Tuning Advisor	349
	XMLSpy	350
	Web Browsers	352
	Visual Studio	354
	Summary	354
APPENDIX A	W3C and Other References	357
	W3C Specifications	357
	Other Useful Documents	359
APPENDIX B	SQL Server XQuery Data Types	361
APPENDIX C	XML Schema Reference	365
	Element Information Items	365
	all Element	366
	annotation Element	366
	any Element	367
	anyAttribute Element	367
	applInfo Element	367
	attribute Element	367
	attributeGroup Element	368
	choice Element	368
	complexContent Element	368
	complexType Element	368
	documentation Element	369
	element Element	369
	extension Element	369
	group Element	370
	import Element	370
	list Element	370
	notation Element	370
	restriction Element	370
	schema Element	371

sequence Element	371
simpleContent Element	371
simpleType Element	372
union Element	372
XML Schema Data Type Facets	372
■ APPENDIX D XQuery/XPath/XML DML Quick Reference	375
XPath	375
XQuery	377
XML DML	380
■ APPENDIX E XSLT 1.0 and XPath 1.0 Reference	381
■ APPENDIX F Glossary	389
■ APPENDIX G Selected T-SQL and .NET Code Listings	401
Chapter 1	401
Chapter 2	402
Chapter 3	409
Chapter 4	415
Chapter 5	420
Chapter 6	424
Chapter 7	425
Chapter 8	427
Chapter 9	432
Chapter 10	435
Chapter 11	438
Chapter 12	439
■ INDEX	447

Foreword

It is my pleasure to present Michael Coles's book, *Pro SQL Server 2008 XML*, which covers one area of Microsoft® SQL Server™ that I spend a large part of my work at Microsoft designing, influencing, and building: the XML support in SQL Server.

Michael covers this complex topic with the focused understanding of a practitioner and the deep background of an experienced industry observer. He presents this large and, to the general database programmer, often somewhat new and surprising area, both in an easy and logical way—covering the client application, .Net programming with XML, and the database development aspects. Lots of examples provide access to the concepts and technologies, and practical tips about usage and performance add relevance. He even shows how XML is being used with other features in SQL Server, such as the spatial support added in SQL Server 2008, which provides support for a subset of GML, Bulk copy's use of XML and XML for Analysis.

His technical content is presented against a historical background of what XML is about, and also points out that XML is not the cure for all ailments in this world, but that it has its place to address several important scenarios.

When I started to work on the XML support in SQL Server in the SQL Server 2000 release cycle, we set out on a journey to provide extensions to our relational database customers that would enable them to work with XML data to address three different but related scenarios that were united in that they all were using XML at some level.

The first scenario, which we focused on in SQL Server 2000, was the ability to integrate existing relational data into the new world of web service and loosely coupled data exchange that was starting to use XML as their lingua franca of data interchange. Features like FOR XML or the mapping schemas of the SQL/XML component were allowing programmers to map their existing relational data into XML and take structured data from XML into their existing relational database.

Since XML with its tag markup structure is well-suited to describe complex, non-regular data shapes, it also quickly became a preferred way by many to represent data that did not easily fit into the relational mold. XML was used either because the data shape was changing too quickly or was not known a priori, or the decomposition and re-composition costs of the complex properties were too high and XML gave a good compromise between queryability and flexibility. This second scenario, often referred to as the scenario of managing semi-structured data management, got support in SQL Server 2005 with the addition of the XML data type, XML Schema collections, and the support for XQuery to query into the XML structure and unlock the information within it. SQL Server 2008 has now even added relational functionality—such as sparse columns and column sets, themselves based on XML—to provide more “relational” support for semi-structured data.

Finally, over the last few years, more and more documents are being represented in XML, be it custom schemas or some standard document schemas such as the Office OpenXML (not to be confused with the OpenXML function in SQL Server) and others. As such uses for XML become more prevalent, the queryability of the underlying document format, namely XML, will

become more important and the current support for XML in SQL Server 2008 will provide a solid foundation to manage such documents together with the other business data.

Michael's experience, written down in this book, gives you good guidance and insight into how the different SQL Server XML technologies can help you with these and similar scenarios. Whether you are a newcomer to SQL Server's XML Support or a seasoned XML user, you will find lots of value in this book with its practical advice and easy to understand explanations and examples.

Enjoy the book!

Michael Rys
Principal Program Manager, Microsoft

About the Author



■ **MICHAEL COLES** has over a dozen years' experience in SQL database design, T-SQL development, and client-server application programming. He has consulted in a wide range of industries, including the insurance, financial, retail, and manufacturing sectors, among others. Michael's specialty is developing and performance-tuning high-profile SQL Server-based database solutions. He currently works as a consultant for a business intelligence consulting firm. He holds a bachelor's degree in information technology and multiple Microsoft and other certifications.

Michael has published dozens of highly rated technical articles online and in print magazines, including *SQL Server Central*, *ASP Today*, and *SQL Server Standard* magazines. Michael is the author of the book *Pro T-SQL 2005 Programmer's Guide* (Apress, 2007) and a contributor to *Accelerated SQL Server 2008* (Apress, 2008).

About the Technical Reviewer

■ **FABIO CLAUDIO FERRACCHIATI** is a senior consultant and a senior analyst/developer using Microsoft technologies. He works for Brain Force (www.brainforce.com) in its Italian branch (www.brainforce.it). He is a Microsoft Certified Solution Developer for .NET, a Microsoft Certified Application Developer for .NET, a Microsoft Certified Professional, and a prolific author and technical reviewer. Over the past ten years, he's written articles for Italian and international magazines and coauthored more than ten books on a variety of computer topics. You can read his LINQ blog at www.ferracchiati.com.

Acknowledgments

Bringing a book like this to you, the reader, is not a solo act. Even though Apress was kind enough to put my name on the cover, there's no way you would be reading these words if not for the entire team at Apress. This book is the product of the work of dozens of my Apress teammates.

With that in mind, I would like to start by thanking my editor Jonathan Gennick, who pulled this project together and oversaw it from first concept to press. I would also like to thank the chief cat-herder, project manager Kylie Johnston, who made sure everyone did what they were supposed to, when they were supposed to. I want to send a special thank you to the technical reviewer and resident LINQ expert, Fabio Claudio Ferracchiati, for keeping me honest. I would also like to thank those team members who spent countless hours copy editing, laying out pages, and contributing in numerous other ways to bring this book to you.

I would like to send an extra special thank you to Michael Rys, project manager for SQL Server XML technology and member of the W3C XML Query Working Group. Thank you, Mr. Rys, for taking the time to answer all of the wild SQL Server XML questions I kept coming up with. I would also like to thank Steve Jones, entrepreneur extraordinaire of *SQL Server Central* fame and SQL Server MVP, and Chuck Heinzelman, editor in chief of *SQL Server Standard* magazine, for their support.

Finally, I would like to thank my family, including my mom, Eric, Jennifer, Chris, Desmond, and Deja. I'd also like to thank my Aunt Linda and her family for their support. And a special thank you to my girlfriend, Donna, for being so supportive and understanding during this process.

And most important, thank you to Devoné and Rebecca—my little angels—for keeping a smile on my face.

Introduction

Back in 1999, I was working for a dot-com when I was first exposed to the interesting new technology known as XML. My imagination ran wild, and I immediately began working on dozens of applications for XML, like custom reporting from our legacy inventory system and Web-based business-to-business ordering. I was as excited as a kid on Christmas morning!

My excitement about XML in those early days began to fade as I quickly ran into its limitations, however. For one thing, the basic XML standard did not define a type system. All content in XML is character data—strings. This meant that I needed to not only convert string data to the correct data type in my own applications, but I also had to validate the content and handle boatloads of potential exceptions in each application.

Another thing that bothered me was the strangely cryptic and underpowered Document Type Definition (DTD) system, the primary means of constraining XML document structure and content. Again, the majority of structure and content validation became the responsibility of the application, making it even more complicated. And the fact that querying XML data required inefficient loops or event-based parsing and comparisons didn't help matters much.

Fortunately for us, XML has matured with the introduction of standards for constraining and typing XML data and efficiently querying XML content. Several XML-based standards have also been introduced to allow efficient and convenient data sharing across platforms.

With SQL Server 2008, Microsoft has taken a different approach to the concept of the “database.” Rather than confining themselves to a basic relational model, Microsoft has taken the stance that your data should be accessible and manageable in all its glorious (and not so glorious) forms. While relational data remains king, XML is the cornerstone of disparate data integration.

This book provides detailed information on XML from the SQL Server 2008 perspective. I'll discuss several aspects of SQL Server 2008 XML, including the XML functionality built right into SQL Server, as well as how to access SQL Server functionality that is not built into T-SQL. I'll even look at client-side XML technologies that are important to SQL Server/XML-based application design. In all, I hope you find this book enjoyable and useful in designing and implementing your own SQL Server and XML-based applications.

Who This Book Is For

This book is written for SQL Server developers by a SQL Server developer. It's written for anyone who wants to know how to retrieve relational data in XML format, shred XML data back to relational format, use XML Schema to strongly type XML data, use XQuery to query XML data, or perform dozens of other SQL Server XML tasks.

In order to take advantage of SQL Server 2008's XML functionality, you will need a basic understanding of T-SQL. Some of the code samples and concepts in the book utilize the SQL Common Language Runtime (SQLCLR), and some are presented as .NET client code. An understanding of the C# language and the Microsoft .NET Framework is useful, though not required, to utilize these samples.

How This Book Is Structured

Pro SQL Server 2008 XML is written as a guide to SQL Server's built-in XML functionality and tools. This book is written for two types of readers:

- The developer who is not familiar with prior SQL Server XML functionality will get the most out of reading the book cover to cover.
- The developer who is familiar with prior implementations of SQL Server XML functionality may get more out of using the book as a reference guide to new features and specific options.

Each chapter of the book addresses a different XML topic, making it easy to locate specific information if you are using it as a reference guide. In each chapter, I've also attempted to build on concepts introduced in prior chapters so that reading the book from start to finish will prove an engaging exercise. Following are brief summaries of each chapter in the book.

Chapter 1

Chapter 1 provides a brief overview of the history of XML, including an overview of the W3C XML recommendation. It's designed to answer the question "What is XML?" Readers who already have knowledge of XML can skip this chapter.

Chapter 2

Chapter 2 discusses the `FOR XML` clause and other legacy SQL Server XML support, such as the `OPENXML` rowset provider. Readers who are well-versed in `FOR XML` may wish to skip this chapter. For those who are already familiar with legacy XML support, this chapter will prove a valuable reference for features like the `FOR XML PATH` clause.

Chapter 3

Chapter 3 serves as the introduction to the SQL Server `xml` data type. This chapter is particularly useful for those readers who have not yet used the `xml` data type in the prior version of SQL Server. In this chapter, I lay the foundation for the discussion of SQL Server 2008's XML functionality, since most of this functionality centers around the `xml` data type. Those readers who are already familiar with the `xml` data type will still want to at least skim this chapter, as I discuss SQL Server's internal management of XML data.

Chapter 4

Chapter 4 builds on the discussion of the `xml` data type in Chapter 3, with an in-depth discussion of XML Schema collections and typed XML. This chapter also compares the SQL Server XML Schema implementation to the W3C standard and describes how to implement your own custom XML schemas. This chapter is a must-read for anyone who wants to use XML Schema to constrain the structure and content of their XML data.

Chapter 5

Chapter 5 builds on the `xml` data type discussion that began in Chapter 3. The `xml` data type methods that allow you to query XML data type instances all rely on XQuery expressions. In this chapter, I will look at the SQL Server implementation of the W3C XQuery recommendation, with a thorough discussion of available XQuery expressions, predicates, and the XQuery/XPath Data Model (XDM). This chapter is a must if you plan to query XML or shred XML data into relational format.

Chapter 6

Chapter 6 continues the discussion of XQuery begun in Chapter 5 by introducing the SQL Server–supported XQuery functions and operators, as well as the SQL Server XML Data Manipulation Language (XML DML) extensions to the standard. As always, the W3C reference is used as the basis for the discussion of SQL Server features. If you plan to manipulate XML data with XML DML or if your intent is to write XQuery expressions that require calculations and functions, you should read both Chapters 5 and 6.

Chapter 7

Chapter 7 provides a thorough discussion of XML indexing in SQL Server. In this chapter, I discuss primary and secondary XML indexes as well as XML full-text indexing. If you plan to store large amounts of XML data as `xml` data within SQL Server, this chapter will teach you how to optimize XQuery performance over your XML data.

Chapter 8

Chapter 8 begins the exploration of extending SQL Server’s XML functionality via the SQLCLR. In this chapter, you will learn how to access a major piece of XML functionality that is not available directly to the `xml` data type—Extensible Stylesheet Language Transformations (XSLT). I also reference the XSLT standard heavily so that you will be able to create your own basic XSL transformations after reading this chapter.

Chapter 9

Chapter 9 discusses SQL Server 2008 HTTP Simple Object Access Protocol (SOAP) endpoints, an XML-based technology that allows you to expose server-side stored procedures and user-defined functions as web service methods. By using HTTP SOAP endpoints, your client applications/web service consumers can access predefined functionality in your database remotely and securely. If you plan on performing cross-platform or web development with a SQL Server back end, this chapter is definitely worth reading.

Chapter 10

Chapter 10 goes into greater depth discussing .NET XML support. With the .NET XML functionality, you can create SQLCLR routines that access remote XML data and services, perform legacy DTD and XML Data-Reduced (XDR) schema validations, and perform client-side manipulations of your SQL Server XML data. If you plan to write SQLCLR XML manipulation and access

routines or you plan to write client-side code that accesses XML stored on SQL Server, then this chapter is for you.

Chapter 11

Chapter 11 discusses Geography Markup Language (GML) support built into SQL Server 2008's new spatial data types, geography and geometry. This chapter is an interesting diversion for those interested in the new spatial data types and an absolute must-read for those interested in sharing spatial data in XML format.

Chapter 12

Chapter 12 provides a detailed discussion of the functionality available through the COM-based SQLXML API. SQLXML provides access to an array of functions like XML bulk load, updategrams and diffgrams, client-side FOR XML formatting, and XSLT-style querying of relational data. Those who support legacy applications that use this functionality should definitely read this chapter, as should those who are interested in SQLXML's functionality.

Chapter 13

Chapter 13 is an overview and discussion of the new .NET Language-Integrated Query (LINQ to XML). LINQ to XML is a powerful method of querying XML data using a SQL-like syntax directly from C# or Visual Basic. LINQ to XML is definitely an exciting new technology, and I'd recommend that anyone interested in client-side XML development should read this chapter.

Chapter 14

Chapter 14 rounds out the main content of the book with summaries and examples of common XML applications that are useful to SQL Server developers. In this chapter, I discuss SQL Server XML support tools for bulk loading flat files, optimizing SQL queries, and managing SQL Server Analysis Services (SSAS). I also look at support applications that help developers create, edit, and test XML data, like Altova® XMLSpy® XML editor software.

Appendix A

Appendix A is a list of W3C recommendations and other standards that I referenced throughout this book. This short list is useful if you need to quickly locate an XML-specific standard.

Appendix B

Appendix B describes the SQL Server implementation of the W3C XQuery/XPath Data Model and all the built-in data types available to XML Schema and XQuery. This appendix is designed to act as a quick reference, useful if you are creating XML Schema collections or writing XQuery expressions against typed `xml` instances.

Appendix C

Appendix C is a reference to the XML elements that compose an XML schema. Descriptions of all SQL Server–supported XML schema elements are provided, with their available attributes and default values.

Appendix D

Appendix D is a quick reference to XQuery, XPath, and XML DML, as implemented in SQL Server. This appendix is designed to be referenced when you are writing XQuery expressions or XPath location paths for use with the `FOR XML PATH` clause.

Appendix E

Appendix E is a quick reference to XSLT 1.0 and XPath 1.0, as implemented by the .NET Framework. This appendix is designed for use when you are creating custom XSLT stylesheets.

Appendix F

Appendix F is a glossary of the terminology used throughout the book. For those new to XML, many of the terms may be new. Likewise, you may want to follow up and get more information about a specific term than was provided in the inline description. The glossary provided me an opportunity to further expand on many terms, allowing me to add more information than could easily fit in the inline text.

Appendix G

Appendix G rounds out the book with a sampling of selected T-SQL and .NET code listings, chosen specifically to demonstrate key concepts. Where possible, I have added additional descriptive text to the code listings to further explain the options and settings selected, as well as various design decisions.

Conventions

To help make reading *Pro SQL Server 2008 XML* a more enjoyable experience, and to help you get as much out of it as possible, I've used standardized formatting conventions throughout the book.

C# code is shown in a special code font. Note that C# code is case sensitive.

```
while (i < 10)
```

T-SQL source code is shown in code font, with keywords capitalized. Note that data types in the T-SQL code are lowercased to help improve readability.

```
DECLARE @x xml;
```

XML code is shown in code font with attribute and element content in boldface for readability. Note that some code samples and results have been reformatted in the book for easier reading. XML ignores white space so the significant content of the XML has not been altered.

```
<book publisher = "Apress">Pro SQL Server 2008 XML</book>
```

Note Notes, tips, and warnings are displayed in a special font with solid bars placed over and under the content.

SIDEBARS

Sidebars include additional information relevant to the current discussion and other interesting facts. Sidebars are shown on a gray background.

Prerequisites

To make the most of this book, you should have access to SQL Server 2008 and SQL Server Management Studio (SSMS). Alternatively you can use the SQLCMD utility to execute code on SQL Server. You should also download and install the SQL Server 2008 AdventureWorks 2008 database from <http://www.codeplex.com>. All code samples in this book are designed to run on the AdventureWorks 2008 database, unless otherwise stated.

To run sample client applications and to compile and deploy SQLCLR samples, you will need C# 2005 or C# 2008. Note that the Express Editions will compile and run the client code samples, but they will not compile or deploy the SQLCLR source code. For the best overall experience, I highly recommend compiling, deploying, and executing C# code samples from within Visual Studio 2005 or the 2008 IDE.

Downloading the Code

This book contains several code samples demonstrating the concepts presented. All of these code samples are available for download in one ZIP file from the Source Code/Downloads section of the Apress web site. To get the download, go to www.apress.com, click on the Quick Links option on the menu, and then click on Source Code/Downloads.

Contacting the Author

The author and the Apress team have made every effort to ensure that this book is free from errors and defects. Unfortunately, the occasional error does slip past us, despite our best efforts. In the event that you find an error in the book, please let us know! You can submit errors to Apress by visiting www.apress.com, locating the book page for this book, and clicking Submit Errata. Alternatively, feel free to drop a line directly to the author at michaelco@optonline.net.



Enter XML

Welcome to *Pro SQL Server 2008 XML*. This book will cover the basics and advanced topics of SQL Server–based XML development, including a review of legacy XML support in prior versions of SQL Server, and a discussion of new features introduced in SQL Server 2005 and SQL Server 2008. Throughout this book I will discuss the implementation of several advanced XML standards via SQL Server, including XPath, XQuery, XSLT, XML Schema, and XML DML. I will also provide coverage of other advanced XML-related topics, like SQLCLR and client-side .NET XML capabilities and Microsoft’s .NET LINQ to XML technology.

Throughout this book, I will provide step-by-step code samples to demonstrate the concepts presented. All samples are designed to run with the Microsoft AdventureWorks sample database, unless otherwise noted. The sample code from this book is available for download at the Apress web site (www.apress.com/download).

ADVENTUREWORKS SAMPLE DATABASE

As mentioned, the code samples in this book are designed to be run against the Microsoft AdventureWorks sample database, unless otherwise specified in the text. Microsoft has decided to offer the SQL Server 2008 version of the AdventureWorks sample database and applications on its CodePlex web site. The URL is www.codeplex.com/MSFTDBProdSamples. If you don’t yet have the AdventureWorks database installed on a test server, I highly recommend that you visit CodePlex and download it. The AdventureWorks database and business scenarios are documented on the MSDN web site at msdn2.microsoft.com/en-us/library/ms124501.aspx.

In this chapter, I’ll provide a brief background of XML in SQL Server, a quick primer on the World Wide Web Consortium (W3C) XML Recommendation, a comparison of XML to other data formats, and a brief overview of XML functionality in SQL Server 2008.

Looking Back at SQL Server XML

When SQL Server 2000 was released, Microsoft was just beginning a big push to thoroughly immerse its entire product line in XML. XML was integrated into SQL Server 2000 by adding the FOR XML clause to the SELECT statement and adding access to various Component Object Model (COM) components via stored procedures and functions. Some XML support was

provided through integration with Internet Information Services (IIS). XML data in SQL Server 2000 was truly a second-class citizen, driving many developers to avoid using SQL Server for all but the simplest of XML storage and retrieval tasks. The main problems with SQL Server 2000 XML support included the following:

- **Limited functionality.** SQL Server 2000 XML support was provided primarily by the FOR XML clause of the SELECT statement, the OPENXML function, and a couple of stored procedures to create XML documents in memory and remove them when finished. There was no built-in Transact-SQL (T-SQL) support for querying or modifying XML data.
- **Complicated to use.** SQL Server 2000 XML support relied on the old-style Large Object (LOB) data types, including TEXT and NTEXT. SQL Server 2000 LOB data types were kludgy at best.
- **Inefficient implementation.** SQL Server 2000 XML support also relied heavily on COM components and external libraries, making it much less efficient than a “native” solution. Additionally, if you failed to explicitly remove a document from memory, you were likely to cause more than a few server-side memory leaks.

SQL Server 2008 ups the ante by providing efficient native T-SQL support for XML, with XML-centric improvements to T-SQL statements, built-in XPath, XQuery, and XML DML support, the native xml data type, XML indexes, XML views, and more. SQL Server 2008's SQLCLR integration can also help make XML manipulation even more flexible as you'll see in Chapter 8. This book is an in-depth exploration of SQL Server 2008's powerful XML functionality.

What Is XML?

No discussion of SQL Server XML capabilities would be complete without a discussion of the underlying technology, XML. In this section, I'll discuss the W3C XML Recommendation.

XML is designed to be a simple, fast, and flexible text format derived from Standard Generalized Markup Language (SGML), as defined by the ISO (International Organization for Standardization) 8879 standard. The XML Recommendation, and its related recommendations, are maintained by the W3C, a standards body with the mission of developing interoperable technologies for the World Wide Web. The W3C XML 1.0 specification defines a set of rules for adding structure and context to data through the use of markup. In addition to the XML 1.0 specification, the W3C has proposed dozens of additional XML-based specifications to standardize data transfer and sharing between applications, XML processing, querying, sharing, and manipulation. The latest versions of the XML 1.0 and XML 1.1 Recommendations are available at www.w3.org/TR/xml and www.w3.org/TR/xml11, respectively.

Work on the XML recommendation initially began in 1996, when it was chartered by the W3C. Design work on XML continued through 1997, and XML 1.0 became a formal W3C Recommendation in early 1998. Though largely defined as a subset of SGML, XML 1.0 also adapted technology from various other sources, including the Text Encoding Initiative (TEI), Hypertext Markup Language (HTML), and Extended Reference Concrete Syntax (ERCS), among others. During the creation of the XML recommendation, the ISO SGML standard was updated to maintain consistency with XML. The XML 1.1 Recommendation adds support for additional character sets, additional encodings, and extended support for control characters in currently supported encodings. Generally speaking, unless you have a specific need for the capabilities of XML 1.1 (such as Unicode 2.0 or Extended Binary Coded Decimal Interchange Code [EBCDIC])

control character support), it is recommended that you use XML 1.0. SQL Server 2008 supports the XML 1.0 Recommendation.

RECOMMENDATIONS VS. STANDARDS

In W3C terms, a “recommendation” is equivalent to a “standard” put forth by a sanctioned standards organization like ISO or ANSI (American National Standards Institute). Presumably the W3C chose to describe their work in terms of recommendations instead of standards because the W3C is a voluntary organization with no power to enforce acceptance of their standards. Individuals, organizations, and standards bodies are free to accept or ignore W3C recommendations at will. XML and its related recommendations have, however, been adopted as industry standards.

The XML 1.0 Recommendation was created with a very specific set of design goals in mind. The following is a summary of these goals:

- **XML should be easy to process.** XML can be processed with simple custom-made string parsers, although there are a wide variety of prebuilt parsers freely available to facilitate XML processing.
- **XML should be straightforward to use over the Internet.** XML is created with plain text, generally using one of several predefined standardized character sets (UTF-8, UTF-16, and so on). It is designed for easy transmission through firewalls and over the Internet using standardized Internet protocols like HTTP (Hypertext Transfer Protocol). Binary formatted data often requires heavy manipulation for transmission using HTTP (or other protocols) over the Internet.
- **XML should be human legible.** XML is, by its plain-text nature and self-documenting structure, easy for humans to read. This makes debugging problematic XML easier than debugging binary data files.
- **XML should be easy to create.** Unlike proprietary binary data formats, XML can be easily created by anyone with a simple text editor, a more complex XML-specific editor, or an automated process.
- **XML should support a wide variety of applications.** Because it is designed to be flexible and extensible, and because it inherently supports international character sets, a very wide variety of applications can use XML.
- **XML standards should be formal and concise.** This design goal was created to ensure that the XML standard was “programmer-friendly.” The idea behind this goal required eliminating “consultant-speak” and “pretty-talk” from the standard and instead providing formal notations and syntax that XML implementers could use to create standardized products quickly and efficiently.
- **XML should be compatible with SGML.** XML was designed with cooperation of the SGML standard committee, so XML is compatible with the SGML standard. In fact, compatibility was so important that some portions of the SGML standard were changed to ensure compatibility during development of the XML standard.

- **XML should have a minimal amount of optional features, ideally zero.** This design goal was a response to SGML's history of adding optional features in an attempt to make SGML as general-purpose as possible. The problem is that these optional features often made document interchange impossible. XML eliminates these types of optional features; any XML parser should be able to read any XML document as long as it follows the standard.
- **XML should be designed and standardized quickly.** This goal was adopted in order to put a standard in place before the big software developers adopted a wide range of conflicting proprietary standards to accomplish similar goals. Work began on XML in mid-1996, with the first working draft presented later that year. XML 1.0 was adopted as a W3C Recommendation in early 1998.

Note I've presented these design goals in order of importance, as I see it anyway, for SQL Server–based XML programmers. This order differs from the order in which they are presented in the XML Recommendation and may not reflect others' view of the importance of each.

To meet these goals, the designers of XML made several design decisions, including the decision that terseness of marked up XML was not important. This directly contradicts many other formats that strive to store their data in terse and compact formats. As a result, XML data often has a proportionately large quantity of markup information included in it. The nature of XML data, however, does tend to make it highly compressible by modern data compression algorithms, if storage space is a high priority.

Defining XML Data

XML data is composed of several types of items:

- The Document Type Definition (DTD) is a special structure used to define entity declarations and structure validation information (note that SQL Server XML does not support the validation aspect).
- XML elements are containers for character data (CDATA) content in XML documents. Each XML element is defined by matching start and end tags used to encapsulate other XML elements and data. XML elements provide structure to XML data.
- XML attributes are closely tied to elements. Attributes provide additional context, content, and metadata to your XML markup data.
- XML comments are denoted by `<!--` and `-->` delimiters. XML provides support for comments to allow developers to add human-readable documentation to their XML data.
- XML processing instructions are marked by `<?` and `?>` delimiters. A processing instruction is a means to provide additional metadata to a processing application.
- XML character references and entity character references are constructs that allow you to insert special characters in your XML data.

Consider the simple XML document in Listing 1-1, which represents a simple selection of high-grossing movies in XML format.

Listing 1-1. *Sample Movies XML Document*

```
<?xml version="1.0" encoding="UTF-16"?>
<!-- High-grossing movie listing -->
<movies>
  <film>
    <?style superhero?>
    <name>Spider-Man</name>
    <releaseDate>2002-05-03-05:00</releaseDate>
    <gross area="world-wide">821706375.00</gross>
    <gross area="domestic">403706375.00</gross>
    <director>Sam Raimi</director>
    <cast>
      <actor>Maguire, Tobey</actor>
      <actor>Dafoe, Willem</actor>
      <actor>Dunst, Kirsten</actor>
    </cast>
  </film>
  <film>
    <?style superhero-sequel?>
    <name>Spider-Man 2</name>
    <releaseDate>2004-06-30-05:00</releaseDate>
    <gross area="world-wide">783924485.00</gross>
    <gross area="domestic">373585825.00</gross>
    <director>Sam Raimi</director>
    <cast>
      <actor>Maguire, Tobey</actor>
      <actor>Franco, James</actor>
      <actor>Dunst, Kirsten</actor>
      <actor>Molina, Alfred</actor>
    </cast>
  </film>
  <film>
    <?style superhero-sequel?>
    <name>Spider-Man 3</name>
    <releaseDate>2007-05-04-05:00</releaseDate>
    <gross area="world-wide">888977494.00</gross>
    <gross area="domestic">336027292.00</gross>
    <director>Sam Raimi</director>
    <cast>
      <actor>Maguire, Tobey</actor>
      <actor>Dunst, Kirsten</actor>
      <actor>Franco, James</actor>
      <actor>Church, Thomas Haden</actor>
    </cast>
  </film>
</movies>
```

```
</film>
</movies>
```

This simple example of a well-formed XML document includes several elements, including the root element `<movies>`, the `<film>` elements nested within it, and the subelements nested within them. This hierarchical structure is standard fare for XML, although XML data does not have to have a strongly regular structure as in the example. For instance, you could have `<cast>` (or other) elements outside of the `<film>` elements if it made sense for your application.

The example also includes XML comments, which are included within the `<!--` and `-->` comment indicators. XML comments are nodes that are processed by XML parsers like other XML nodes. Comment nodes are normally not used by applications during processing because they contain human-readable comments. Finally, the example also contains processing instructions that can be used by the application during processing.

XML Requirements

The sample XML in Listing 1-1 does not include any declarations. Declarations are created via DTDs. SQL Server supports a very small subset of DTDs, allowing you to expand entity references in your XML data and assign default values to attributes. I will discuss DTDs further in Chapter 3.

Note SQL Server does not use DTDs to constrain the format of XML data or the content of elements and attributes. This is a common usage of DTDs as defined by the W3C XML 1.0 Recommendation. To constrain the content and structure of your XML, use XML schema collections instead, which are a much more powerful solution.

I also did not include character references in the example. Character references come in two forms: *character entity references* and *numeric character references*. XML defines a small set of predeclared character entity references so that you can include otherwise reserved characters in your XML data. The process of converting special characters in XML to character references is known as *entitizing*. XML data that contains nonentitized special characters will cause XML parsers to reject the XML during processing. Table 1-1 lists the predeclared character entity references supported by XML.

Table 1-1. XML Predeclared Character Entity References

Entity	Description
&	The & entity is used when you want to include the ampersand (&) in your XML data.
<	The < entity is used when you want to include the less-than sign (<) in your XML data.
>	The > entity is used when you want to include the greater-than sign (>) in your XML data.
'	The ' entity is used when you want to include the apostrophe (') in your XML data.
"	The " entity is used when you want to include the quotation mark (") in your XML data.

Numeric character references look similar to character entity references, but instead of a name, they are represented by `&#` followed by a decimal or hexadecimal number and a semi-colon. Numeric character references can be used to represent any valid Unicode character, even those that already have a predeclared character entity reference. The less-than character (`<`) can be represented using any of the following character references in your XML data:

```
&lt;
&#60;
&#x3c;
```

The first character reference is the predeclared character entity reference for the less-than sign. The second is the decimal numeric character reference for the same character. The final example shows the hexadecimal version of the numeric character reference for that character. Notice that the hexadecimal version has a lowercase `x` character between the number sign (`#`) and the first hexadecimal digit of the character reference. The lowercase `x` indicates that the numeric character reference is hexadecimal instead of decimal. Table 1-2 shows a small selection of common numeric character references.

Table 1-2. *Sample of Common Numeric Character References*

Description	Symbol	Code
quote	"	<code>&#x0022;</code>
ampersand	&	<code>&#x0026;</code>
less-than	<	<code>&#x003c;</code>
greater-than	>	<code>&#x003e;</code>
cent sign	¢	<code>&#x00a2;</code>
pound symbol	£	<code>&#x00a3;</code>
yen symbol	¥	<code>&#x00a5;</code>
copyright	©	<code>&#x00a9;</code>
registered	®	<code>&#x00ae;</code>
degrees	°	<code>&#x00b0;</code>
plus-minus	±	<code>&#x00b1;</code>
superscript-2	²	<code>&#x00b2;</code>
superscript-3	³	<code>&#x00b3;</code>
fraction-1/4	¼	<code>&#x00bc;</code>
fraction-1/2	½	<code>&#x00bd;</code>
fraction-3/4	¾	<code>&#x00be;</code>
times	×	<code>&#x00d7;</code>
divide	÷	<code>&#x00f7;</code>
pi	π	<code>&#x03c0;</code>
ndash	—	<code>&#x2013;</code>
mdash	—	<code>&#x2014;</code>
euro symbol	€	<code>&#x20ac;</code>
trademark	™	<code>&#x2122;</code>

XML VS. HTML

HTML coders will recognize similarities between XML and HTML immediately. Both are markup languages based on subsets of the grand-daddy of markup languages, SGML. Their common ancestry means they share similar element, attribute, and comment delimiters. In both markup languages elements define the overall structure of the document. That's where the similarity ends, however.

These two markup languages are designed for completely different purposes. The purpose of HTML is to format data for display. HTML is a standard with predefined tags and attributes, is not case sensitive, and does not preserve white space. XML, on the other hand, is designed to format and structure data. XML carries no requirement to carry additional formatting information, and it has no predefined tags—you create your own tags based on your XML application. XML is also case sensitive and preserves white space.

The Extensible HTML (XHTML) Recommendation is an XML application that redefines HTML in terms of XML. Because it is based on XML, XHTML is stricter than plain HTML in terms of structure, format, and case sensitivity; although XHTML does support less strict validation modes for backward-compatibility purposes.

Well-Formed and Valid XML

XML data comes in one of two basic forms. XML data can be represented as a fragment or it can be a well-formed document. The SQL Server `xml` type can handle both forms of XML data automatically. XML data must meet the following criteria to be considered well-formed:

1. The XML data must contain one or more elements.
2. The XML data must contain one, and only one, root element. This is the element that contains all other elements within the XML document.
3. All elements within the XML document must be properly nested within one another.

XML structure and content can further be constrained by assigning an `xml` instance to an XML schema collection. The XML schema collection contains XML schemas that are defined per the W3C XML Schema Recommendation. XML schemas provide a flexible and powerful tool for constraining XML data. I will detail SQL Server support for the W3C XML Schema Recommendation in Chapter 4.

Note Although the XML recommendation specifies using DTDs for simple XML document validation, SQL Server supports only a limited subset of DTDs. SQL Server does not support DTD XML structure and content validation.

Considering Other Formats

Although this book is about SQL Server XML functionality, I don't want you to walk away with the idea that XML is the only game in town. The sample data I provided in Listing 1-1 can always be represented in a more terse and compact format (remember in XML, terseness is not considered important). For instance, the same data represented in the common Comma Separated Values (CSV) format used by Microsoft Excel might look like Listing 1-2.

Listing 1-2. Sample Movie Data in CSV Format

```
Spider-Man,2002-05-03-05:00,821706375.00,403706375.00,Sam Raimi,"Maguire,
Tobey","Dafoe, Willem","Dunst, Kirsten",""
Spider-Man 2,2004-06-30-05:00,783924485.00,373585825.00,Sam Raimi,"Maguire,
Tobey","Franco, James","Dunst, Kirsten","Molina, Alfred"
Spider-Man 3,2007-05-04-05:00, 888977494.00, 336027292.00,Sam Raimi,"Maguire,
Tobey","Dunst, Kirsten","Franco, James","Church, Thomas Haden"
```

The CSV format, though much more compact than the XML version, removes the context, structure, and self-documenting tags provided by the XML. It can also be more difficult to expand the CSV version to include more fields, add optional data to records, or modify and troubleshoot the existing format. For instance, consider Listing 1-3, which adds an additional `<actor>` element, expands the `<actor>` elements of the XML version to include an `actor_id` attribute and a list of additional movies some actors have acted in.

Listing 1-3. Modified Sample XML

```
<?xml version="1.0" encoding="UTF-16"?>
<!-- High-grossing movie listing -->
<movies>
  <film>
    <?style superhero?>
    <name>Spider-Man</name>
    <releaseDate>2002-05-03-05:00</releaseDate>
    <gross area="world-wide">821706375.00</gross>
    <gross area="domestic">403706375.00</gross>
    <director>Sam Raimi</director>
    <cast>
      <actor actor_id="MA011">Maguire, Tobey
        <movie>Spider-Man 3</movie>
        <movie>The Good German</movie>
        <movie>Spider-Man 2</movie>
        <movie>Seabiscuit</movie>
      </actor>
      <actor actor_id="DA982">Dafoe, Willem
        <movie>Clear and Present Danger</movie>
        <movie>Mississippi Burning</movie>
        <movie>Platoon</movie>
      </actor>
      <actor actor_id="DU208">Dunst, Kirsten
        <movie>Spider-Man 3</movie>
        <movie>Interview with the Vampire</movie>
      </actor>
    </cast>
  </film>
</movies>
```

These modifications to the XML data make it harder (though not impossible) to represent the data as a single CSV file. CSV format also eliminates additional information about the contents, including the hierarchical structure of the data, comments, and the character set encoding information. Every time the structure of the data changes (e.g., more fields are added or fields are rearranged), the application parsing and querying the data must be modified, often significantly. Finally, the simple CSV format in the example requires you to reference fields by their ordinal position, which can lead to subtle errors and bugs that are difficult to troubleshoot.

The XML version of the data, on the other hand, allows you to query the data elements by name and relative position. For instance, to retrieve the director name of the movie *Spider-Man* in the XML file, you would essentially say “retrieve the <director> element of the <film> where the <name> is ‘Spider-Man.’”

Bear in mind also that CSV is not a standardized format. It is a de facto format conjured up in the early days of computing. A CSV file created on a Windows PC might not be readable on a UNIX computer, and vice versa. Even CSV files created by different Microsoft programs can prove incompatible with one another. This makes it nearly impossible to create a general-purpose CSV parser. Text-based formats like Windows initialization (.INI) files suffer from similar limitations.

Binary formats, like the proprietary formats used by COM-related technologies and even office productivity applications like Microsoft Excel are much more compact than XML, CSV, and other text-based formats; but they suffer some drawbacks as well. These formats are not as easy to transmit over the Internet using standard protocols like HTTP; each format requires its own proprietary parsing tools to “look inside” the data, and they tend to be much harder to modify and debug. Figure 1-1 shows a portion of the binary data that composes the sample in Excel worksheet format.

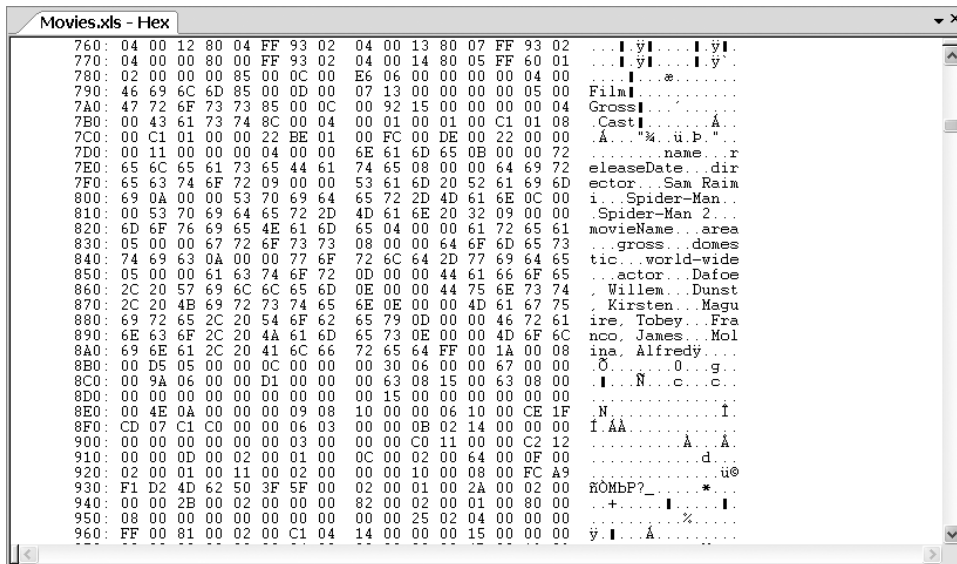


Figure 1-1. Sample Excel spreadsheet binary data

Since we're using SQL Server, it's important to note that data is often better represented using the relational model. The XML sample presented in this chapter was chosen specifically because it is relatively easy to convert to relational format. Figure 1-2 shows the sample data when converted to tables in a relational database.

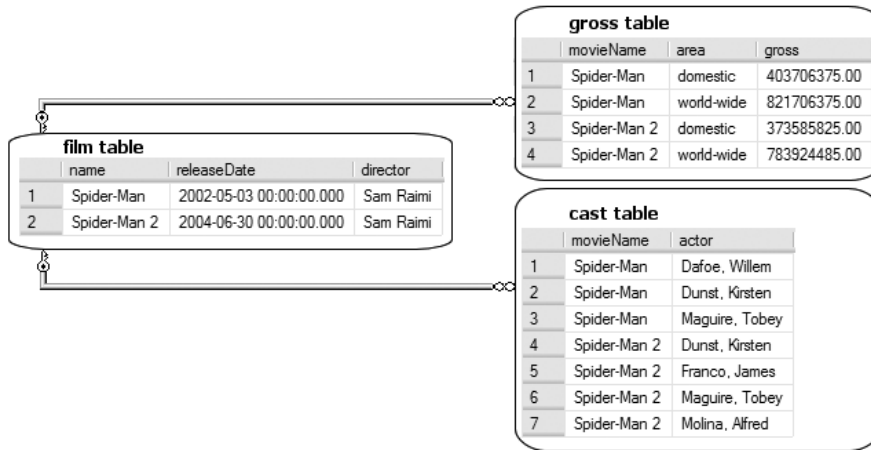


Figure 1-2. Sample data represented in relational database tables

Whether your data is better represented using XML or the relational model is a question that must be answered on a per-project basis, guided by your business rules and requirements. Keep in mind that these two choices do not have to exist exclusive of one another, however. SQL Server offers tools to convert relational data to XML and vice versa, and to manipulate and query both relational data and XML. In the next section, you'll look at some things to take into consideration when deciding whether or not XML is the best choice for representing your data.

When to Use XML

When should you choose XML format over the other possibilities? Some of the factors that can help guide your decision toward using XML to represent data include the following:

- You need a platform-independent model that requires a portable data format.
- Your data is inherently ordered, and the position of data elements provides additional context. Hierarchical data is a prime example where the position of data elements provides context.
- You anticipate querying or updating your data based on its structure.
- Your data is sparse or you anticipate significant structure changes.
- Your data is semi-structured; for example, it has no fixed schema, an implicit or irregular structure, or is nested and heterogeneous.
- Your data represents a containment hierarchy.

SQL Server 2008's native XML manipulation and querying facilities are particularly useful in the following situations:

- You want to manipulate or share XML data while taking advantage of SQL Server's transactional capabilities.
- You want to take advantage of SQL Server's administrative functionality to back up, recover, or replicate your XML data.
- You need to ensure that your server-side XML data is valid and well-formed.
- You want your XML and relational data to interoperate.
- You want to take advantage of SQL Server's XQuery and XPath capabilities.
- You want to optimize your queries of XML data using indexes and SQL Server's query optimizer.

XML is a particularly good choice as a communication format for loosely coupled systems or when trying to make legacy systems communicate with modern servers. It's also an excellent choice when storing data for applications that lend themselves to using data in a marked-up format, such as document management and presentation applications. Modeling semistructured data in an ad hoc fashion also lends itself well to using XML. There are many standard applications for XML already defined in almost every industry, so XML is often an excellent choice when standard communication formats and data sharing are high priorities.

Whether, and when, to use XML is the subject of much debate among SQL Server professionals. There are situations where XML is not the best tool for the job, including the following:

- Your data is highly structured and dense (non-sparse).
- Order is unimportant to your data.
- You do not want to query data based on its structure.
- Your data is best represented using entity references.

Additionally, if you do not intend to query or manipulate your XML data on SQL Server, but just want to store and retrieve it, you should use the `varchar` or `nvarchar` data types instead of SQL Server's `xml` data type. This is most often the case when you simply use SQL Server as a storage repository for your XML data and perform all your XML querying and manipulation in the middle tier or in a client-side application. Also store your XML data as `varchar` or `nvarchar` if you need to store it exactly, as a character-for-character copy, for auditing purposes or for regulatory compliance. This is a common scenario when you are storing data about digital financial transactions, and you need to maintain an audit trail of the transactions but do not need to manipulate or query the data in the XML.

EXACTLY XML

The `xml` data type does not necessarily store an exact character-for-character copy of your XML. The XML Schema Recommendation specifies that XML is converted to an abstract tree-like representation, known as an XML Information Set (Infoset) for persistence or querying purposes. The XQuery Data Model (XDM) Recommendation builds on the XML Infoset and extends the XML Schema model even further. The Infoset allows for loss of characters or other internal manipulations on the XML elements, so long as the meaning of the content is not lost or distorted. In some situations, like regulatory compliance (e.g., compliance with the Sarbanes-Oxley Act of 2002) and auditing, maintaining just the “meaning” is not good enough. You must maintain literal copies of your data. In instances where this is a requirement, it makes sense to store your XML data using the `varchar` or `nvarchar` data type, not the `xml` data type. I will discuss the XQuery 1.0 and XQuery 2.0 Data Model in more detail later in Chapter 4, during the discussion of XML schema collections.

Some database developers and administrators are vehemently opposed to storing XML data in a relational database. Fortunately for them, they do not necessarily have to store XML in the database in order to take advantage of SQL Server’s XML querying and manipulation capabilities. The `xml` data type can be used to declare variables that can be used for querying, validating, and manipulating XML data server side. It can also be used to declare parameters for functions and stored procedures.

What’s New in SQL Server 2008 XML

SQL Server 2008 provides several enhancements over SQL Server 2000 in terms of XML support and some enhancements over SQL Server 2005. While much of the backward-compatible XML-specific functionality from SQL Server 2000 is available in SQL Server 2008, most of it has been deprecated in favor of the new features and functionality. This section gives a broad overview of the major enhancements to XML support, which include the following items:

- New `xml` data type
- XML schema collections
- XML indexes
- `FOR XML` enhancements, including XPath support in the `FOR XML PATH` clause
- XQuery and XML DML support
- SQLCLR `xml` data type support
- Improvements to legacy XML functionality, including improvements to the `sp_xml_preparedocument` procedure
- HTTP Simple Object Access Protocol (SOAP) endpoints

In addition, there have been enhancements made to the `xml` data type implementation over the SQL Server 2005 version, including the following items:

- Full support for the XML Schema `xs:date`, `xs:time`, and `xs:dateTime` data types has been added in SQL Server 2008.
- Support has been added for XML Schema–defined lists of unions and unions of lists.
- Improvements to XML Schema union types have been implemented.
- Support has been added for XQuery FLWOR expression `let` clauses.

SQL Server 2008's SQLCLR functionality also provides extensive support for XML via .NET Framework 2.0 classes. With .NET 2.0, enhancements were made to existing .NET 1.1 classes, and new classes and features have been added. These new features are discussed in greater detail in Chapter 8.

The `xml` Data Type

Prior to SQL Server 2005, SQL Server provided extremely limited support for storing, managing, and manipulating XML data. SQL Server 2000 implemented its XML capabilities through implementation of the `FOR XML` clause and kludgy LOB data type operations combined with specialized system-stored procedures. SQL Server 2005 introduced the `xml` data type, promoting XML data storage and manipulations to first-class status in SQL Server.

The `xml` data type remains one of the most important XML-specific features in SQL Server 2008. The `xml` data type supports the storage of typed XML documents and fragments that have been validated against an XML schema collection and untyped XML data which has not. The `xml` data type can be used to declare columns in a table, T-SQL variables, parameters, and as the return type of a function. Data can also be cast to and from the `xml` data type. In addition, the `xml` data type brings with it a set of methods useful for querying, shredding, and manipulating XML data. The `xml` data type is described in detail in Chapter 3.

XML Schema Collections

XML schema collections provide the ability to validate your documents for well-formedness and validity according to XML schemas that follow the W3C XML Schema standard. SQL Server 2008 provides support for creating server-side XML schema collections that can be used to validate XML documents for structure and data typing. A very powerful feature, XML schema collections and the W3C XML Schema Recommendation are discussed in detail in Chapter 4.

XML Indexes

In the SQL Server XML model, whenever you query or manipulate XML data, the data is first converted to a relational format in a process known as *shredding*. This process can be time consuming when manipulating large XML documents or when querying large numbers of `xml` data type instances. SQL Server 2008 supports indexing of `xml` data type columns. Indexing `xml` columns helps the SQL optimizer significantly improve query performance on XML data stored in the database. The performance is improved by building an index of your XML data by converting it to a relational format, a process known as *preshredding*. The XML index preshredding process

eliminates the shredding step during a query or XML data manipulation, resulting in much faster and less resource-intensive XML query operations. New DML statements have been added to T-SQL to make XML index management relatively easy. XML indexes will be discussed in detail in Chapter 7.

FOR XML

SQL Server includes improvements to the legacy `FOR XML` clause. One improvement is tighter integration with the new `xml` data type, including options to generate native `xml`-typed results. `FOR XML` results can be assigned to variables of the `xml` data type, with additional support for nesting `FOR XML` queries, an improvement on the SQL Server 2000 `FOR XML` clauses, which were limited only to the top level of a `SELECT` statement. The `FOR XML PATH` mode, also carried over from SQL Server 2005, is an improvement over the legacy `FOR XML EXPLICIT` mode. With built-in support for XPath-style expressions, `FOR XML PATH` makes generating XML in explicit structures much easier than was possible in SQL Server 2000.

The `FOR XML RAW` mode has also been improved with additional features, including the ability to rename the default row element name, the ability to explicitly specify the root node, and the ability to retrieve your data in an element-centric format. The `FOR XML AUTO` and `FOR XML EXPLICIT` modes have also been improved with additional options and settings.

While some options have been deprecated, several additional options have been added to the `FOR XML` clauses since the SQL Server 2000 version, including the `ELEMENTS XSINIL` option, which generates elements for NULLs in the result set, and `XMLSCHEMA`, which generates an inline XML Schema Definition (XSD) in your XML result. The power of the `FOR XML` clause will be explored in detail in Chapter 2.

XQuery and XML DML Support

The new `xml` data type provides several methods to allow querying and modification of XML data. These new methods, including the `query()`, `value()`, `exist()`, `nodes()`, and `modify()` methods, support XQuery querying, XML shredding, and XML DML manipulation of your XML data. The SQL Server 2008 XQuery implementation is a powerful subset of the W3C XML Query Language specification, featuring support for path expressions, `FLWOR` (for-let-where-order-by-return) expressions, standard functions and operators, and XML DML statements. XQuery will be discussed in depth in Chapter 5, and supported XQuery functions and operators and XML DML will be covered in Chapter 6.

HTTP SOAP Endpoints

A powerful feature introduced in SQL Server 2005, SQL Server 2008 continues providing support for native HTTP SOAP endpoints, which use the XML-based SOAP protocol. HTTP SOAP endpoints provide an efficient, secure, easy-to-configure option for providing SQL Server-based web service support. The built-in HTTP SOAP endpoint support makes it much easier to expose SQL Server functionality as web services than was previously possible via the Internet Information Server (IIS)-based web services available in SQL Server 2000. We will discuss HTTP SOAP endpoints in Chapter 9.

■ **Note** HTTP SOAP endpoints are not available in SQL Server 2008 Express Edition.

Summary

SQL Server 2008 XML functionality includes all of the functionality that was introduced with SQL Server 2005, improved legacy SQL 2000–compatible functionality, and dozens of new features and enhancements to existing functionality. This chapter provided an overview of XML, its history, and improvements to XML support in SQL Server 2008.

I talked about how XML stacks up to other data formats and looked at some of the items that you should consider when deciding whether or not XML fits in with your design goals. Along the way I contrasted XML to HTML, and considered instances when it makes sense to store your data in strict character format instead of using the `xml` data type. I also briefly considered alternatives to XML for data storage, manipulation, and transmission, and I also provided some general guidelines to help determine when and where SQL Server's XML capabilities can be useful.

In order to understand where you're going, it helps to know where you came from. With that in mind, I provided a brief overview of the history of SQL Server XML support going back to SQL Server 2000 functionality. I then looked forward with a summary of SQL Server 2008 XML enhancements, including the `xml` first-class data type, the enhanced `FOR XML` clause, XML schema collections, XML indexes, XQuery and XML DML support, and HTTP SOAP endpoints. Each of these topics, and much more, will be discussed in great detail in later chapters.

In the next chapter, I'll discuss the `FOR XML` clause and support for other legacy XML functionality, which was introduced in SQL Server 2000.



FOR XML and Legacy XML Support

SQL Server 2008 supports XML generation from relational data via the FOR XML clause of the SELECT statement. The FOR XML clause is a flexible and versatile way to quickly and easily construct XML from your relational data. SQL Server 2008 continues the tradition of FOR XML clause support, including the following enhancements to the FOR XML clause:

- The addition of PATH mode simplifies, and adds flexibility to, the process of specifying an explicit structure for your XML data.
- The XSINIL and ABSENT modifiers for the ELEMENTS option provide fine-grained control over SQL NULL representation in your XML result.
- The TYPE option for converting your result XML to the xml data type.
- Support is included for nested FOR XML queries.

SQL Server 2008 also provides legacy support for the FOR XML EXPLICIT clause, which allows you to specify an explicit structure for your resultant XML, and the OPENXML function, which converts XML documents to relational form.

In this chapter, I'll discuss the many faces of the FOR XML clause and the OPENXML function. I'll also discuss the OPENROWSET rowset provider in this chapter.

Using the FOR XML Clause

Since the 2000 release, SQL Server has provided built-in support for converting your relational data to XML format quickly and easily with the SELECT statement's FOR XML clause. The FOR XML clause provides a simple, powerful, and flexible tool for relational-to-XML data conversions. SQL Server 2008 provides several enhancements to the FOR XML clause, which I will discuss in detail in this section. First though, I'll provide an introduction and overview to the FOR XML clause and its modes.

The FOR XML clause appears at the end of a SELECT statement. It provides four modes of XML generation—RAW, AUTO, PATH, and EXPLICIT. Each conversion mode has its own set of possible options, and each mode provides a trade-off between simplicity and the level of control allowed over the structure and content of the resulting XML document.

While `FOR XML RAW` and `FOR XML AUTO` are useful for quickly generating XML with automatically generated element and attribute names, `FOR XML PATH` provides much greater control and flexibility over the end result. `FOR XML EXPLICIT` also provides a high degree of control over your resulting XML; but this option is deprecated and should not be used for new development. In fact, if you have existing applications that use `FOR XML EXPLICIT`, you might consider assessing the changes required to update it to `FOR XML PATH` as soon as possible. Table 2-1 is a quick summary of `FOR XML` clause usage scenarios.

Table 2-1. *FOR XML Clause Usage Scenario*

FOR XML Clause	Usage Scenarios
<code>FOR XML RAW</code>	<code>FOR XML RAW</code> is useful for ad hoc <code>FOR XML</code> querying or for when the structure of the resultant XML is not known beforehand. The results of <code>FOR XML RAW</code> can change drastically if the structure of the source table changes.
<code>FOR XML AUTO</code>	<code>FOR XML AUTO</code> is also useful for ad hoc <code>FOR XML</code> querying, but for when you wish to use the table names in the resultant XML. This is particularly useful when you need to map the XML result back to the original columns in the source tables. As with <code>FOR XML RAW</code> , the results of <code>FOR XML AUTO</code> can change if the source table changes.
<code>FOR XML PATH</code>	<code>FOR XML PATH</code> is designed for explicitly defining your XML result structure. This is a safer option than the <code>RAW</code> or <code>AUTO</code> modes in production code because you always know the result XML structure in advance, even if the table structure changes.
<code>FOR XML EXPLICIT</code>	<code>FOR XML EXPLICIT</code> is the original method for explicitly defining XML result structure. The operation of <code>FOR XML EXPLICIT</code> is more complex and less intuitive than <code>FOR XML PATH</code> . You should use <code>PATH</code> mode instead of <code>EXPLICIT</code> mode when you want to define an explicit structure for your XML results.

Each of the `FOR XML` modes supports a variety of options which can be included in comma-separated form as part of the `FOR XML` clause. The options include `ROOT`, which adds a root node to your XML with an element name that you specify. Without the `ROOT` option, `FOR XML RAW` generates an XML fragment. You will want to specify the `ROOT` option when you need well-formed XML with a single root node. If you do not need a single root node (XQuery does not require source XML to have a single root node, for instance), then don't worry about adding one. Adding unnecessary nodes to your XML requires additional storage space and can add complexity to XML querying and manipulation.

The `TYPE` option, when specified, returns your `FOR XML` result as an `xml` data type instance. This is particularly useful when you need to nest `FOR XML` queries or assign the result to an `xml` variable or persist it to an `xml` column.

Schema options include `XMLDATA`, which prepends an XML Data-Reduced (XDR) schema to the front of your XML result, and the `XMLSCHEMA` option which inserts an inline XML schema to the beginning of your XML result.

Caution Avoid using the `XMLDATA` option, since it is deprecated and Microsoft has announced that it will be removed from a future version of SQL Server. If you have code that uses the `XMLDATA` option, it's a good idea to begin the process of converting it to use the `XMLSCHEMA` option instead.

Other available options include `BINARY BASE64`, which encodes binary data in Base64 format, and the `ELEMENTS` option, which returns data as subelements instead of attributes. The `ELEMENTS` option supports two additional modifiers, `XSINIL` and `ABSENT`. The `XSINIL` modifier tells FOR XML to represent SQL NULLs with an `xsi:nil = "true"` attribute in the resultant XML. The `ABSENT` feature, which is the default, specifies that NULLs should be eliminated from the XML result.

Figure 2-1 shows the FOR XML clause options supported by each of the FOR XML modes.

FOR XML Clause Options Chart								
	XMLDATA*	XMLSCHEMA	ELEMENTS XSINIL	ELEMENTS ABSENT	BINARY BASE64	TYPE	ROOT	(ElementName')
FOR XML AUTO	●	●	●	●	●	●	●	
FOR XML RAW	●	●	●	●	●	●	●	●
FOR XML PATH			●	●	●	●	●	●
FOR XML EXPLICIT	●				●	●	●	

*The XMLDATA option is deprecated. Use XMLSCHEMA instead.

Figure 2-1. FOR XML clause options by mode

PATH Mode

The FOR XML PATH clause was first introduced in SQL Server 2005, and it continues to be supported in SQL Server 2008. This clause replaces the more complex FOR XML EXPLICIT clause, which I will describe later. Unlike RAW and AUTO modes, PATH mode requires you to explicitly define the structure of your XML result. While it requires more planning and development work on your part, the trade-off is that you gain greater control and flexibility over the generated XML. You can also rest assured that, unlike RAW and AUTO modes, your middle-tier and client-side applications will always know the exact structure of the resultant XML in advance.

PATH mode is also a lot easier to use than the deprecated EXPLICIT mode because it lets you use XPath syntax to explicitly define the structure of your XML result. It also eliminates the need for the odd EXPLICIT mode “bang” syntax and its kludgy “universal table format.” Listing 2-1 demonstrates how to use FOR XML PATH to retrieve information about selected AdventureWorks employees in XML format. The result of this query is shown in Figure 2-2.

Tip Unless otherwise noted, all sample Transact-SQL (T-SQL) queries and Data Manipulation Language (DML) statements are designed to run against the sample AdventureWorks database. See Chapter 1 for information on how to obtain and install the AdventureWorks sample database if you don’t have it already.

Listing 2-1. *Sample FOR XML PATH Query*

```

SELECT emp.NationalIDNumber AS "Employee/@ID",
       emp.HireDate AS "Employee/Hire-Date",
       per.LastName AS "Employee/Name/Last",
       per.FirstName AS "Employee/Name/First",
       per.MiddleName AS "Employee/Name/Middle"
FROM HumanResources.Employee emp
INNER JOIN Person.Person per
    ON emp.BusinessEntityID = per.BusinessEntityID
WHERE emp.BusinessEntityID IN ( 2, 3 )
FOR XML PATH,
    ELEMENTS XSINIL;

```

**Figure 2-2.** *Result of FOR XML PATH query*

In PATH mode, FOR XML uses column names or column aliases as XPath expressions to define the structure and names of XML nodes and attributes in your XML result. In Listing 2-1, the employee's ID column is named `Employee/@ID`, indicating that the value returned by the query is an attribute of the `Employee` element, named `ID`. By contrast, the employee's hire date is named `Employee/Hire-Date`, indicating it is a subelement of the `Employee` node.

Tip Most of the time you will want to use an alias for your column names with `FOR XML PATH`, since the name will usually contain forward slash characters (/), at signs (@), and node test parentheses. You will want to quote these identifiers because of the special characters involved.

As with RAW mode and AUTO mode, the results of PATH mode are not well-formed by default. Also each row is indicated by a row element in the XML result. PATH mode has a very specific set of rules for XPath-style column naming, shown in the following list:

- Column names in the SELECT clause with `FOR XML PATH` are case sensitive, even in a database with a case-insensitive collation. A column named `Employee` is different from a column named `EMPLOYEE` as far as `FOR XML PATH` is concerned.
- Columns in the result set with no name specified are created inline. This means that columns containing xml data are inserted directly in the XML result, while results of other data types are inserted as text nodes. Nameless columns are generated by scalar subqueries and computed columns when no alias is specified.
- Columns with names beginning with an at sign (@) are mapped as attributes of the row element for each corresponding row.
- Columns with names that do not begin with an at sign (@) and do not contain a forward slash (/) are mapped as subelements of the row element.
- Column names that contain one or more forward slashes (/) and do not contain an at sign (@) are mapped as subelements in a hierarchical structure below the row element. As an example, Listing 2-1 includes a column named `Employee/Hire-Date`, which indicates the following hierarchical structure:

```
<Employee>
  <Hire-Date />
</Employee>
```

- Columns that have the same prefix are grouped together as elements in the same hierarchical structure, under the same subelement. Listing 2-1 includes three columns with the same prefix, `Employee/Name/Last`, `Employee/Name/First`, and `Employee/Name/Middle`. These columns are grouped together in the hierarchy like the following:

```
<Employee>
  <Name>
    <Last />
    <First />
    <Middle />
  </Name>
</Employee>
```

- In FOR XML PATH queries, the order of columns in the SELECT clause is important. If a column with a different prefix appears between columns with the same prefix, FOR XML PATH will break its grouping. SQL Server will generate multiple subelements with the same name when a grouping is broken. If, for instance, a column named Employee/Email-Address is placed between the Employee/Name/Last and Employee/Name/First columns, the hierarchy will be formed like this:

```
<Employee>
  <Name>
    <Last />
  </Name>
  <Email-Address />
  <Name>
    <First />
    <Middle />
  </Name>
</Employee>
```

- Columns named with the wildcard asterisk character (*) or the name node() are inserted inline under the row element.
- Columns with names that contain one or more forward slashes (/) and end with an identifier beginning with an at sign (@) are mapped as attributes of the element named before them in the hierarchy. Listing 2-1 includes a column named Employee/@ID, for instance. This indicates an attribute of the Employee element named @ID.

Additionally, FOR XML PATH XPath expressions cannot begin or end with a forward slash, and they cannot contain the // axis step operator. XPath node tests can be used as column names in PATH mode. Node tests are described in detail in the “Using XPath Node Tests” section of this chapter.

The ELEMENTS XSINIL option used in Listing 2-1 tells FOR XML to include NULLs as elements with the xsi:nil = “true” attribute. Employee Roberto Tamburello has a NULL middle name, for instance. When you eliminate the ELEMENTS XSINIL option from the query, as shown in Listing 2-2, FOR XML eliminates NULL elements from the output.

Listing 2-2. *Employee XML Without ELEMENTS XSINIL*

```
SELECT emp.NationalIDNumber AS "Employee/@ID",
       emp.HireDate AS "Employee/Hire-Date",
       per.LastName AS "Employee/Name/Last",
       per.FirstName AS "Employee/Name/First",
       per.MiddleName AS "Employee/Name/Middle"
FROM HumanResources.Employee emp
INNER JOIN Person.Person per
  ON emp.BusinessEntityID = per.BusinessEntityID
WHERE emp.BusinessEntityID IN ( 2, 3 )
FOR XML PATH;
```

The result of removing `ELEMENTS XSINIL` is shown in Figure 2-3. Notice that Mr. Tamburello's middle name element is completely eliminated from the results.

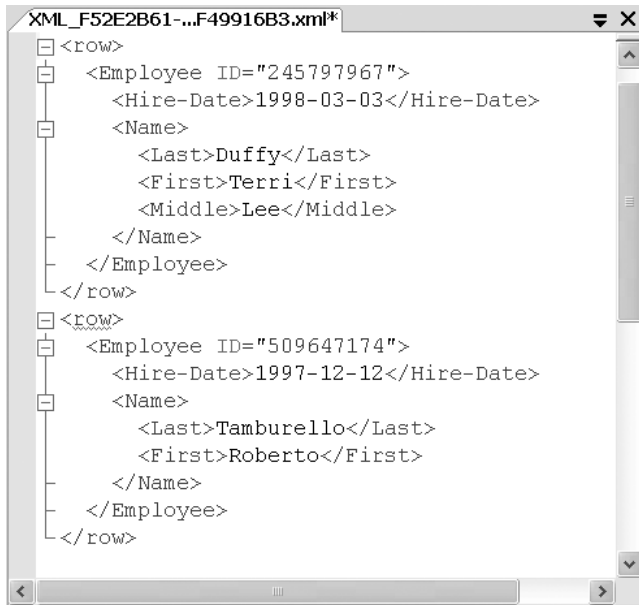


Figure 2-3. Result of `FOR XML PATH` without `ELEMENTS XSINIL`

`FOR XML PATH` also supports the `TYPE`, `BINARY BASE64`, and `ROOT` options. It does not support the `XMLSCHEMA` and `XMLDATA` options for inline schema generation.

RAW Mode

`FOR XML RAW` is useful for quickly generating XML from your source relational data. RAW mode is a “no muss, no fuss” method for generating XML data quickly and easily. Though it is simple, `FOR XML RAW` requires you to give up a good deal of flexibility and control over your XML result structure. In its simplest form, RAW mode represents each row of your relational data as a single XML node with attributes representing each column. Listing 2-3 shows a simple AdventureWorks department listing in XML format. The result of this simple query is shown in Figure 2-4.

Listing 2-3. Sample `FOR XML RAW` Query

```
SELECT d.DepartmentID,
       d.Name,
       d.GroupName
FROM HumanResources.Department d
WHERE d.DepartmentID IN ( 7, 8 )
FOR XML RAW ( 'MyNode' );
```

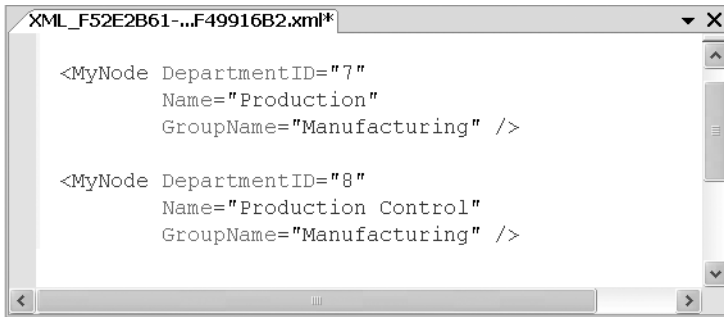


Figure 2-4. Result of sample FOR XML RAW query

FOR XML RAW assigns a default name of row to each row. I changed the row element name to MyNode in the sample by specifying the optional element name in parentheses after the RAW keyword.

Tip Some XML results shown in this book have been formatted for easy reading. The content of the XML, however, has not been altered.

One thing you might notice immediately about this result is that it is not well-formed—it does not have a single top-level element. You can change this behavior by specifying the optional ROOT keyword of the FOR XML RAW clause. Listing 2-4 revises the previous query by adding a root node named TheRootNode to form an XML document instead of a fragment. The result of the simple example in Listing 2-4 is the well-formed XML shown in Figure 2-5.

Listing 2-4. FOR XML RAW Query with ROOT Option

```
SELECT d.DepartmentID,
       d.Name,
       d.GroupName
FROM HumanResources.Department d
WHERE d.DepartmentID IN ( 7, 8 )
      FOR XML RAW ('MyNode'),
      ROOT('TheRootNode');
```

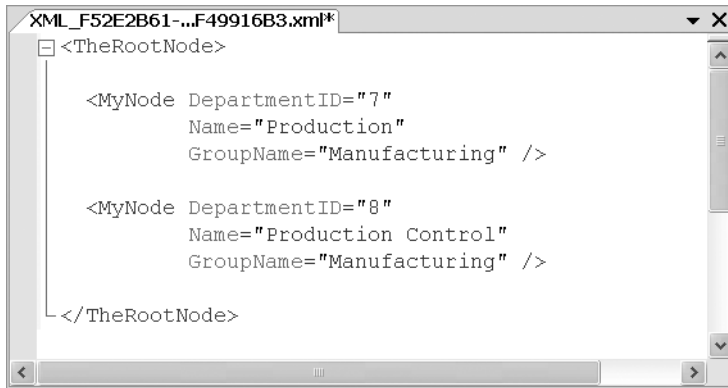


Figure 2-5. Result of FOR XML RAW query with ROOT option

The default output format, with one element per row and each column represented as an attribute, is known as *attribute-centric* XML. You can use the `ELEMENTS` option to generate your result in element-centric format, using subelements to represent columns in the output. Listing 2-5 modifies Listing 2-4 to produce an element-centric format. The element-centric result of this query is shown in Figure 2-6.

Listing 2-5. FOR XML RAW with ELEMENTS Option

```
SELECT d.DepartmentID,  
       d.Name,  
       d.GroupName  
FROM HumanResources.Department d  
WHERE d.DepartmentID IN ( 7, 8 )  
      FOR XML RAW ('MyNode'),  
      ROOT('TheRootNode'),  
      ELEMENTS;
```

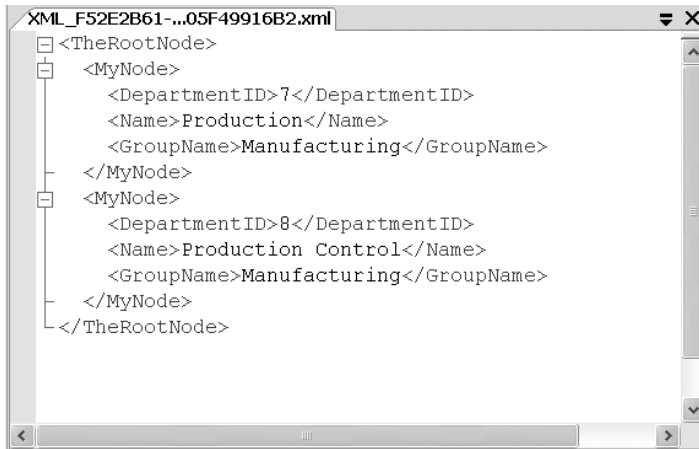


Figure 2-6. *FOR XML RAW element-centric result*

FOR XML RAW also supports inline XML schema generation with the XMLSCHEMA option. Listing 2-6 demonstrates how to add the XMLSCHEMA option to the example query in Listing 2-5, resulting in the XML with an inline XML schema definition shown in Figure 2-7.

Listing 2-6. *FOR XML RAW with XMLSCHEMA Option*

```
SELECT d.DepartmentID,
       d.Name,
       d.GroupName
FROM HumanResources.Department d
WHERE d.DepartmentID IN ( 7, 8 )
      FOR XML RAW ('MyNode'),
      ROOT('TheRootNode'),
      ELEMENTS,
      XMLSCHEMA;
```




Figure 2-7. FOR XML RAW result with inline XML schema

Note XML schema definitions allow you to constrain the structure and content of your XML data. I will cover SQL Server support for XML schema definitions in depth in Chapter 4.

The equivalent option for generating an XDR schema is shown in Listing 2-7. Note that this option should not be used in new development work; I give this example because you may have to support existing code that uses the XMLDATA option. Notice that you cannot use the ROOT option or specify the row element name when the XMLDATA option is used.

Listing 2-7. *FOR XML RAW Query with XMLDATA Option*

```
SELECT d.DepartmentID,
       d.Name,
       d.GroupName
FROM HumanResources.Department d
WHERE d.DepartmentID IN ( 7, 8 )
FOR XML RAW,
ELEMENTS,
XMLDATA;
```

When querying BINARY or VARBINARY columns with FOR XML RAW, you have to use the BINARY BASE64 option. This option translates your binary data to a Base64 representation in your XML result. Listing 2-8 is a sample query using the BINARY BASE64 option with FOR XML RAW.

Listing 2-8. *FOR XML RAW with BINARY BASE64 Option*

```
SELECT ProductPhotoID,
       ThumbnailPhoto,
       ThumbnailPhotoFileName
FROM Production.ProductPhoto
WHERE ProductPhotoID = 1
FOR XML RAW,
ELEMENTS,
BINARY BASE64;
```

Figure 2-8 shows a partial result of using the BINARY BASE64 option. The ThumbnailPhoto element in the XML result contains the Base64-encoded binary photo from the Production.ProductPhoto table.

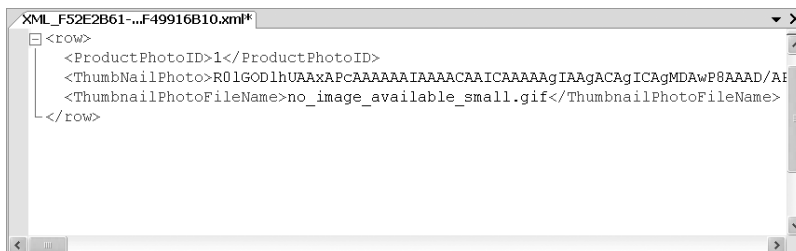


Figure 2-8. *FOR XML RAW result with BINARY BASE64 option*

FOR XML RAW mode is useful for ad hoc relational-to-XML querying, but I highly recommend PATH mode's explicit structuring for production systems that need FOR XML functionality.

AUTO Mode

The FOR XML AUTO clause, like the FOR XML RAW clause, is simple and easy to use. As with FOR XML RAW, AUTO mode retrieves relational data and automatically generates element and attribute names. The FOR XML AUTO clause attempts to make smarter choices about element and attribute names by naming nodes after the tables involved. Listing 2-9 demonstrates a simple single table FOR XML AUTO query that retrieves some addresses from the AdventureWorks Person.Address table, resulting in the XML shown in Figure 2-9.

Listing 2-9. *Single Table FOR XML AUTO Query*

```
SELECT AddressID,  
       AddressLine1,  
       AddressLine2,  
       City  
FROM Person.Address  
WHERE AddressID IN ( 532, 533 )  
       FOR XML AUTO;
```

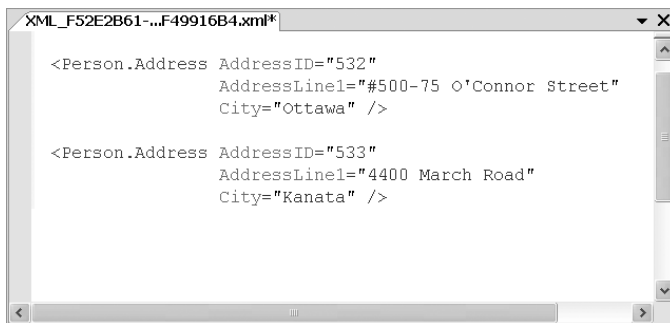


Figure 2-9. *Result of single table FOR XML AUTO query*

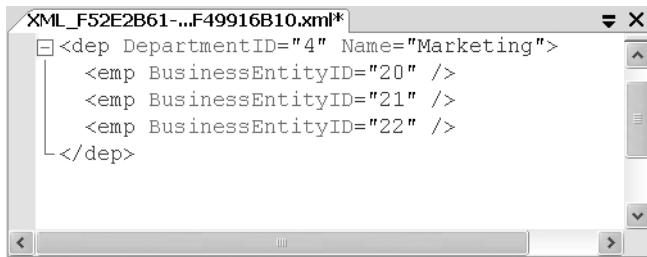
Notice that the nodes are named for the source table, in this instance Person.Address. The node naming rules for a multi-table join are slightly different. AUTO mode uses the source table names of the columns in the SELECT list to name, and to hierarchically structure, the nodes in the resulting XML. Listing 2-10 demonstrates FOR XML AUTO on an INNER JOIN between two tables. The resulting XML of this query automatically nests the emp elements inside the dep elements, as shown in Figure 2-10.

Listing 2-10. *Joined Tables FOR XML AUTO Query*

```

SELECT dep.DepartmentID,
       dep.Name,
       emp.BusinessEntityID
FROM HumanResources.Department dep
INNER JOIN HumanResources.EmployeeDepartmentHistory emp
  ON dep.DepartmentID = emp.DepartmentID
WHERE emp.BusinessEntityID BETWEEN 20 AND 22
      FOR XML AUTO;

```

**Figure 2-10.** *Result of FOR XML AUTO with INNER JOIN*

You may notice that AUTO mode, like RAW mode, is not guaranteed to generate well-formed XML by default. There is no root element generated by the sample in Listing 2-9. If you want to guarantee well-formed XML, AUTO mode can accept the ROOT option to generate well-formed XML. In fact, except for the *element_name* option, FOR XML AUTO accepts all of the options that FOR XML RAW supports.

Like RAW mode, the default AUTO mode options generate attribute-centric results. AUTO mode lets you override this default behavior with the ELEMENTS option, as shown in Listing 2-11. The resulting XML uses elements to represent tables with properly nested columns as elements also, as shown in Figure 2-11.

Listing 2-11. *FOR XML AUTO Query with ELEMENTS Option*

```

SELECT dep.DepartmentID,
       dep.Name,
       emp.BusinessEntityID
FROM HumanResources.Department dep
INNER JOIN HumanResources.EmployeeDepartmentHistory emp
  ON dep.DepartmentID = emp.DepartmentID
WHERE emp.BusinessEntityID BETWEEN 20 AND 22
      FOR XML AUTO,
      ELEMENTS;

```

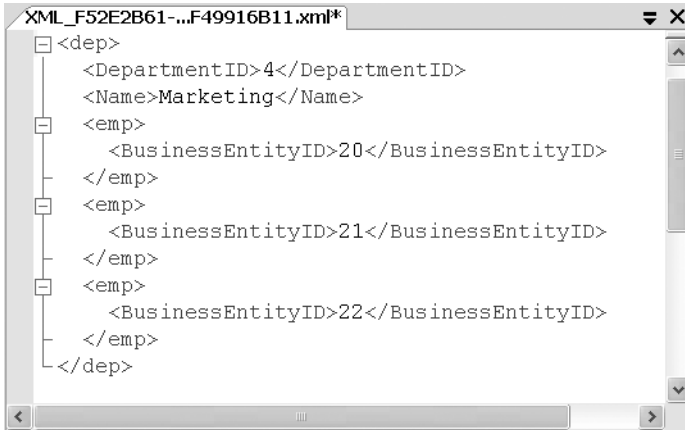


Figure 2-11. *FOR XML AUTO element-centric result*

As with `FOR XML RAW`, `AUTO` mode also supports inline XML schema generation with the `XMLSCHEMA` option and Base64 encoding of binary data with the `BINARY BASE64` option.

Caution Both `RAW` and `AUTO` modes support the deprecated `XMLDATA` option, which generates an inline XML XDR schema. Don't use this option, as it will be removed in a future version of SQL Server. Instead use the `XMLSCHEMA` option to generate inline XML schema definitions.

`FOR XML AUTO` is useful for ad hoc queries. It can also be useful in production settings, so long as the relational table structure and column names are very stable, or if the structure of your XML data doesn't have to be well-known. Unfortunately, there's often no guarantee that table structures in a database will remain the same over time. If you rely on the structure of your XML content to be stable, I highly recommend using `PATH` mode to ensure that your XML structure is well-known, even if the underlying relational structures are changed.

FOR XML AUTO LOGGING

I mentioned in this section that you might find a use for `FOR XML AUTO` mode in situations where the structure of your XML data does not need to be well-known. Right about now you might be asking yourself one critical question: “What is this guy talking about?” To answer that question, you’ll look at an example where `FOR XML AUTO` works well, logging DML actions against a table through what I call a “dynamic logging trigger.” This is a trigger that can be installed as is on most database tables. It will log every insert, update, or delete action against the table. It retrieves information about itself dynamically and uses `FOR XML AUTO` to log all data that has been affected by the DML action.

To start this example, I’ll first create a logging table that will contain the information for every DML action performed against any table with the dynamic logging trigger installed on it. The code to create this table is as follows:

```
CREATE TABLE dbo.LogDML
(
    Id            int NOT NULL IDENTITY(1, 1) PRIMARY KEY,
    SchemaName    varchar(100) NOT NULL,
    TableName     varchar(100) NOT NULL,
    TriggerName   varchar(100) NOT NULL,
    LogTime       datetime NOT NULL,
    UserName      varchar(100) NOT NULL,
    SPID          int NOT NULL,
    Delta         xml
);
GO
```

Most of the table columns are self-describing. `SchemaName`, for instance, is the name of the current schema, `TableName` is the name of the current table being logged, and so on. The `Delta` column is an `xml` data type column that will contain XML-formatted snapshots of the before and after states of the data affected by the DML action being logged. The following listing shows a simple version of the dynamic logging trigger:

```
CREATE TRIGGER HumanResources.LogDML_Shift
ON HumanResources.Shift
FOR UPDATE, INSERT, DELETE
AS
BEGIN
    DECLARE @Delta xml;

    SELECT @Delta = COALESCE((
        SELECT *
        FROM deleted
        FOR XML AUTO), '<deleted/>') +
        COALESCE((
            SELECT *
            FROM inserted
            FOR XML AUTO), '<inserted/>')

    DECLARE @TriggerID int;
```

```

DECLARE @TableID int;
DECLARE @SchemaID int;
DECLARE @SchemaName varchar(100);

SET @TriggerID = @@PROCID;

SELECT @TableID = parent_id
FROM sys.triggers
WHERE object_id = @TriggerID;

SELECT @SchemaID = t.schema_id,
       @SchemaName = s.name
FROM sys.tables t
INNER JOIN sys.schemas s
ON t.schema_id = s.schema_id
WHERE t.object_id = @TableID;

INSERT INTO dbo.LogDML (SchemaName, TableName, TriggerName, LogTime,➤
UserName, SPID, Delta)
SELECT @SchemaName, OBJECT_NAME(@TableID),➤
OBJECT_NAME(@TriggerID),
GETDATE(), USER_NAME(), @@SPID, @Delta;
END
GO

```

The sample trigger is created on the `HumanResources.Shift` table, but this exact same trigger could be created on just about any table with no changes to the body of the trigger. The trigger grabs all relevant information, like the trigger's name, the name of the table it's attached to, and the current schema name from SQL Server's catalog views. An in-depth discussion of catalog views is beyond the scope of this book, but they are something every developer should become familiar with—they contain a wealth of information about your database and server.

The `FOR XML AUTO` clause is used to generate an XML-formatted before-and-after snapshot of the data affected by the DML action, based on the structure and content of the deleted and inserted virtual tables. Because I'm using the `FOR XML AUTO` clause, I will always get a complete snapshot of the underlying data even if the underlying table structure changes. In this instance, I don't care if the structure of the resulting XML changes, as long as it accurately reflects the data being affected by the DML action. The following DML statements demonstrate the dynamic logging trigger's functionality:

```

INSERT INTO HumanResources.Shift (Name, StartTime, EndTime)
VALUES ('Noon', '12:00:00', '20:00:00');

UPDATE HumanResources.Shift
SET Name = 'Afternoon'
WHERE Name = 'Noon';

DELETE FROM HumanResources.Shift
WHERE ShiftId > 3;

```

The dynamic logging trigger logs each of these DML actions, including the FOR XML AUTO snapshot. Following is the XML-formatted snapshot of the before-and-after data for the UPDATE statement:

```
<deleted ShiftID = "4"
    Name = "Noon"
    StartTime = "12:00:00"
    EndTime = "20:00:00"
    ModifiedDate = "2008-01-28T22:23:25.437" />
<inserted ShiftID = "4"
    Name = "Afternoon"
    StartTime = "12:00:00"
    EndTime = "20:00:00"
    ModifiedDate = "2008-01-28T22:23:25.437" />
```

This is just one example of when FOR XML AUTO can come in handy in production environments. This example works because it's more important to capture an accurate and complete picture of the state of the data than to generate data in a specific XML structure that is well-known to external applications. If it's more important that you expose data in a well-known format to external users and applications, then I highly recommend you use FOR XML PATH instead.

EXPLICIT Mode

The FOR XML EXPLICIT clause is a flexible, but complex, method of generating XML with an explicit structure. Because EXPLICIT mode is deprecated, I'm just giving an overview of its functionality here, primarily because you may need to support existing FOR XML EXPLICIT queries (or convert them to PATH mode syntax) in legacy applications. For this reason, it may be important to understand how EXPLICIT mode works.

Caution FOR XML EXPLICIT is deprecated in SQL Server 2008 and will be removed in a future version of SQL Server. Do not use FOR XML EXPLICIT for new development, and start planning now to upgrade existing code to use FOR XML PATH instead as soon as possible.

Unlike PATH mode, EXPLICIT mode requires you to specify the structure using “bang notation,” in which parts of the structure definition are separated by exclamation points. Specifically, EXPLICIT mode uses an Element!Tag!Attribute!Directive notation, which takes data returned in a “universal table format” and converts it to XML. The universal table format is a very specific format that has the following characteristics:

- The format has a tag column and a parent column, which determine the hierarchical structure of the XML.
 - The tag column is an identifier for the level of the XML element. A tag value of 1 is the root element. It must have an associated parent of NULL.
 - The parent column points back to the parent tag value for an element.

- Each column of the query must be named using the Element!Tag!Attribute!Directive notation mentioned previously.
 - The Element portion of the identifier specifies the XML element name that will be generated.
 - The Tag portion is a tag number that helps determine how the elements are nested in the XML result.
 - The Attribute portion is the name of the attribute that is created for the given Element if no xml, data, or cdata Directive is specified. If an xml, data, or cdata Directive is specified, the Attribute portion specifies the subelement name that is generated.
 - The optional Directive portion of the identifier specifies additional information about how to generate XML from the data in the column. Possible values for the Directive portion of the identifier are listed in Table 2-2.
- If the Attribute and Directive portions of an identifier are not specified, an empty Attribute portion and element directive are implied. An identifier like [Quantity!2] is equivalent to the identifier [Quantity!2!!element].

FOR XML EXPLICIT supports a wide range of Directive options, which are listed in Table 2-2.

Table 2-2. *FOR XML EXPLICIT Directive Options*

Directive	Definition
ID, IDREF, IDREFS	These directives encode the column values as intra-document links.
hide	This directive hides the node.
element	This directive generates a contained element instead of an attribute. Entities are encoded in the value, such as > for the greater-than sign (>).
elementxsnil	This directive generates an element with an xsi:nil="true" attribute for NULLs. By default, no elements are generated for NULLs.
xml	This directive generates a contained element, just like the element directive. The difference is that the xml directive does not encode entities.
cdata	This directive wraps the data with a CDATA section. The content is not entity-encoded.
xmltext	This directive wraps the column in a single element that is integrated into the document.

Listing 2-12 demonstrates the complexity involved in using the FOR XML EXPLICIT clause to generate explicitly formatted XML.

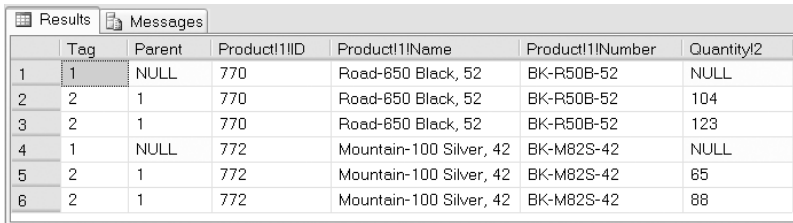
Listing 2-12. *FOR XML EXPLICIT Query Example*

```

SELECT 1 AS Tag,
      NULL AS Parent,
      p.ProductID AS [Product!1!ID],
      p.Name AS [Product!1!Name],
      p.ProductNumber AS [Product!1!Number],
      NULL AS [Quantity!2]
FROM Production.Product p
WHERE p.ProductID IN ( 770, 772 )
UNION ALL
SELECT 2 AS Tag,
      1 AS Parent,
      p.ProductID,
      p.Name,
      p.ProductNumber,
      pi.Quantity
FROM Production.ProductInventory pi
INNER JOIN Production.Product p
      ON p.ProductID = pi.ProductID
WHERE p.ProductID IN ( 770, 772 )
ORDER BY [Product!1!ID], [Product!1!Number], [Quantity!2]
FOR XML EXPLICIT;

```

This query is composed of two separate queries that are unioned together to generate a result in the universal table format. The universal table format is shown in Figure 2-12, and the final XML result generated from this universal table format with `FOR XML EXPLICIT` is shown in Figure 2-13.



	Tag	Parent	Product!1!ID	Product!1!Name	Product!1!Number	Quantity!2
1	1	NULL	770	Road-650 Black, 52	BK-R50B-52	NULL
2	2	1	770	Road-650 Black, 52	BK-R50B-52	104
3	2	1	770	Road-650 Black, 52	BK-R50B-52	123
4	1	NULL	772	Mountain-100 Silver, 42	BK-M82S-42	NULL
5	2	1	772	Mountain-100 Silver, 42	BK-M82S-42	65
6	2	1	772	Mountain-100 Silver, 42	BK-M82S-42	88

Figure 2-12. *Universal table format result set*

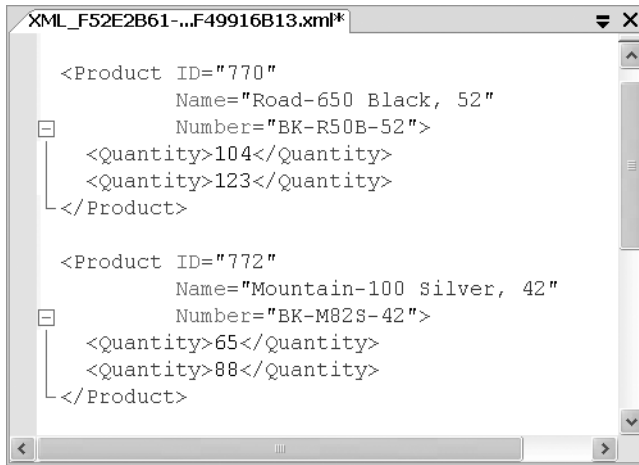


Figure 2-13. *FOR XML EXPLICIT example query result*

The FOR XML EXPLICIT clause also supports the BINARY BASE64, TYPE, ROOT, and XMLDATA options, which operate the same as they do for the other FOR XML modes. I won't go into further detail on FOR XML EXPLICIT operation because it is deprecated and will be removed from SQL Server in a future release. However, it is important to understand EXPLICIT mode if you must support legacy applications that use it. If you do have legacy code that uses the FOR XML EXPLICIT clause, you should begin planning to change it to use the simplified FOR XML PATH clause. If you write new code, avoid EXPLICIT mode and use PATH mode instead.

Using XPath Node Tests

Node tests in XPath are used to test whether nodes meet specific criteria. The FOR XML PATH, XPath-style syntax supports a subset of node test names that act as functions. These node tests are used to add specific types of nodes to your XML result. When node tests are used as part of a FOR XML PATH column name or alias, they always occur at the end of the XPath expression. Because XPath is case sensitive, node tests are also case sensitive. SQL Server supports several node tests, which are listed in Table 2-3 with a summary of FOR XML PATH XPath-style naming conventions.

Table 2-3. *FOR XML PATH Node Tests and Naming*

Column Name	Result
<code>text()</code>	The <code>text()</code> node test adds the string value of the column to the XML as a text node.
<code>comment()</code>	The <code>comment()</code> node test adds the string value of the column to the XML as a comment node.
<code>node()</code>	The <code>node()</code> node test adds the string value of the column inline under the row element.
<code>element/node()</code>	The <code>element/node()</code> node test format adds the string value of the column inline under the specified element.
<code>data()</code>	The <code>data()</code> node test adds the string value of the column as an atomic value. Spaces are inserted between atomic values in the resulting XML.
<code>processing-instruction(name)</code>	The <code>processing-instruction(name)</code> node test adds the string value of the column to the XML as a processing instruction named <i>name</i> .
<code>*</code>	The <code>*</code> wildcard character adds the string value of the column inline under the row element. This is the same as <code>node()</code> .
<code>element/*</code>	The <code>*</code> wildcard character after an element name or element name path will add the string value of the column inline under the specified element. This is the same as the <code>element/node()</code> format.
<code>@name</code>	The <code>@name</code> syntax adds the string value of the column as an attribute of the row element.
<code>name</code>	The <code>name</code> syntax adds the string value of the column as a subelement of the row element.
<code>element/name</code>	The <code>element/name</code> syntax adds the string value of the column to the XML as a subelement in the specified element hierarchy, under the row element.
<code>element/@name</code>	The <code>element/@name</code> syntax adds the string value of the column to the XML as an attribute of the last element in the specified hierarchy, under the row element.

Listing 2-13 demonstrates using a simple `text()` node test to combine string values in the XML result.

Listing 2-13. *text() Node Test Example*

```
SELECT d.DepartmentID AS "Department/@ID",
       d.Name AS "Department/Name",
       ',' AS "Department/Name/text()",
       d.GroupName AS "Department/Name/text()"
FROM HumanResources.Department d
WHERE d.DepartmentID IN (1, 2)
FOR XML PATH;
```

The `text()` node test takes the string value of a given column and inserts it directly in the XML result as an XML text node. The `text()` node is most useful for combining multiple elements into a single node in the XML result. The two `text()` node tests in the example's XPath-style column aliases are used to combine the department name and the group name, separated by a comma, within a single XML node for each row. Figure 2-14 shows the result of the example.

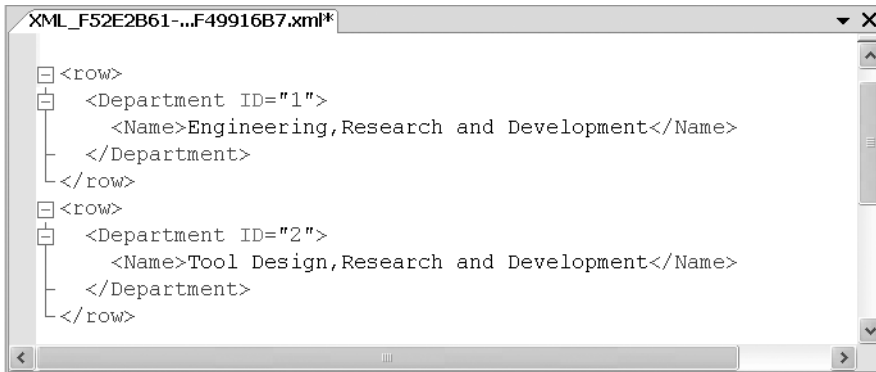


Figure 2-14. *text() node test example result*

The `comment()` node test inserts the value of a given column as an XML comment node. This is useful when you want to use values stored in a table to be used directly as documentation in your XML. Listing 2-14 shows how to use the `comment()` node test.

Listing 2-14. *comment() Node Test Example*

```
SELECT p.ModifiedDate AS "comment()",
       p.AddressID AS "Address/@ID",
       p.AddressLine1 AS "Address/Street",
       p.City AS "Address/City",
       sp.StateProvinceCode AS "Address/State",
       sp.Name AS "Address/State/comment()",
       p.PostalCode AS "Address/Postal-Code"
FROM Person.Address p
INNER JOIN Person.StateProvince sp
    ON p.StateProvinceID = sp.StateProvinceID
WHERE p.AddressID IN (25, 178)
    FOR XML PATH;
```

In this example, the `comment()` node test adds an XML comment node with the date an address was modified and another comment indicating the full name of the state or province in the address. Figure 2-15 shows the results of this example query.

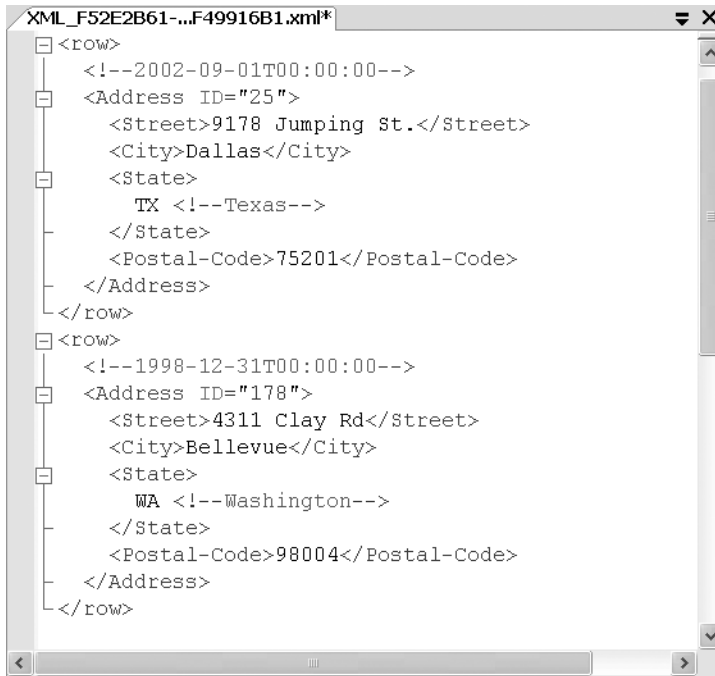


Figure 2-15. Result of `comment()` node test example

The `node()` node test inserts the value of a column inline into the XML result. If the column is an `xml` data type column, it is inserted directly into the result as inline XML. For other data types the value of the column is inserted inline without XML tags. Listing 2-15 demonstrates the `node()` node test in a query.

Listing 2-15. *node() Node Test Example*

```

SELECT pm.ProductModelID AS "Model/@ID",
       pm.Name AS "Model/Name",
       pm.rowguid AS "Model/Name/GUID/node()",
       pm.Instructions AS "Model/Instructions/node()"
FROM Production.ProductModel pm
WHERE pm.ProductModelID = 66
      FOR XML PATH;

```

Listing 2-15 uses the `node()` node test to insert both a GUID value and an `xml` column instance of the source `Instructions` node (and all of its child nodes) into the result set. The results are shown in Figure 2-16.

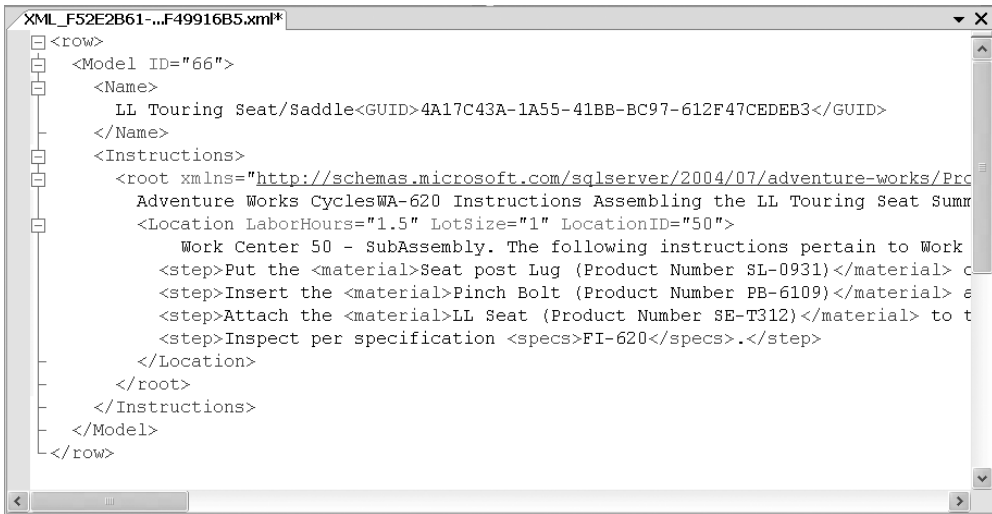


Figure 2-16. *node() node test example result*

Tip The asterisk (*) wildcard character can be used in an XPath expression in place of the node() node test. They are functionally equivalent.

The data() node test inserts the atomic values of the column into the result. If multiple atomic values are generated in a single row (through a subquery), data() creates a space-separated list of items. This is useful for generating lists of items within a single element or attribute. Listing 2-16 uses the data() node test to create a space-separated list of employee pay-rate change dates within a single element.

Listing 2-16. *data() Node Test Query Example*

```
SELECT e.NationalIDNumber AS "Employee/@ID",
       p.FirstName AS "Employee/Name/First",
       p.LastName AS "Employee/Name/Last",
       (
         SELECT eph.RateChangeDate AS "data()"
         FROM HumanResources.EmployeePayHistory eph
         WHERE eph.BusinessEntityID = e.BusinessEntityID
         FOR XML PATH ('')
       ) AS "Employee/Rate-Change-Dates"
FROM HumanResources.Employee e
INNER JOIN Person.Person p
  ON e.BusinessEntityID = p.BusinessEntityID
WHERE e.BusinessEntityID IN ( 4, 16 )
FOR XML PATH;
```

This example is our first example (so far) of a nested FOR XML subquery. The FOR XML subquery uses the data() node test to generate a space-delimited list of pay-rate change dates for each employee. This list is inserted into the XML result as a single xml instance of each Employee element, as shown in Figure 2-17.



Figure 2-17. *data() node test example result*

The processing-instruction(*name*) node test generates processing instructions in the XML result. Processing instructions are machine-readable instructions within your XML—items that generally don't fit with the content of the actual XML data but provide additional information about how the data should be processed. Listing 2-17 shows an example FOR XML PATH query that generates XML with processing instructions.

Listing 2-17. *processing-instruction(name) Node Test Query Example*

```

SELECT p.ProductID AS "Prodcut/@ID",
       p.Name AS "Product/Name",
       p.ProductNumber AS "Product/Number",
       p.Color AS "Product/processing-instruction(color-scheme)",
       CASE pp.LargePhotoFileName
         WHEN 'no_image_available_large.gif' THEN 'No-Image'
         ELSE 'Show-Image'
       
```



```

        END AS "Product/processing-instruction(display-style)",
        pp.LargePhotoFileName AS "Product/Photo"
FROM Production.Product p
INNER JOIN Production.ProductProductPhoto ppp
    ON p.ProductID = ppp.ProductID
INNER JOIN Production.ProductPhoto pp
    ON ppp.ProductPhotoID = pp.ProductPhotoID
WHERE p.ProductID IN ( 716, 717 )
FOR XML PATH;

```

The result includes processing instructions that can be used by an application to determine styles, set color schemes for display, recommend additional color-coordinated products, or for a variety of other application-specific purposes. Figure 2-18 shows the result of including processing instructions in your XML.



Figure 2-18. *processing-instruction(name) node test example result*

Support for XPath node tests in FOR XML PATH queries makes this PATH mode more powerful and flexible than the other FOR XML modes, as well as much easier to use than PATH mode's predecessor, the FOR XML EXPLICIT clause.

MULTIPURPOSE MOVES

Just before beating me soundly, my favorite chess grandmaster gave me the following advice: “Always look for the multipurpose move.” SQL Server’s XML support offers you many opportunities to look for the “multipurpose move,” allowing you to use XML in ways you might not expect. The FOR XML clause has been optimized over the course of nearly a decade of research, development, and solid real-world testing. You can take advantage of that hardcore efficiency in several areas, like string concatenation. Normally concatenating strings in T-SQL requires an inefficient WHILE loop or cursor, but you use FOR XML to efficiently accomplish the task. The following code sample uses FOR XML PATH to create a simple pipe-delimited list of all the AdventureWorks product names:

```
SELECT '|' AS "text()",
       Name AS "text()"
FROM Production.Product
FOR XML PATH('');
```

The partial result looks like the following:

```
|Adjustable Race|Bearing Ball|BB Ball Bearing|Headset Ball Bearings|Blade . . .
```

Astute readers will notice that there is an extra leading pipe character, but this is easily taken care of with SQL Server’s STUFF function, which I’ll demonstrate in the next code sample. FOR XML PATH can be used to generate more complex string concatenations. The following sample code creates a CSV-formatted string, complete with double quotes around the individual items, and removes the extra comma with the STUFF function. It also uses the SELECT statement’s ORDER BY clause to ensure that the results are sorted in alphabetical order by product name, something that was not guaranteed in the previous example.

```
SELECT STUFF(
(
  SELECT ',' AS "text()",
         '"' AS "text()",
         Name AS "text()",
         '"' AS "text()"
  FROM Production.Product
  ORDER BY Name
  FOR XML PATH('')
), 1, 1, '');
```

Notice that the ORDER BY clause is used in a subquery, a construct that’s only allowed when the FOR XML clause or the TOP keyword is used. The partial result looks like this:

```
"Adjustable Race","All-Purpose Bike Stand","AWC Logo Cap","BB Ball Bearing". . .
```

If you look at the query plan for these code samples you’ll see that SQL Server produces extremely efficient plans that are highly optimized via the FOR XML (UDX) Extended Operators. While working with the new SQL Server functionality, keep in mind the idea of multipurpose moves, and you may find interesting applications for both old and new features in corners you would not normally expect.

Adding Namespaces to FOR XML

XML namespaces provide a W3C-defined method to avoid XML node naming conflicts and to add flexibility to XML documents. XML namespaces associate a namespace prefix to a namespace Uniform Resource Identifier (URI). Namespace URIs are often supplied by, or required by, external applications like web services and other XML consumers. SQL Server provides the `WITH XMLNAMESPACES` clause to associate XML namespace prefixes with namespace URIs in the `FOR XML` clause. `WITH XML NAMESPACE`s can accept one or more URIs with either a namespace name or the `DEFAULT` keyword to define each namespace URI and its associated prefix.

XML NAMESPACE URI

Namespaces are defined by their URIs. A namespace URI can actually be any string value you care to use, but it is often a URL or web address. The URL doesn't have to actually point to anything that exists for the namespace to work though. URLs are used for a couple of reasons: they are highly unique, which helps prevent namespace conflicts, and they can (but don't have to) point to actual supporting documentation or XML schema definition documents. The XML namespace prefix is a "friendly-name" for your namespace, but it is expanded to the full namespace URI during processing.

During processing, XML namespace prefixes are expanded to their associated URIs. For example, if you have a namespace with a prefix of `ns1` and a URI of `http://www.microsoft.com/AdventureWorksDB/Person`, the XML processor will internally expand the prefix to the URI whenever it encounters it. For this simple example, the XML processor will internally expand the element name `ns1:Person` into a name like `{http://www.microsoft.com/AdventureWorksDB/Person}Person`. This is why it is extremely important to give your namespaces unique URIs.

You can declare multiple namespaces in a comma-separated list in the `WITH XMLNAMESPACES` declaration. You can also declare a default namespace with the `DEFAULT` clause. Once the namespace(s) are declared using `WITH XMLNAMESPACES`, they can be used to prefix column name aliases in your `FOR XML PATH` XPath expressions. Listing 2-18 demonstrates how to use `WITH XMLNAMESPACES` to generate XML with a default namespace. The result is shown in Figure 2-19.

Listing 2-18. *WITH XMLNAMESPACES* Sample Query

```
WITH XMLNAMESPACES(DEFAULT 'http://www.microsoft.com/AdventureWorksDB/Product')
SELECT p.ProductID AS "Product/@ID",
       p.Name AS "Product/Name",
       p.ProductNumber AS "Product/Number",
       p.Size AS "Product/Size/data()",
       p.SizeUnitMeasureCode AS "Product/Size/data()"
FROM Production.Product p
WHERE p.ProductID = 775
FOR XML PATH;
```

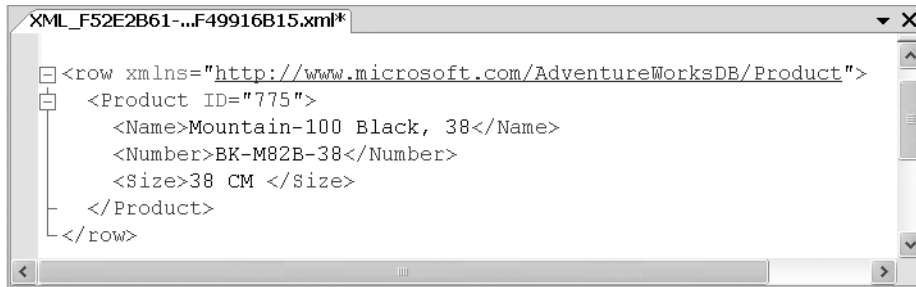


Figure 2-19. *WITH XMLNAMESPACES example with DEFAULT namespace*

As I mentioned, you can also specify explicit namespace URIs in the `WITH XMLNAMESPACES` clause. Once declared, these namespaces can be used as part of the column naming/aliasing conventions. Listing 2-19 demonstrates the use of explicit namespace URIs.

Listing 2-19. *Explicit Namespace URI Example*

```
WITH XMLNAMESPACES(
    N'http://www.microsoft.com/AdventureWorksDB/Person' AS ns1,
    N'http://www.microsoft.com/AdventureWorksDB/Email' AS ns2
)
SELECT p.BusinessEntityID AS "ns1:Person/@ID",
       p.LastName AS "ns1:Person/ns1:Name/ns1:Last",
       p.FirstName AS "ns1:Person/ns1:Name/ns1:First",
       e.EmailAddress AS "ns1:Person/ns1:Email"
FROM Person.Person p
INNER JOIN Person.EmailAddress e
    ON p.BusinessEntityID = e.BusinessEntityID
WHERE p.BusinessEntityID = 775
    FOR XML PATH;
```

The XML generated by this query is shown in Figure 2-20, in all its namespace-qualified glory. Notice that two different namespace URIs were explicitly declared and used to generate these results.



Figure 2-20. *Result of explicit namespace URI example query*

Namespaces are a simple concept, but a very important one that you will see again and again as you work your way through the more powerful XML functionality provided by SQL Server.

Creating Complex FOR XML Queries

FOR XML is designed to return relational data in a hierarchical XML format. A very common practical application of hierarchical data is the classic Bill of Materials (BOM) used by manufacturing businesses. AdventureWorks builds their own bicycles and bicycling products, and they store product BOMs in relational format in the `Production.BillOfMaterials` table. To better understand the concept of the BOM, first take a look at the hierarchical structure of a BOM for AdventureWorks' product 749, the Road-150 Red, 62 bike, as shown in Figure 2-21. The boxes with numbers in the chart represent the ID numbers of the parts that are used to create the parts above them in the hierarchy. So part #482 is used to create part #329, which is used to create part #813, and so on up the tree.

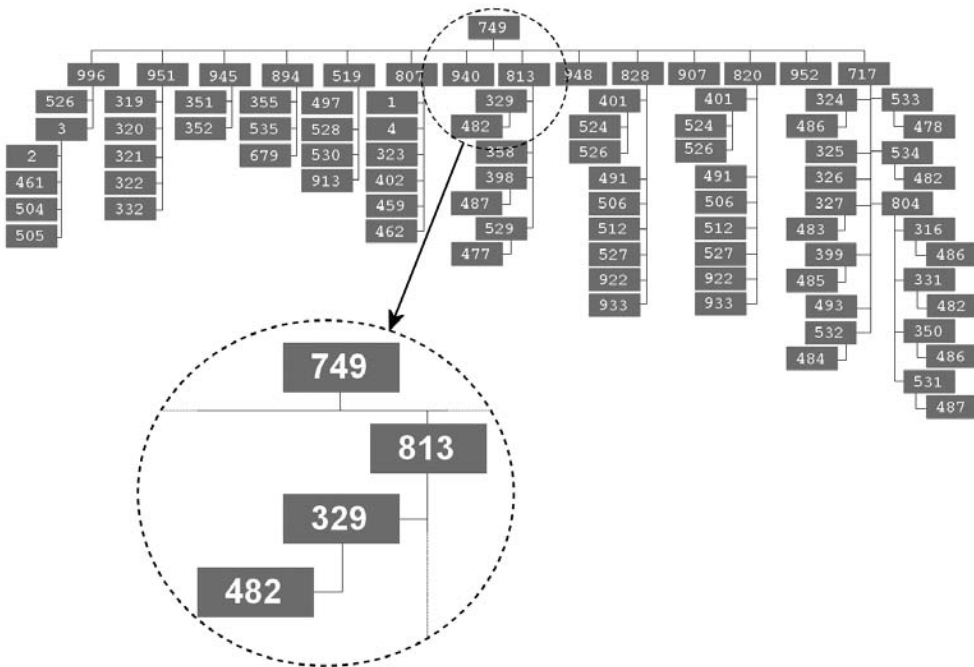


Figure 2-21. BOM for AdventureWorks product 749

Each level of the hierarchical BOM tree is composed of the components beneath it. AdventureWorks supplies a stored procedure named `dbo.uspGetBillOfMaterials` to return product BOMs as a relational result set. Listing 2-20 shows how to call this procedure to retrieve the BOM for a given product number.

Listing 2-20. *dbo.uspGetBillOfMaterials Stored Procedure Call*

```
EXEC dbo.uspGetBillOfMaterials @StartProductID = 749,
    @CheckDate = N'2007-09-02';
```

This procedure call returns the BOM for product 749. A partial listing of this bike and its subcomponents is shown in relational form in Figure 2-22.

	ProductAssembly...	Component...	ComponentDesc	TotalQuan...	StandardC...	ListPrice	BOMLe...	Recur...
1	749	519	HL Road Seat Assembly	1.00	145.87	196.92	1	0
2	749	717	HL Road Frame - Red, 62	1.00	868.6342	1431.50	1	0
3	749	807	HL Headset	1.00	55.3901	124.73	1	0
4	749	813	HL Road Handlebars	1.00	53.3999	120.27	1	0
5	749	820	HL Road Front Wheel	1.00	146.5466	330.06	1	0
6	749	828	HL Road Rear Wheel	1.00	158.5346	357.06	1	0
7	749	894	Rear Derailleur	1.00	53.9282	121.46	1	0
8	749	907	Rear Brakes	1.00	47.286	106.50	1	0
9	749	940	HL Road Pedal	1.00	35.9596	80.99	1	0
10	749	945	Front Derailleur	1.00	40.6216	91.49	1	0
11	749	948	Front Brakes	1.00	47.286	106.50	1	0
12	749	951	HL Crankset	1.00	179.8156	404.99	1	0
13	749	952	Chain	1.00	8.9866	20.24	1	0
14	749	996	HL Bottom Bracket	1.00	53.9416	121.49	1	0
15	519	497	Pinch Bolt	4.00	0.00	0.00	2	1
16	519	528	Seat Lug	1.00	0.00	0.00	2	1
17	519	530	Seat Post	1.00	0.00	0.00	2	1
18	519	913	HL Road Seat/Saddle	1.00	23.3722	52.64	2	1
19	717	324	Chain Stays	2.00	0.00	0.00	2	1

Figure 2-22. *Relational bill of materials for AdventureWorks product 749*

This BOM represents the same components of product 749 that you saw in Figure 2-21. It's common in FOR XML examples to use the AdventureWorks-supplied query (or a slight variation of it) and simply add FOR XML to generate the BOM in XML format. This simple method is shown Listing 2-21.

Listing 2-21. *Using FOR XML to Generate an XML BOM*

```
DECLARE @ComponentID AS int;
SET @ComponentID = 749;
```

```

WITH BOMCTE(ProductAssemblyID, ComponentID, Qty, BOMLevel, RecursionLevel)
AS
(
    SELECT ProductAssemblyID,
           ComponentID,
           CAST(PerAssemblyQty AS integer),
           BOMLevel,
           0
    FROM Production.BillofMaterials
    WHERE ComponentID = @ComponentID
           AND ProductAssemblyID IS NULL
           AND EndDate IS NULL

    UNION ALL

    SELECT BOM.ProductAssemblyID,
           BOM.ComponentID,
           CAST(BOMCTE.Qty * BOM.PerAssemblyQty AS integer),
           BOM.BOMLevel,
           BOMCTE.RecursionLevel + 1
    FROM BOMCTE
    JOIN Production.BillofMaterials AS BOM
      ON BOM.EndDate IS NULL
      AND BOM.ProductAssemblyID = BOMCTE.ComponentID
)
SELECT B.ProductAssemblyID,
       B.ComponentID,
       P.Name AS [ProductDescription],
       P.StandardCost,
       P.ListPrice,
       B.Qty,
       B.BOMLevel,
       B.RecursionLevel
FROM BOMCTE AS B
INNER JOIN Production.Product AS P
  ON P.ProductID = B.ComponentID
FOR XML PATH(N'item'),
      ROOT('items'),
      ELEMENTS;

```

The results reflect the components that make up item 749 in XML format, as shown in Figure 2-23.



Figure 2-23. BOM in XML format

This method of using FOR XML with a recursive Common Table Expression (CTE) has the advantage of simplicity. It can be used to retrieve BOMs of virtually any depth. There is something missing from this format, though, namely the hierarchical structure indicating the component “tree” of the product, an area in which XML is supposed to excel. There is a way to get that true hierarchical representation with FOR XML, though, and it all begins with the query shown in Listing 2-22.

Listing 2-22. Base Query of the FOR XML Hierarchical BOM

```
DECLARE @ProductID int;
SET @ProductID = 749;

SELECT a.ComponentID AS "@id",
       p.ProductNumber AS "@number",
       p.Name AS "name",
       p.Color AS "color",
       p.ListPrice AS "list-price",
       a.PerAssemblyQty AS "quantity",
       p.Size AS "size",
```



```

    p.SizeUnitMeasureCode AS "unit-of-measure"
FROM Production.BillOfMaterials a
INNER JOIN Production.Product p
    ON a.ComponentID = p.ProductID
WHERE p.ProductID = @ProductID
FOR XML PATH(N'item'),
    ROOT(N'items'),
    TYPE;

```

This simple FOR XML query returns the first level of the BOM for product 749. To get a true nested hierarchical BOM structure in XML format, you can nest the basic FOR XML query in Listing 2-22 multiple times, as shown in Listing 2-23.

Listing 2-23. *Nested FOR XML Hierarchical BOM*

```

DECLARE @ProductID int;
SET @ProductID = 749;

SELECT a.ComponentID AS "@id",
    p.ProductNumber AS "@number",
    p.Name AS "name",
    p.Color AS "color",
    p.ListPrice AS "list-price",
    a.PerAssemblyQty AS "quantity",
    p.Size AS "size",
    p.SizeUnitMeasureCode AS "unit-of-measure",
    (
        SELECT b.ComponentID AS "@id",
            p.ProductNumber AS "@number",
            p.Name AS "name",
            p.Color AS "color",
            p.ListPrice AS "list-price",
            b.PerAssemblyQty AS "quantity",
            p.Size AS "size",
            p.SizeUnitMeasureCode AS "unit-of-measure",
            (
                SELECT c.ComponentID AS "@id",
                    p.ProductNumber AS "@number",
                    p.Name AS "name",
                    p.Color AS "color",
                    p.ListPrice AS "list-price",
                    c.PerAssemblyQty AS "quantity",
                    p.Size AS "size",
                    p.SizeUnitMeasureCode AS "unit-of-measure",
                    (
                        SELECT d.ComponentID AS "@id",
                            p.ProductNumber AS "@number",
                            p.Name AS "name",

```

```

        p.Color AS "color",
        p.ListPrice AS "list-price",
        d.PerAssemblyQty AS "quantity",
        p.Size AS "size",
        p.SizeUnitMeasureCode AS "unit-of-measure",
    (
        SELECT e.ComponentID AS "@id",
               p.ProductNumber AS "@number",
               p.Name AS "name",
               p.Color AS "color",
               p.ListPrice AS "list-price",
               e.PerAssemblyQty AS "quantity",
               p.Size AS "size",
               p.SizeUnitMeasureCode AS "unit-of-measure"
        FROM Production.BillofMaterials e
        INNER JOIN Production.Product p
            ON e.ComponentID = p.ProductID
        WHERE e.ProductAssemblyID = d.ComponentID
        FOR XML PATH (N'item'), TYPE
    )
    FROM Production.BillofMaterials d
    INNER JOIN Production.Product p
        ON d.ComponentID = p.ProductID
    WHERE d.ProductAssemblyID = c.ComponentID
    FOR XML PATH (N'item'), TYPE
)
FROM Production.BillofMaterials c
INNER JOIN Production.Product p
    ON c.ComponentID = p.ProductID
WHERE c.ProductAssemblyID = b.ComponentID
FOR XML PATH (N'item'), TYPE
)
FROM Production.BillofMaterials b
INNER JOIN Production.Product p
    ON b.ComponentID = p.ProductID
WHERE b.ProductAssemblyID = a.ComponentID
FOR XML PATH(N'item'), TYPE
)
FROM Production.BillofMaterials a
INNER JOIN Production.Product p
    ON a.ComponentID = p.ProductID
WHERE p.ProductID = @ProductID
FOR XML PATH(N'item'), ROOT(N'items'), TYPE;

```

THE TYPE OPTION

You might notice the use of the TYPE option in the FOR XML subqueries in Listing 2-23. The TYPE option ensures that the data is returned as an xml data type instance. This is particularly important when using FOR XML in subqueries. If you use FOR XML in subqueries without the TYPE option, the result is returned as entitized character data—that is to say, instead of an XML result that looks like this:

```
<quantity>1.00</quantity>
```

The result will look more like this:

```
&lt;quantity&gt;1.00&lt;/quantity&gt;
```

This can cause problems, since the subquery XML data will not be recognized as real XML by SQL Server or client applications.

The hierarchical BOM result of this nested FOR XML query is shown in Figure 2-24.



Figure 2-24. Partial result of hierarchical BOM query

As you can see in the result, the components that make up product 749 are nested in a well-formed hierarchical XML document. Each subcomponent is properly nested within the parent components above it, forming a complete BOM that reflects not only the overall contents of the finished product, but also the proper structure of the components and their subcomponents. I will build on this basic BOM query in Chapter 4 during the discussion of recursive hierarchical XML schemas.

OPENXML Rowset Provider

In the old days (circa 2000), SQL Server XML support had the feel of a hodgepodge of loosely coupled functionality held at arm's length from the database engine. SQL Server 2008 provides access to procedures and functionality that give you some measure of backward-compatibility with legacy SQL Server 2000 code, but these features are somewhat archaic and kludgy to use and I would advise steering clear of them in new development. I'll discuss them in this section because you may have to support the older code that uses this legacy functionality, at least until you have an opportunity to upgrade the code to take advantage of the modern XML storage, querying, and manipulation functionality in SQL Server. In this section, I'll discuss these backward-compatibility features.

The classic SQL Server OPENXML function is a rowset provider that allows you to query XML documents like a table or view. OPENXML essentially allows you to shred XML documents, much like the `xml` data type `nodes()` method. OPENXML requires the use of system-stored procedures and is more complex to use than the `nodes()` method; it is a best practice to use the `xml` data type and its `nodes()` method to shred your XML data.

Tip For new development use the `xml` data type `nodes()` method instead of OPENXML. If you have a lot of legacy OPENXML code to support, it's a good idea to start assessing the changes to use the `xml` `query()` method.

That being said, there are many legacy applications that use OPENXML, and there seem to be a lot of developers publishing new XML shredding examples that use OPENXML exclusively. In all, OPENXML is a topic that's worth covering.

The OPENXML rowset provider accepts an *idoc* parameter, which is a handle to the internal representation of your XML document, created by using the `sp_xml_preparedocument` system-stored procedure. I'll describe `sp_xml_preparedocument` later in this section. It also accepts a *rowpattern* parameter, which is the XPath pattern to identify which XML nodes will be processed as rows. The *idoc* and *rowpattern* parameters are both required. A third parameter, *flags*, is optional and can be one of the values shown in Table 2-4.

Table 2-4. *OPENXML flags Parameter Values*

Value	Description
0	This flag specifies an attribute-centric mapping of the data. This is the default if the <i>flags</i> parameter is not passed to OPENXML.
1	This flag specifies attribute-centric mapping of the data.
2	This flag specifies element-centric mapping of the data.
3	This flag combines flags 1 and 2 to provide attribute-centric mapping first followed by element-centric mapping for all rows not yet mapped.
8	This flag indicates that the consumed data should not be copied to the overflow property @mp:xmltext.
9	This flag combines flags 1 and 8 to provide attribute-centric mapping of the data and to indicate that consumed data should not be copied to the overflow property.
10	This flag combines flags 2 and 8 to provide element-centric mapping of the data and to indicate that consumed data should not be copied to the overflow property.
11	This flag combines flags 1, 2, and 8 to provide attribute-centric mapping first followed by element-centric mapping for all rows not yet mapped. It also indicates that consumed data should not be copied to the overflow property.

The WITH clause can be used with the OPENXML function to define the table schema for the result set that is returned. The schema is defined as a set of columns, each with a column name, a SQL Server data type, and an XPath location path indicating the source of the data. Listing 2-24 demonstrates how to use OPENXML to retrieve the XML-formatted instructions for AdventureWorks product model #10 and convert the steps to relational form. The results of this query are shown in Figure 2-25.

Listing 2-24. *Sample OPENXML Query*

```
-- First, retrieve an XML document from the Production.ProductModel table
DECLARE @x xml;

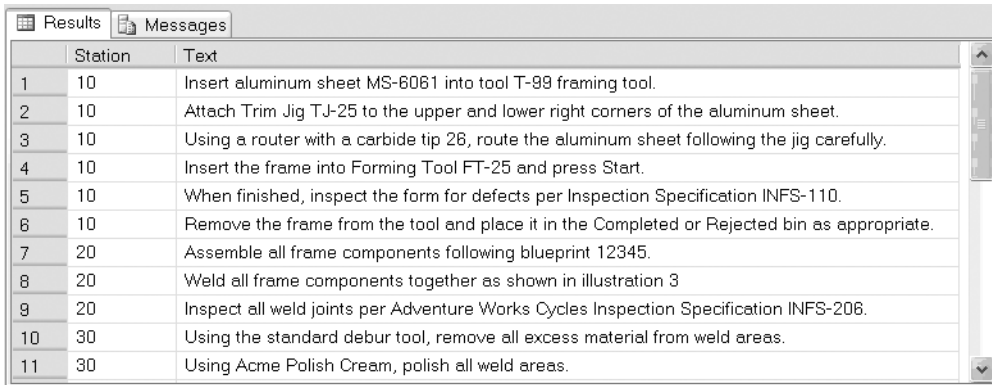
SELECT @x = pm.Instructions
FROM Production.ProductModel pm
WHERE pm.ProductModelID = 10;

-- Now create a document handle to the XML document
DECLARE @idoc int;

EXECUTE sp_xml_preparedocument @idoc OUTPUT, @x,
    '<ns xmlns:a="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManuInstructions" />';
```

```
-- The next step is to query the XML document using OPENXML
SELECT [Station], [Text]
FROM OPENXML(@idoc, '//a:Location/a:step', 3)
WITH (
    [Station] int ' ../@LocationID',
    [Text] nvarchar(max) '.'
);

-- Finally, remove the XML document from memory
EXECUTE sp_xml_removedocument @idoc;
```



	Station	Text
1	10	Insert aluminum sheet MS-6061 into tool T-99 framing tool.
2	10	Attach Trim Jig T-J-25 to the upper and lower right corners of the aluminum sheet.
3	10	Using a router with a carbide tip 26, route the aluminum sheet following the jig carefully.
4	10	Insert the frame into Forming Tool FT-25 and press Start.
5	10	When finished, inspect the form for defects per Inspection Specification INFS-110.
6	10	Remove the frame from the tool and place it in the Completed or Rejected bin as appropriate.
7	20	Assemble all frame components following blueprint 12345.
8	20	Weld all frame components together as shown in illustration 3
9	20	Inspect all weld joints per Adventure Works Cycles Inspection Specification INFS-208.
10	30	Using the standard debur tool, remove all excess material from weld areas.
11	30	Using Acme Polish Cream, polish all weld areas.

Figure 2-25. Results of sample OPENXML query

The sample query in Listing 2-24 demonstrates several key OPENXML concepts. I'll address each of these ideas in turn. The first concept presented is the use of the `sp_xml_preparedocument` stored procedure to prepare the XML document for querying in memory. This procedure accepts an XML document and returns an INT document handle. OPENXML uses the document handle returned by this procedure to query the XML document, making OPENXML completely dependent on the procedure. The `sp_xml_preparedocument` stored procedure can accept an XML document defined as a `varchar`, `nvarchar`, `xml`, or other character data type.

Another important idea here is the addition of a namespace URI as a third parameter in the call to `sp_xml_preparedocument`. This is not always required, but since the XML I am querying in the example declares a specific namespace URI, I have to declare the same namespace URI in my `sp_xml_preparedocument` procedure call. You might also notice that, though the source XML document declares a default namespace, I explicitly prefix my XPath location paths with the declared namespace prefix.

Tip If your XML document declares namespaces, you must use those same namespace URIs in your OPENXML query. Failing to do so will result in no data being returned.

Jumping ahead to the end of the listing, another closely related concept is presented. The `sp_xml_removedocument` procedure must be called once you've finished querying your XML in order to release the memory allocated to the previously prepared XML document. The `sp_xml_removedocument` procedure accepts the document handle of the XML document to release as a parameter. This XML document “prepare-and-release” strategy is a throwback to SQL Server 2000's unmanaged COM-based XML legacy.

Caution Every document created in memory by `sp_xml_preparedocument` absolutely has to be removed from memory by a call to `sp_xml_removedocument`, with no exceptions. If you fail to remove previously prepared documents from memory, you can count on memory leaks and regular reboots.

In between the calls to `sp_xml_preparedocument` and `sp_xml_removedocument` lies the heart of the routine, the OPENXML query. This query accepts a document handle, XPath location path, and the *flags* parameter. The XPath location path specified is the context node for the XPath query.

OPENXML VS. NODES()

The OPENXML rowset provider is based on the old COM model and it relies on the Microsoft XML Core Services library (MSXML) to perform XML manipulation and shredding. When SQL Server invokes MSXML it automatically reserves 1/8th of your SQL Server's memory to processing the XML document. That means if you have a 2 GB SQL Server, 250 MB is assigned to process that XML. It doesn't matter how big your XML document is—1 KB or 100 MB—the server automatically assigns 1/8th of your memory to the XML cache. The nodes method is not COM-based and is much better at dynamically allocating memory than OPENXML. This makes the score OPENXML 0, nodes() method +1.

I've performed several ad hoc tests of OPENXML performance vs. the `xml` data type nodes() method, and I've found the performance of nodes() is roughly the same as OPENXML on nonindexed XML data. As far as speed goes, both methods gain a point. OPENXML is now at +1, nodes() is at +2.

Although the speed difference appears to be negligible on nonindexed data, the nodes() method absolutely flies on a column with a primary XML index. In my simple ad hoc tests, I've recorded substantial speed increases—by as much as 50 percent. By my reckoning the nodes() method clearly beats OPENXML, 3 to 1. And the moral of the story is, get in the habit of using the nodes() method instead of OPENXML. It will help you avoid OPENXML's memory issues and, in some situations, increase performance.

I also define the schema of the result set using the `WITH` clause. In this case, the query returns two columns: the station number at which each work-step is performed, and the text description of each work-step. Both of these two column definitions include a column name, a SQL Server data type, and an XPath location path. The column location paths are defined relative to the current context node. I will discuss XPath queries and location paths in greater detail in Chapter 5.

Tip You don't have to define an explicit schema for the OPENXML query. If you don't, however, SQL Server uses a default "edge table" format, which contains a lot more information than you're likely to need and possibly splits up the data in a manner that you do not want. I recommend defining your own explicit schema for greater control of OPENXML results when possible.

Though OPENXML is useful knowledge to have when maintaining legacy code or for querying XML on legacy SQL Server 2000 systems, the functionality it provides has been superseded by the methods of the SQL Server 2008 `xml` data type. These methods provide greater control, flexibility, and ease of use via the more powerful XQuery language for querying XML data. I will discuss the `xml` data type methods in Chapter 3 and the XQuery language in Chapter 5.

OPENROWSET XML Loading

OPENROWSET is a rowset provider that pulls the contents of a specified file from the file system into SQL Server. This section will focus on using it to load XML data from secondary storage.

OPENROWSET is a rowset provider, so it will return the contents of the specified *data_file* as a relational set of rows. By specifying `SINGLE_BLOB`, `SINGLE_CLOB`, or `SINGLE_NCLOB`, the entire file is returned in a single column as a Binary Large Object (BLOB), Character Large Object (CLOB), or National Character Large Object (NCLOB), respectively. Listing 2-25 demonstrates how to use the OPENROWSET rowset provider to load a sample XML file into a SQL Server `xml` variable. Note that when using the BULK option with a `SINGLE_BLOB`, `SINGLE_CLOB`, or `SINGLE_NCLOB`, the column returned is named `BulkColumn`. The `state-list.xml` document is a sample XML file included in the file downloads. (More information about where and how to download sample files can be found in the Introduction to this book.)

Note Microsoft currently recommends that you use the `SINGLE_BLOB` option when using the OPENROWSET BULK option. Avoid `SINGLE_CLOB` and `SINGLE_NCLOB` when possible.

Listing 2-25. Loading an XML File with OPENROWSET

```
DECLARE @x xml;

SELECT @x = xCol.BulkColumn
FROM OPENROWSET (BULK 'c:\state-list.xml', SINGLE_BLOB) AS xCol;

SELECT @x;
```

A portion of the value of the `@x` variable after this code completes is shown in Figure 2-26.

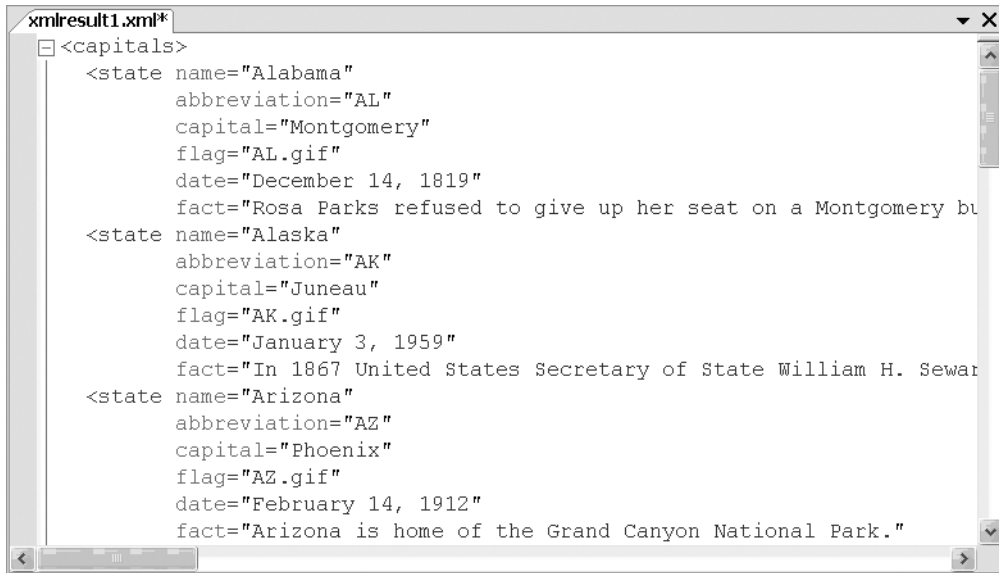


Figure 2-26. Using OPENROWSET to load XML from secondary storage

Summary

SQL Server provides several tools for converting relational data to XML, and vice versa. In this chapter we began looking at the “legacy tools,” or the tools that have been around since SQL Server 2000. Although many of these tools have been upgraded since their introduction, they are still largely based on the same syntax, with most of the same backward-compatible options intact.

A lot of the information in this chapter, such as the discussion of OPENXML and the FOR XML EXPLICIT clause, will be particularly useful to those who have to support legacy applications and those who need to upgrade old programs. The functionality of many of the legacy functions, procedures, and rowset providers has been superseded by the xml data type and its methods.

In this chapter, I discussed the FOR XML clause, including the RAW mode, AUTO mode, PATH mode, and the deprecated EXPLICIT mode. I also talked about how to use OPENXML to retrieve XML data in relational format and looked at using the OPENROWSET rowset provider to load XML data from secondary storage.

In the next chapter, you will begin looking at the details of the powerful SQL Server 2008 xml data type and its methods and capabilities.



The xml Data Type

The centerpiece of SQL Server 2008 Extensible Markup Language (XML) development is the `xml` data type. Introduced in SQL Server 2005, this Large Object (LOB) data type supports storage of both typed and untyped XML documents and XML fragments. Each XML instance can be up to 2.1 GB in size.

In this chapter, I will discuss the `xml` data type, its many uses, and its built-in methods. I'll discuss how to create `xml` data type instances, cast `xml` from and to other data types, use inline Document Type Definitions (DTDs) in your XML, and restrict your `xml` type content to complete documents or fragments. I will also introduce XML schema collections that allow you to validate your `xml` instances and the `xml` data type methods for querying and manipulating your XML data.

Creating xml Instances

Creating `xml` instances is a simple matter of assigning XML data to an `xml` variable, column, parameter, or user-defined function return value. XML instances can be explicitly cast from other types using the Transact-SQL (T-SQL) `CAST` and `CONVERT` functions; or they can be implicitly cast (in many instances) from `varchar`, `nvarchar`, or `varbinary` data types. Explicitly casting using the `CONVERT` function gives you additional options during the conversion process, like controlling white-space handling or applying an inline DTD.

Listing 3-1 shows how to cast `nvarchar` data to an `xml` instance, both implicitly and explicitly using the `CAST` function.

Listing 3-1. Casting `nvarchar` Data to `xml` Using `CAST`

```
DECLARE @imp_x xml,  
        exp_x xml,  
        source nvarchar(200);  
  
SET @source = N'<?xml version = "1.0"?>  
<message>  
  <to>SQL Server Team</to>  
  <from>Michael Coles</from>  
  <subject>Thanks</subject>  
  <content>Thanks for the new version of SQL Server</content>  
</message>';
```

```

/* Implicit conversion to xml */
SET @imp_x = @source;

/* Explicit conversion to xml */
SET @exp_x = CAST(@source AS xml);

```

The CONVERT function performs the same basic function as CAST, but CONVERT offers additional options like the use of an internal DTD and preservation of insignificant white space. An example of the CONVERT function, as it pertains to the xml data type, is shown in Listing 3-2.

Listing 3-2. *Casting nvarchar Data to xml Using CONVERT*

```

DECLARE @x xml,
        @source nvarchar(200);

SET @source = N'<?xml version = "1.0"?>
<message>
  <to>SQL Server Team</to>
  <from>Michael Coles</from>
  <subject>Thanks</subject>
  <content>Thanks for the new version of SQL Server</content>
</message>';

/* Explicit conversion to xml */
SET @x = CONVERT(xml, @source, 0);

```

The CONVERT function can accept XML instance data in character format. It can be either a varchar, nvarchar, varbinary, or other character or binary data type. The third parameter indicates the style and can be any of the values listed in Table 3-1.

Table 3-1. *CONVERT xml Style Parameter Values*

Value	Description
0	This is the default value. Style 0 uses default parsing behavior. The default parsing behavior discards insignificant white space in the XML data and does not allow the use of an internal DTD.
1	Style 1 preserves insignificant white space in the XML data and, like style 0, does not allow the use of an internal DTD.
2	Style 2 enables internal DTD use and discards insignificant white space.
3	Style 3 enables internal DTD use like style 2, but it preserves insignificant white space.

Generally, the CONVERT function default style (style 0) is acceptable, and is equivalent to using the CAST function. If your XML data contains an inline DTD you will want to use style 2. If you need to preserve insignificant white space in your XML data, however, you will want to use style 1 or 3.

Casting and Converting

SQL Server `xml` data type instances can be populated from other data types via implicit or explicit casting or conversion. The following data types can be cast to the `xml` data type:

- The `varchar` data type can be implicitly or explicitly cast to the `xml` data type. When the `varchar` data type is used, the `xml` declaration encoding is not required. If specified, it cannot be set to `utf-16` or another 16-bit encoding. It can be set to an 8-bit character encoding, such as `utf-8`, `iso-8859-1`, or `windows-1252`. The default 8-bit character encoding used is `utf-8`.
- The `nvarchar` data type can also be implicitly or explicitly cast to the `xml` data type. When `nvarchar` is used as a source, the `xml` declaration encoding cannot be set to `utf-8` or another 8-bit encoding. It can, however, be set to `utf-16`. If no encoding is specified, the default 16-bit encoding used is `utf-16`.
- The `varbinary` data type can also be implicitly or explicitly cast to the `xml` data type. If the `varbinary` data type is used, however, there are some special considerations:
 - If the `varbinary` data does not contain a byte order mark, an 8-bit encoding is assumed.
 - If the `varbinary` data contains a byte order mark, a 16-bit encoding is assumed.
 - Using 16-bit data with an 8-bit encoding specified in the `xml` declaration can result in some strange and unexpected results full of international characters. This can happen if you cast `nvarchar` data to `varbinary`, and then cast the result to `xml`, for instance.
- The `char`, `nchar`, and `binary` data types can also be implicitly or explicitly cast to the `xml` data type. However, if using the `binary` data type, the XML data contained must be the exact length specified, with no additional padding. The variable-length data types tend to be much more flexible than their fixed-length counterparts.

Note Most of the examples given in this book implicitly cast `xml` instances from `nvarchar` string constants.

WHAT IS THE BYTE ORDER MARK?

The *byte order mark*, sometimes abbreviated BOM (but not to be confused with Bill of Materials from previous chapters), is the Unicode character at code point U+FEFF. The Unicode character 0xFEFF is stored at the beginning of Unicode files. To understand the usefulness of the byte order mark, you need to look at the way different processors store and retrieve 16-bit code units. Intel-compatible processors, for instance, store 16-bit code units in reverse-ordered byte-sized chunks, in what is commonly called *little-endian* format. Motorola, and other families of processors, store 16-bit code units in *big-endian* format with the most significant byte first.

When a Unicode file is read into memory, the byte order mark is used as a guide to automatically determine how the processor stores 16-bit code units. When a file is read in on an Intel-compatible processor, the byte order mark will be read as the byte 0xFF followed by the byte 0xFE. On other processors that use the big-endian format, the byte order will be 0xFE followed by 0xFF.

As mentioned in the “DTDs” section of this chapter, XML data with an inline DTD must be converted to an `xml` instance by using the `CONVERT` function with a style setting of 2 or 3.

SQL Server `xml` data type instances can be explicitly cast or converted to an `nvarchar`, a `varchar`, or a `varbinary` representation. Implicit casting from the `xml` data type is not allowed, however. You must use the `CAST` or `CONVERT` function to perform the conversion. The `CAST` operator is useful for converting character or binary data to an `xml` instance when the default conversion settings are adequate. If you are using DTDs or need to preserve white space, then you need to use the `CONVERT` function.

As mentioned in Chapter 1, the data stored for your `xml` instances is not an exact character-for-character copy of the string representation of your XML data. SQL Server converts your data to an efficient 16-bit internal representation based on the XQuery/XPath Data Model (XDM) and performs other tasks, like expanding CDATA sections and entitizing special characters as required. Listing 3-3 shows a simple example of one of these changes.

Listing 3-3. *xml Internal Representation vs. String Representation*

```
DECLARE @x xml;

SET @x = N'<?xml version = "1.0"?>
<partners>
<![CDATA[
    Dewey, Cheatham, & Howe
]]>
</partners>';

SELECT @x;
```

The result is shown in Figure 3-1. Notice how SQL Server expands the CDATA section and properly entitizes the ampersand character (&) to & internally.

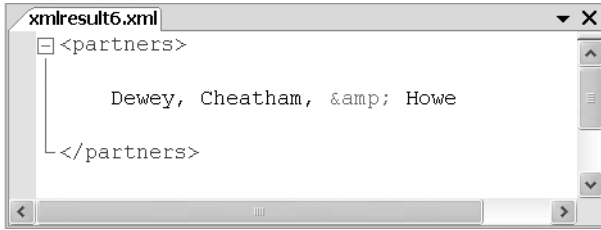


Figure 3-1. *Difference in Internal xml representation and string representation*

This is important to understand if you are planning to store XML data in your database for auditing or other purposes that require an exact character-for-character copy of the data. The XDM allows vendor products to store internal representations of typed XML data that are not character-for-character copies, so long as the meaning of the data is not changed. For instance, if your XML data contained the value 1.000, SQL Server might store it simply as the integer value 1. The numeric values 1 and 1.0000 are equivalent, so this is a perfectly valid representation. However, for auditing purposes where the exact XML is required, this might not be sufficient. I will expand on this issue further in Chapter 4 when I discuss the XDM.

Cross-Platform Tip The ISO SQL/XML Standard defines the XMLCAST, XMLSERIALIZE, and XMLPARSE functions as the standard mechanisms for casting XML data to and from other types. Syntactically, the XMLCAST function is defined to reduce to a simple CAST function call. The reason given for the extra XMLCAST keyword is to help programmers remember the restrictions involved when casting XML.

Using xml Parameters and Return Types

As shown in previous examples, the xml data type can be used to declare local SQL Server variables or table columns. It can also be used as a parameter type for user-defined functions and stored procedures or as a function return type. The syntax to declare an xml parameter or user-defined function return type is the same as it is for declaring an xml variable or column.

When the xml data type is used as a parameter for a SQL Common Language Runtime (SQLCLR) user-defined function or procedure, it maps to the Microsoft .NET Framework System.Data.SqlTypes.SqlXml data type. I'll discuss this in more detail in Chapter 8.

Creating Well-Formed and Valid XML

XML data can be well-formed documents, either typed or untyped, or XML fragments. SQL Server defines two facets for the xml data type—DOCUMENT for well-formed XML documents or CONTENT for XML fragments. Bear in mind that, even though an XML fragment does not need to have a single top-level root node, it must still follow the other rules for well-formed XML. When xml columns, variables, and parameters are declared without a facet, they default to CONTENT.

Note In XML-speak, a *facet* is a constraint or restriction on XML content. The SQL Server `xml` data type supports the two facets mentioned in this section to constrain XML data to be well-formed or fragments. Facets are also used by the XDM, which I will discuss in Chapter 4.

Untyped XML is simply XML that is not associated with an XML schema collection. Typed XML data can be stored in a column by declaring the data type of the column as `xml` and explicitly associating it with an XML schema collection. Listing 3-4 is the `CREATE TABLE` statement for the AdventureWorks sample database `Production.Illustration` table, which has an untyped `xml` column named `Diagram`.

Listing 3-4. *Production.Illustration Table with Untyped xml Column*

```
CREATE TABLE Production.Illustration (  
    IllustrationID int IDENTITY(1,1) NOT NULL PRIMARY KEY,  
    Diagram xml,  
    ModifiedDate datetime NOT NULL DEFAULT (GETDATE())  
);
```

As you can see, the declaration for the `Diagram` column `Production.Illustration` table indicates that there will be no checks made for XML data well-formedness or validity. The `Production.ProductModel` table of the AdventureWorks database, on the other hand, has two columns named `CatalogDescription` and `Instructions` that are both associated with XML schema collections. These associations create typed `xml` columns. Listing 3-5 shows the `CREATE TABLE` statement for the `Production.ProductModel` table.

Cross-Platform Tip The SQL Server `xml` data type `CONTENT` and `DOCUMENT` facets are equivalent to the ISO SQL/XML Standard `XML (ANY DOCUMENT)` and `XML (ANY CONTENT)` variations, respectively.

XML Schema Collections

While the `Diagram` column of the `Production.Illustration` table is declared as untyped `xml` with the default `CONTENT` facet, the `Production.ProductModel` table has two `xml` columns named `CatalogDescription` and `Instructions` that are associated with XML schema collections. These associations create typed `xml` columns. Listing 3-5 shows the `CREATE TABLE` statement for the `Production.ProductModel` table.

Listing 3-5. *Production.ProductModel Table with Typed xml Columns*

```
CREATE TABLE Production.ProductModel(  
    ProductModelID int IDENTITY(1,1) NOT NULL PRIMARY KEY,  
    [Name] name NOT NULL,  
    CatalogDescription xml (CONTENT  
        Production.ProductDescriptionSchemaCollection),
```



```

Instructions xml (CONTENT
    Production.ManuInstructionsSchemaCollection),
rowguid uniqueidentifier ROWGUIDCOL NOT NULL DEFAULT (NEWID()),
ModifiedDate datetime NOT NULL DEFAULT (GETDATE())
);

```

The `Production.ManuInstructionsSchemaCollection` XML schema collection consists of the XML schema shown in Listing 3-6.

Listing 3-6. *Production.ManuInstructionsSchemaCollection XML Schema Collection*

```

CREATE XML SCHEMA COLLECTION Production.ManuInstructionsSchemaCollection
AS
N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:t="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
    ProductModelManuInstructions"
  targetNamespace="http://schemas.microsoft.com/sqlserver/2004/07/➤
    adventure-works/ProductModelManuInstructions"
  elementFormDefault="qualified">
  <xsd:element name="root">
    <xsd:complexType mixed="true">
      <xsd:complexContent mixed="true">
        <xsd:restriction base="xsd:anyType">
          <xsd:sequence>
            <xsd:element name="Location" maxOccurs="unbounded">
              <xsd:complexType mixed="true">
                <xsd:complexContent mixed="true">
                  <xsd:restriction base="xsd:anyType">
                    <xsd:sequence>
                      <xsd:element name="step"
                        type="t:StepType"
                        maxOccurs="unbounded" />
                    </xsd:sequence>
                  </xsd:restriction>
                </xsd:complexContent>
              </xsd:complexType>
            </xsd:element>
            <xsd:attribute name="LocationID"
              type="xsd:integer"
              use="required" />
            <xsd:attribute name="SetupHours"
              type="xsd:decimal" />
            <xsd:attribute name="MachineHours"
              type="xsd:decimal" />
            <xsd:attribute name="LaborHours"
              type="xsd:decimal" />
            <xsd:attribute name="LotSize"
              type="xsd:decimal" />
          </xsd:restriction>
        </xsd:complexContent>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:schema>

```

```

        </xsd:sequence>
    </xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
</xsd:element>
<xsd:complexType name="StepType" mixed="true">
    <xsd:complexContent mixed="true">
        <xsd:restriction base="xsd:anyType">
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="tool" type="xsd:string" />
                <xsd:element name="material" type="xsd:string" />
                <xsd:element name="blueprint" type="xsd:string" />
                <xsd:element name="specs" type="xsd:string" />
                <xsd:element name="diag" type="xsd:string" />
            </xsd:choice>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
</xsd:schema>';

```

This XML schema defines elements and attributes in the order in which they are expected to occur, the number of occurrences expected, and their data types. In Chapter 4, I will discuss the W3C XML Schema standard in detail and show you how to create your own XML schema collections.

DTDs

In addition to XML schema collections, the SQL Server 2008 `xml` type supports a very small subset of XML DTDs. You can define a DTD inline at the beginning of your XML data. SQL Server 2008 support for DTDs is limited to two functions:

- Inline DTDs can be used to assign default values to attributes in your XML data.
- Inline DTDs can be used to expand entity references in your XML data.

The W3C XML standard defines several other features of DTDs, including use of DTDs to constrain the structure and textual content of elements and attributes in XML data. The SQL Server `xml` data type does not support these DTD features, however. Instead of using DTDs to constrain content, you can use more expressive, robust, and powerful XML schema collections.

Note SQL Server does not support external DTDs, which are loaded from an external file during XML parsing. The standalone attribute in the `xml` declaration of the XML data is also ignored by SQL Server.

XML data that contains an inline DTD cannot be assigned directly to an `xml` instance; you must use the `CONVERT` function with the `style` parameter set to 2 or 3. An example of creating an XML instance with an inline DTD is shown in Listing 3-7. The `xml` variable contains the instance data shown in Figure 3-2 after application of the inline DTD.

Listing 3-7. *Creating an XML Instance with a DTD*

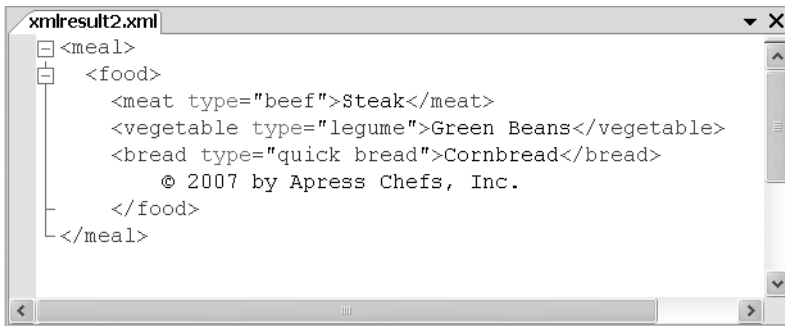
```

DECLARE @x xml;

SET @x = CONVERT(xml, N'<?xml version="1.0"?>
<!DOCTYPE meal [
    <!ATTLIST bread type CDATA "quick bread">
    <!ENTITY copyright "&#xA9; 2007 by Apress Chefs, Inc.">
]>
<meal>
    <food>
        <meat type="beef">Steak</meat>
        <vegetable type="legume">Green Beans</vegetable>
        <bread>Cornbread</bread>
        &copyright;
    </food>
</meal>', 2);

SELECT @x;

```

**Figure 3-2.** *XML data after application of inline DTD*

The DTD always begins with a `<!DOCTYPE` declaration followed by a name and an open bracket (`[`). The DTD ends with a closing `>` character combination.

The first thing to notice about the result is that SQL Server has expanded the entity `©right;`, defined in the DTD, to its full value “(c) 2007 by Apress Chefs, Inc.” The XML parser automatically expands entity references defined in the DTD out to their full representation. Following is the format to define your own entity references within the DTD:

```
<!ENTITY entity-name "entity-value">
```

The entity is referenced within the body of your XML data using the standard entity reference format `&#entity-name;`.

The next thing to notice is that SQL Server has added a default attribute value to the `<bread>` element. Defining default attribute values requires the following:

```
<!ATTLIST element-name attribute-name CDATA "default-value">
```

In this format, *element-name* and *attribute-name* refer to the specific element and attribute that *default-value* is tied to. CDATA is the attribute type and is effectively the only type recognized by SQL Server. The *default-value* is the default assigned to the attribute when no value is assigned explicitly within the XML data, as is the case with the `bread` element in Listing 3-7.

DTD LITE

Old hands at DTDs might recognize the absence of several DTD features from the SQL Server DTD implementation. Consider the enumerated lists of possible values, which are standard in the `<!ATTLIST>` declaration. DTD developers might be used to declarations like the following:

```
<!ATTLIST bread type (sliced | quick bread | roll) "quick bread">
```

This declaration limits the type attribute of the `bread` element to the values `sliced`, `quick bread`, and `roll`, with a default value of `"quick bread"`. SQL Server's limited support for DTDs, however, treats the enumerated list just like a CDATA attribute type, and the attribute is not limited to the list of enumerated values as you might expect.

The `<!ELEMENT>` declaration is another unsupported feature. While `<!ELEMENT>` declarations are properly parsed, they are basically ignored. SQL Server will properly parse syntactically correct DTDs, but apart from the supported features the declarations in the DTD are ignored. If your DTD contains syntax errors, however, SQL Server will return an error like the following:

XML DTD has been stripped from one or more XML fragments. External subsets, if any, have been ignored.

Keep these limitations in mind if you decide to use DTDs in your SQL Server XML.

You might also notice that the DTD itself is not stored with the XML in the `xml` instance. This is a result of SQL Server internally converting your XML data to a W3C XDM representation. The W3C XDM is based on the W3C Infoset and XML Schema. There are certain advantages to storing XML data using the XDM, which I'll cover in Chapter 4. For now, it's important to know that the internal representation of your XML is not an exact representation of the character string you assign or cast to an `xml` instance, and this behavior is consistent with the standard.

Cross-Platform Tip SQL Server uses the XDM to store your `xml` instances internally, but the ISO SQL/XML Standard (circa 2003) used the older W3C Infoset exclusively. The SQL/XML Standard has been recently updated to use the XDM, which provides greater flexibility and more power than the Infoset.

Using XML Type Methods

The `xml` data type features several built-in methods that allow you to manipulate XML instance data. These methods allow you to query, modify, or shred your XML data into relational form. The `xml` data type methods are listed in Table 3-2.

Table 3-2. *xml Data Type Methods*

Method	Description
<code>query()</code>	The <code>query()</code> method allows you to perform an XQuery on your <code>xml</code> instance. The result returned is untyped XML.
<code>value()</code>	The <code>value()</code> method allows you to perform an XQuery on your <code>xml</code> instance and returns a scalar value cast to a SQL Server data type.
<code>exist()</code>	The <code>exist()</code> method allows you to specify an XQuery on your <code>xml</code> instance and returns a SQL bit value of 1 if the XQuery returns a result, 0 if the XQuery returns no result, or NULL if the <code>xml</code> instance is NULL.
<code>modify()</code>	The <code>modify()</code> method allows you to execute XML Data Manipulation Language (XML DML) statements against an <code>xml</code> instance. The <code>modify()</code> method can only be used with a SET clause or statement.
<code>nodes()</code>	The <code>nodes()</code> method allows you to shred <code>xml</code> instances. <i>Shredding</i> is the process of converting your XML data to relational form.

As you can see from Table 3-2, the `xml` data type methods rely on SQL Server's XQuery and XML DML functionality. XQuery, in turn, relies on SQL Server's implementation of the W3C XDM. The `xml` data type methods are instance methods (as opposed to static methods) accessed using the `object.method()` notation, where the object is an `xml`-declared object, like a column or local variable. Each of the `xml` methods is described in further detail throughout the remainder of this chapter.

Cross-Platform Tip XML DML is not formally defined by the W3C, nor is it part of the ISO SQL/XML Standard.

Using the `query()` Method

To begin the discussion of the `query()` method, I'll solve a simple business problem involving a given XML document. For this example, I'll provide a sample XML document that comprises a list of bookstores in New York City. You need to retrieve the street address for each bookstore. Listing 3-8 demonstrates.

Listing 3-8. *Simple `query()` Method Example*

```
DECLARE @x xml;
SET @x = N'<?xml version = "1.0"?>
<bookstores company = "Borders Group">
  <store name = "Borders">
```

```

    <address>
      <street>2 PENN PLAZA</street>
      <city>NEW YORK</city>
      <state>NY</state>
      <postal-code>10121-0101</postal-code>
      <country>US</country>
      <geo>
        <lat>40.749278</lat>
        <long>-73.992078</long>
      </geo>
    </address>
  </store>
  <store name = "Waldenbooks">
    <address>
      <street>318 E FAIRMOUNT AVENUE</street>
      <city>LAKEWOOD</city>
      <state>NY</state>
      <postal-code>14750-2007</postal-code>
      <country>US</country>
      <geo>
        <lat>42.098387</lat>
        <long>-79.305532</long>
      </geo>
    </address>
  </store>
  <store name = "Borders Express">
    <address>
      <street>1401 ROUTE 300</street>
      <city>NEWBURGH</city>
      <state>NY</state>
      <postal-code>12550-2990</postal-code>
      <country>US</country>
      <geo>
        <lat>41.518067</lat>
        <long>-74.068843</long>
      </geo>
    </address>
  </store>
</bookstores>';

```

```
SELECT @x.query(N'//street');
```

The `xml` data type's `query()` method is designed to meet this type of requirement. As shown in the example, the `query()` method allows you to query XML data using the flexible and powerful XQuery language. Notice the form of the `query()` method, which is familiar to programmers in object-oriented languages like C#:

```
xml_obj.query ( xquery )
```

In this format, `xml_obj` represents the xml data type instance object and the `xquery` parameter is the XQuery string. The actual XQuery used in Listing 3-8 is `//street`. This query retrieves all nodes named `street` anywhere they occur in the `xml` variable instance. The `query()` method returns all `street` nodes as untyped XML. The result is shown in Figure 3-3.

Tip Most xml data type methods are called using the `xml_obj.query()` format. I'll point out the differences in this chapter for those xml methods that differ from this format.

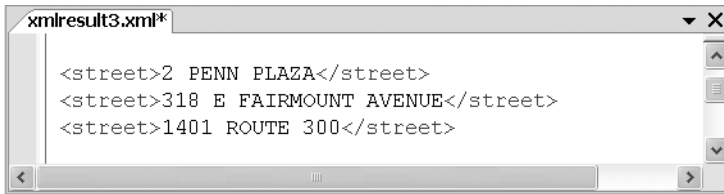


Figure 3-3. Result of simple XQuery

The `query()` method, and other xml data type methods, can handle fairly complex XQuery queries, including those that use complex path expressions, built-in functions, and XQuery for-let-where-order by-return (or FLWOR) expressions. You'll learn the details of XQuery and FLWOR expressions in Chapter 5, including how to create your own complex XQuery queries.

Cross-Platform Tip If you are coming from an ISO SQL/XML platform, it might help you to understand how SQL Server's xml data type methods relate to the standard. These Cross-Platform Tips will provide quick tips on how SQL Server xml methods relate to the ISO SQL/XML Standard. The `query()` method used in the example is analogous to the ISO SQL/XML Standard `XMLQUERY` function.

Using the value() Method

While querying XML data to retrieve a sequence of nodes is useful, sometimes you just want to retrieve a single scalar value from your XML. Consider an XML document containing book information, including price and release date. Listing 3-9 shows how to use the `value()` method to retrieve the price and release date as scalar SQL Server values from just such an XML document.

Listing 3-9. Simple `value()` Method Example

```
DECLARE @x xml;
SET @x = N'<?xml version = "1.0"?>
<book>
  <title>Harry Potter and the Deathly Hallows</title>
  <author>Rowling, J.K.</author>
  <isbn>0545010225</isbn>
```

```
<release-date>2007-07-21Z</release-date>
<price>34.99</price>
</book>';
```

```
SELECT @x.value(N'(/book/price)[1]', 'decimal(5, 2)') AS Price,
       @x.value(N'(/book/release-date)[1]', 'date') AS Release_Date;
```

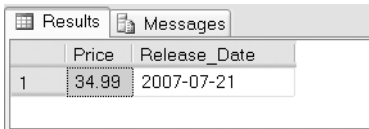
The `value()` method performs an XQuery query against an `xml` data instance and returns the result as a scalar value. The result is then cast to the SQL Server data type specified. The scalar value is cast to a specified SQL Server data type. The `value()` method accepts two parameters, an XQuery string and a SQL Server data type as a string, as shown in the following:

```
xml_obj.value ( xquery, data_type )
```

In this format, *xml_obj* is an `xml` data type instance object, *xquery* is the XQuery query to perform, and *data_type* is the string name of the SQL Server data type to which the result should be cast. The XQuery query submitted to the `value()` method must return a singleton atomic value. SQL Server has to be able to determine that a singleton atomic value will be returned during the static typing phase of the XQuery query compilation. You will often see queries followed by numeric predicates in the form `[n]`, where *n* is a positive integer greater than zero, to ensure that a statically typed singleton atomic value is returned.

Note The *data_type* specified for the `value()` method cannot be `xml`, `sql_variant`, a SQLCLR user-defined type, or one of the deprecated `text`, `ntext`, or `image` data types. Also notice that the *data_type* parameter is a character-type value with the name of the data type, so it should be enclosed in single quotes.

The two `value()` method calls retrieve the book's price and release-date, casting them to the SQL Server decimal and date data types, respectively. The two XQuery queries are `(/book/price)[1]` and `(/book/release-date)[1]`. Notice that both queries are followed immediately by the numeric predicate `[1]`, guaranteeing that a singleton atomic value is returned. (You'll explore numeric and other predicates, as well as the XQuery static typing system, in greater detail in Chapter 5.) The result of the sample query is shown in Figure 3-4.



	Price	Release_Date
1	34.99	2007-07-21

Figure 3-4. Result of `value()` method example

Cross-Platform Tip The `value()` method functionality can be replicated in the ISO SQL/XML Standard by using the `XMLQUERY` function and the standard SQL `CAST` function.

Using the exist() Method

Sometimes you want to retrieve a sequence of nodes or a scalar value from your XML, but other times you don't want to retrieve anything—you just want to know if a node exists or not. Listing 3-10 uses the `exist()` method against a simple restaurant dessert menu in XML format to determine if an item of type "pie" exists on the menu.

Listing 3-10. *exist() Method Example*

```
DECLARE @x xml;
SET @x = N'<?xml version = "1.0"?>
<dessert-menu>
  <item type = "pie">
    <name>Cherry Pie</name>
    <serving-info>
      <size>1 slice</size>
      <calories>277</calories>
      <price>2.99</price>
    </serving-info>
  </item>
  <item type = "cookie">
    <name>Peanut Butter Blossom</name>
    <serving-info>
      <size>3 cookies</size>
      <calories>348</calories>
      <price>1.29</price>
    </serving-info>
  </item>
  <item type = "cake">
    <name>German Chocolate Cake</name>
    <serving-info>
      <size>1 slice</size>
      <calories>793</calories>
      <price>3.99</price>
    </serving-info>
  </item>
</dessert-menu>';

SELECT CASE @x.exist(N'/dessert-menu/item[@type eq "pie"]')
        WHEN 1 THEN N'Pie is on the menu'
        WHEN 0 THEN N'Pie is not on the menu'
        ELSE 'The XML instance is NULL'
END;
```

The `exist()` method accepts an XQuery query as a parameter and returns a bit value indicating whether the query returns a nonempty result set. The possible return values of the `exist()` method are shown in Table 3-3. The result of the sample is shown in Figure 3-5.

Table 3-3. *exist() Method Return Values*

Return Value	Criteria
0	The XQuery query returns an empty result set.
1	The XQuery query returns a nonempty result set.
NULL	The <i>xml_obj</i> instance is NULL.

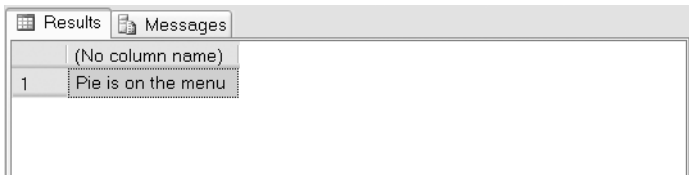


Figure 3-5. *Result of exist() method example*

The `exist()` method example introduces another feature of XQuery queries. Before I get into the details, let's take a look at the actual XQuery used in this example:

```
/dessert-menu/item[@type eq "pie"]
```

In this case, I am using a *value comparison* in the XQuery path predicate. The value comparison operator `eq` is the equality operator. It checks two singleton atomic values for equality. (I'll discuss value comparisons, path expressions, and XQuery predicates in great detail in Chapter 5.)

In this query, I am telling XQuery to return all `item` nodes under the root `dessert-menu` node, but I'm also telling XQuery to limit the results only to those `item` nodes that have a `type` attribute with a value equal to "pie". Since a node matching this path expression does exist in the XML data, the `exist()` method returns 1, and the CASE expression in the example uses the return value to return the string `Pie is on the menu`.

Cross-Platform Tip The `exist()` method is equivalent to the ISO SQL/XML `XMLEXISTS` predicate. While `exist()` returns 0, 1, or NULL, the `XMLEXISTS` predicate returns TRUE, FALSE, or UNKNOWN.

Using the `nodes()` Method

The ability to shred XML, or convert it to relational form, is very useful. It is often easier to manipulate and utilize XML data in relational form in SQL Server, as well as in many client applications. In SQL Server 2000, this could only be accomplished through the OPENXML rowset provider. While OPENXML is still around, current best practice is to use the `nodes()` method to shred your XML data. In Listing 3-11, I'll use the `xml` data type `nodes()` and `value()` methods to convert a simple bill of materials, often used in manufacturing operations, to relational format.

Listing 3-11. *nodes() Method Example*

```

DECLARE @x xml;
SET @x = N'<?xml version = "1.0"?>
<bill-of-materials>
  <finished-good name = "kiddie picnic table">
    <material name = "pine lumber">
      <item qty = "2">
        <dimensions uom = "mm">50 x 50 x 1100</dimensions>
      </item>
      <item qty = "4">
        <dimensions uom = "mm">50 x 25 x 800</dimensions>
      </item>
      <item qty = "8">
        <dimensions uom = "mm">50 x 25 x 400</dimensions>
      </item>
      <item qty = "2">
        <dimensions uom = "mm">50 x 50 x 475</dimensions>
      </item>
      <item qty = "4">
        <dimensions uom = "mm">50 x 50 x 180</dimensions>
      </item>
      <item qty = "6">
        <dimensions uom = "mm">50 x 50 x 75</dimensions>
      </item>
      <item qty = "5">
        <dimensions uom = "mm">100 x 25 x 800</dimensions>
      </item>
      <item qty = "8">
        <dimensions uom = "mm">50 x 25 x 800</dimensions>
      </item>
    </material>
    <material name="bolts">
      <item qty = "6">
        <dimensions uom = "mm">100 x 7</dimensions>
      </item>
      <item qty = "24">
        <dimensions uom = "mm">75 x 7</dimensions>
      </item>
    </material>
    <material name="washers">
      <item qty = "30">
        <dimensions uom = "mm">7</dimensions>
      </item>
    </material>
    <material name="nuts">
      <item qty = "30">
        <dimensions uom = "mm">7</dimensions>

```

```

    </item>
</material>
<material name = "8-gauge treated screws">
    <item qty = "36">
        <dimensions uom = "mm">45</dimensions>
    </item>
</material>
</finished-good>
</bill-of-materials>';

```

```

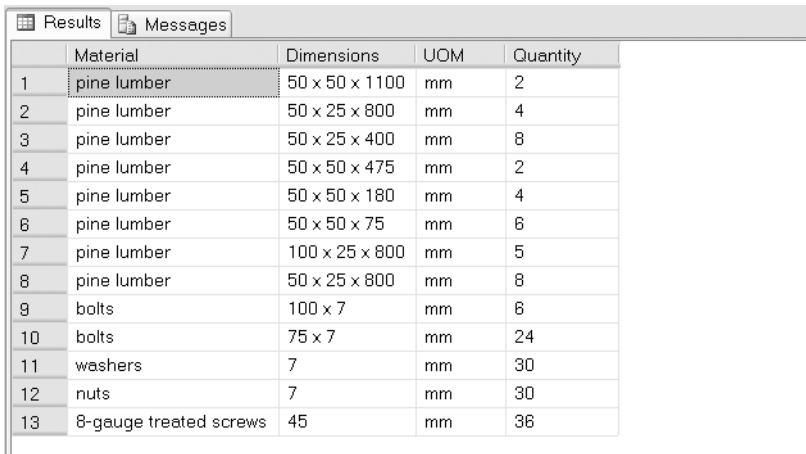
SELECT my_table. my_col.value(N'../@name', N'nvarchar(100)') AS Material,
       my_table. my_col.value(N'./dimensions[1]', N'nvarchar(50)') AS Dimensions,
       my_table. my_col.value(N'./dimensions/@uom[1]', N'nvarchar(10)') AS UOM,
       my_table. my_col.value(N'./@qty', N'int') AS Quantity
FROM @x.nodes(N'//item') AS my_table ( my_col );

```

The format for the `nodes()` method is slightly different from the standard `xml` method syntax, as shown in the following:

```
xml_obj.nodes ( xquery ) AS table ( column )
```

In this format, *xml_obj* is an `xml` data type object instance and the *xquery* parameter is an XQuery query. The *table* (*column*) alias specifies a virtual table and column name to be returned by the `nodes()` method. The column specified is an `xml` data type column, which you can use to further manipulate and query your XML data. Before I dive into the details of this example, I'll present the results as shown in Figure 3-6.



	Material	Dimensions	UOM	Quantity
1	pine lumber	50 x 50 x 1100	mm	2
2	pine lumber	50 x 25 x 800	mm	4
3	pine lumber	50 x 25 x 400	mm	8
4	pine lumber	50 x 50 x 475	mm	2
5	pine lumber	50 x 50 x 180	mm	4
6	pine lumber	50 x 50 x 75	mm	6
7	pine lumber	100 x 25 x 800	mm	5
8	pine lumber	50 x 25 x 800	mm	8
9	bolts	100 x 7	mm	6
10	bolts	75 x 7	mm	24
11	washers	7	mm	30
12	nuts	7	mm	30
13	8-gauge treated screws	45	mm	36

Figure 3-6. Results of `nodes()` method example

In Listing 3-11, I introduced even more XQuery language features. The `nodes()` method XQuery query looks like this:

```
//item
```

The leading forward slashes (//) are shorthand for the descendant-or-self axis step. This particular axis step tells XQuery to retrieve all nodes named `item` and their children, no matter where they occur within the XML data. (You'll learn more about axis steps in Chapter 5.) The `nodes()` method returns a virtual table aliased with the name `my_table` with a single `xml` data type column named `my_col`. The `value()` method is then used on the `my_col` column to shred the XML data into scalar values using XQuery. The XQuery queries used are as follows:

```
../@name
(../dimensions)[1]
(../dimensions/@uom)[1]
../@qty
```

One XQuery feature I'm using for the first time here is the *context node*. The context node is the current node being referenced by the query. In Listing 3-11 the context node for each row returned by the `nodes()` method is the `item` node returned for that row. The context node is specified in the XQuery query using the period (.) character in the path expression. The *reverse step indicator* (..) is an axis step, indicating that the parent node of the current context node should be referenced. In the example, the `../@name` XQuery query returns the value of the `name` attribute for the material node that is a parent for each `item` node.

Cross-Platform Tip The `nodes()` method, when used with the `values()` method, can be used to approximate the functionality of the ISO SQL/XML Standard `XMLTABLE` function.

Using the `modify()` Method

To round out SQL Server's XQuery functionality, the SQL Server team implemented XML DML. XML DML is defined as a set of extensions to XQuery that allow you to modify the data in your `xml` data type instances. As an example, consider a grocery store inventory in XML format, from which you want to delete a single item. Listing 3-12 uses XML DML to delete chunked white albacore from the inventory.

Listing 3-12. *modify()* Method Example

```
DECLARE @x xml;
SET @x = N'<?xml version = "1.0"?>
<inventory store-number = "9834">
  <product ean = "051500241776">
    <name>Jif Creamy Peanut Butter</name>
    <size>28 oz</size>
  </product>
  <product ean = "0024600010030">
    <name>Morton Iodized Salt</name>
    <size>26 oz</size>
  </product>
  <product ean = "0086600000138">
    <name>Bumble Bee Chunked White Albacore in Water</name>
```

```

    <size>6 oz</size>
  </product>
<product ean = "0013130006125">
  <name>Cream of Wheat Enriched Farina</name>
  <size>28 oz</size>
</product>
</inventory>';
```

```
SET @x.modify (N'delete (/inventory/product[@ean = "0086600000138"]/name)');

SELECT @x;
```

This use of the `modify()` method deletes the product node with an `ean` attribute set to "0086600000138", as well as all the child nodes it contains. The result is shown in Figure 3-7.

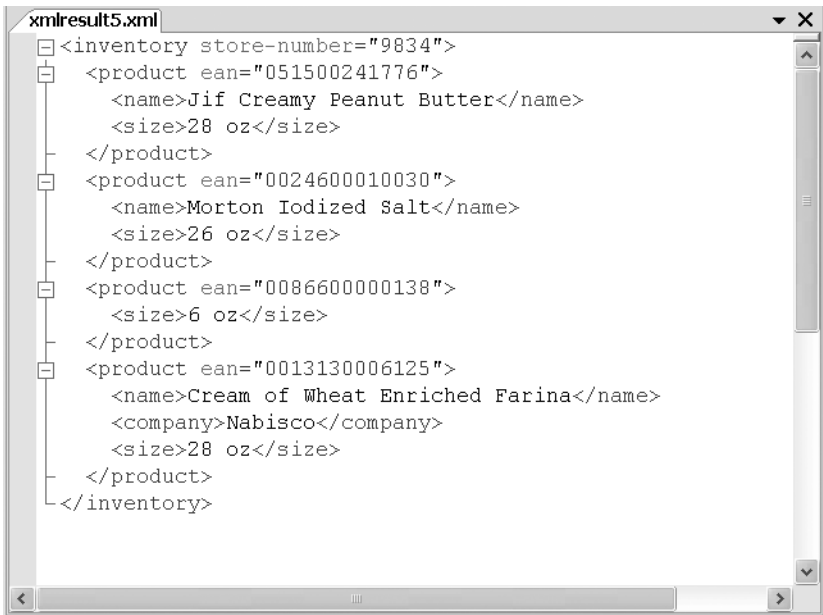


Figure 3-7. Result of the `modify()` method

XML DML supports three keywords, which are listed in Table 3-4. (I'll discuss the details of XML DML in Chapter 6.)

Table 3-4. XML DML Keywords

Keyword	Description
delete	Deletes the node specified by an XQuery path expression.
insert	Inserts one or more nodes as the children or siblings of a node specified by an XQuery path expression.
replace value of	Updates the value of a node specified by an XQuery path.

Summary

The `xml` data type was introduced in SQL Server 2005 as a truly integrated, first-class data type designed for manipulating, storing, and querying XML data. The `xml` data type provides many advantages over the legacy functions and procedures provided for handling XML server-side prior to SQL Server 2005. In this chapter, I covered the functionality and uses of the `xml` data type, the foundation of SQL Server 2008 XML support. Other topics covered also included the following:

- Declaring and populating an `xml` data type instance
- Restricting `xml` instances to well-formed and/or valid XML data
- Using the `xml` data type methods `query()`, `value()`, `exist()`, `nodes()`, and `modify()`
- Casting and converting `xml` instance data to and from character and binary data
- Using DTDs with the `xml` data type
- Declaring `xml` data type parameters and return values

I also introduced several key topics, which will be discussed in great detail in later chapters, including the following:

- Using XML schema collections to constrain XML data
- Utilizing the XQuery 1.0 and XPath 2.0 Data Model
- Performing queries using XQuery
- Modifying `xml` instances with XML DML

In the next chapter, you'll explore XML schema collections and dig into the details of SQL Server support for the W3C XDM.



XML Schema Collections

In this chapter, you will look at designing and implementing XML schema collections, which are used to create typed XML data. An XML schema collection uses the XQuery/XPath Data Model (XDM) to define the structure of, and constrain the values contained by, your typed `xml` instance objects. I will discuss SQL Server statements for creating and managing XML schema collections, and you will work with XML Schema data types. Throughout this chapter, you will build a sample XML schema collection to create a typed `xml` instance for a sample recursive Bill of Materials (BOM) in XML format.

Introducing XML Schema

XML Schema is the W3C recommendation that defines the structure and constrains the content of XML data. SQL Server supports a subset of the W3C XML Schema recommendation through XML schema collections.

XML schema collections are just what the name implies—collections of XML schemas imported into a SQL Server database. XML schemas are defined using the XML Schema Definition (XSD) language. I introduced XML schema collections in Chapter 3 with a look at the Adventure-Works Production.ManuInstructionsSchemaCollection schema collection. In this chapter, I'll dig into the details of XSD and how you can use it to create your own custom XML schema collections.

XML RECOMMENDATIONS

I will refer to several W3C XML-related recommendations throughout this book. These recommendations are freely available via the W3C web site at www.w3.org. Some of the more important recommendations I will be discussing in this chapter include the following:

- The XQuery 1.0 and XPath 2.0 Data Model Recommendation is available at www.w3.org/TR/xpath-datamodel/.
- The XML Information Set Recommendation is available at www.w3.org/TR/xml-infoset/.
- The XML Schema Recommendation, Parts 1 and 2, are available at www.w3.org/TR/xmlschema-1/ and www.w3.org/TR/xmlschema-2/, respectively.

I'll point out links to other W3C recommendations, and other standards, throughout this chapter and the rest of the book. I've also included a reference listing of essential W3C XML recommendations in Appendix A.

SQL Server provides the `CREATE XML SCHEMA COLLECTION`, `DROP XML SCHEMA COLLECTION`, and `ALTER XML SCHEMA COLLECTION` statements to register and manage your XML schema collections in the database.

XML schemas are created using XSD, the formal language that contains mechanisms for defining the structure, and constraining the content, of typed XML data. XSD is defined in terms of XML, so an XML schema document is actually an XML document. I will focus on XSD as implemented by SQL Server 2008 in this chapter. An XSD document defines the following for a given set of XML documents:

- Order of elements and attributes in the document
- Hierarchical structure of elements in the document
- Whether or not a given element in the document can include text
- Data types of elements and attributes in the document
- Defaults and fixed values for elements and attributes in the document

POST-SCHEMA-VALIDATION INFOSET

I mentioned before that the `xml` data type does not store a literal representation of your XML data. Instead, it stores a Post-Schema-Validation Infoset (PSVI). The PSVI is based on the XML Information Set (Infoset) Recommendation, which defines an abstract model for representing XML documents. The PSVI is an augmented datatype-aware Infoset that contains XML Schema data type information. XML documents that undergo the process of XML Schema validation are converted to the PSVI model for querying and manipulation efficiency. Conversion from a literal XML document to a PSVI object is a one-way operation. The PSVI strips some meta-data and extraneous information (such as character references and noncritical white space) during the conversion. Because of this, the literal XML document cannot be re-created from the PSVI. PSVI instances map to XDM instances, which I will discuss in the next chapter.

XML data that conforms to the structure and constraints of an XSD document is considered *valid*. A SQL Server `xml` data type instance that is associated with an XML schema collection is *typed*. It's a good practice to always reference your XSD documents to the `http://www.w3.org/2001/XMLSchema` namespace, as shown in Listing 4-1.

Listing 4-1. XML Schema Namespace Declaration

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- XSD content removed -->
</xsd:schema>
```

A NOTE ABOUT NAMESPACES

A namespace is a mechanism for qualifying XML element and attribute names, allowing multiple identifiers from different namespaces to have the same name. Namespaces are declared using attributes that take the following form:

```
xmlns:prefix="URI"
```

The *prefix* in the declaration is the namespace prefix used throughout the document to refer to the XML namespace. The *URI* in the definition is a quote-delimited Uniform Resource Identifier (URI). The SQL Server AdventureWorks XML schema collections use the namespace prefix `xsd` to refer to the XML Schema URI. The relevant W3C recommendations use the namespace prefix `xs`. The point being that the namespace prefix you assign doesn't matter, so long as the URI is properly set to `http://www.w3.org/2001/XMLSchema`. For simplicity's sake, and for uniformity with the AdventureWorks samples, I will use the `xsd` namespace prefix for the XML Schema namespace throughout this book.

The W3C Recommendation for Namespaces in XML 1.0 is located at www.w3.org/TR/REC-xml-names/.

Documenting with Annotations

XSD supports annotations, providing a way to add both human-readable and machine-readable materials to your XML documents. The `<annotation>` element can appear below the `<schema>` element or at the beginning of most other XML schema constructions, and can contain either of two child elements: `<documentation>` or `<appinfo>`.

The difference between these two `<annotation>` child elements is in their recommended usage. The `<documentation>` element is the preferred method for adding human-readable documentation to your XML schema. The `<appinfo>` element, on the other hand, is used to provide information to applications. Listing 4-2 adds a simple `<annotation>` element to the previous listing.

Listing 4-2. Adding an `<annotation>` Element to the XSD Element

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation>
      This XML schema will ultimately define a recursive Bill of Materials.
    </xsd:documentation>
  </xsd:annotation>
  <!-- XSD content removed -->
</xsd:schema>
```

Note The difference between the `<documentation>` and `<appinfo>` annotation elements is demonstrated in a sample XML schema in the XML Schema Part 2 document at www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#schema.

Using Declaration Components

The XML Schema recommendation supports three types of declaration components: `<element>` declarations, `<attribute>` declarations, and `<notation>` declarations. These declaration components define the structure of your typed XML data. One of the simplest XML schemas you can create includes only a single simple element declaration, as shown in Listing 4-3.

Note In this section, I'll build on the XML schema to add new features as they are discussed. As I build on the code samples, I'll highlight the changes to the XML schema (and other important code) with **bold** lettering.

Listing 4-3. *Simple XML Schema*

```
DECLARE @schema xml;
SET @schema = N'<?xml version = "1.0"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "item" />
</xsd:schema>';

CREATE XML SCHEMA COLLECTION dbo.SimpleTestSchemaCollection
AS @schema;
GO

DECLARE @x xml ( DOCUMENT dbo.SimpleTestSchemaCollection );
SET @x = N'<?xml version = "1.0"?>
<item>
</item>';

SELECT @x;
GO

DROP XML SCHEMA COLLECTION dbo.SimpleTestSchemaCollection;
```

You can use a string literal to define your XSD document when creating an XML schema collection, but in this example I chose to use an `xml` instance variable. I'll consider other examples later that use a string literal instead. The XSD document in the example has a root `<schema>` element and contains a single `<element>` declaration, as shown in Listing 4-4.

Listing 4-4. *Simple XML Schema*

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "item" />
</xsd:schema>
```

A valid typed XML document for this XML schema consists of a root element named `<item>`, as illustrated in Figure 4-1. Because I have not introduced any other constraints on the content,

the root element may contain any other type of content or no content (as in the example). Any XML document that does not conform to the XML schema collection will cause a validation error.



Figure 4-1. A valid XML document based on a simple XML schema

All XML schemas must begin with the `<schema>` element. The `<schema>` element is always the root element, and it has no parent elements. The `<element>` declaration included in the sample is the simplest possible declaration I could make, and it includes only the recommended name attribute. I will use XSD to develop more complex XML schemas as I progress through this chapter, including optional attributes of some of the most common XSD elements.

Note I'll be diving into the details of XSD in this chapter, including sample XML schemas and descriptions of XML schema components. Appendix C contains a concise guide to XSD elements and attributes supported by SQL Server.

Creating Complex Elements

The example in Listing 4-3 contains a *simple element*. Unlike simple elements, *complex elements* can contain other elements or attributes, or they can be defined so that they must be empty. You define an XSD complex type using the `<complexType>` element, and you can use three *model group schema components* to define complex elements in XSD:

- The `<all>` model group schema component tells XSD that each child element of a given element may appear in any order.
- The `<choice>` model group schema component tells XSD that only one child element of a given element may appear.
- The `<sequence>` model group schema component tells XSD that all child elements of a given element must occur in the specified order.

Listing 4-5 looks at creating a more expressive XML schema with a complex type using the `<all>` model group schema component.

Listing 4-5. *XML Schema and Valid XML Document with Complex Type Definition*

```

CREATE XML SCHEMA COLLECTION dbo.ComplexTestSchemaCollection_all
AS
N'<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="item">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="id" />
        <xsd:element name="number" />
        <xsd:element name="name" />
        <xsd:element name="color" />
        <xsd:element name="list-price" />
        <xsd:element name="standard-cost" />
        <xsd:element name="size" />
        <xsd:element name="unit-of-measure" />
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>';
GO

DECLARE @x XML (dbo.ComplexTestSchemaCollection_all);

SET @x = N'<?xml version="1.0"?>
<item>
  <id>749</id>
  <name>Road-150 Red, 62</name>
  <number>BK-R93R-62</number>
  <color>Red</color>
  <standard-cost>2171.2942</standard-cost>
  <list-price>3578.27</list-price>
  <unit-of-measure>CM</unit-of-measure>
  <size>62</size>
</item>';

SELECT @x;
GO

DROP XML SCHEMA COLLECTION dbo.ComplexTestSchemaCollection_all;

```

In this example, all the elements defined within the `<all>` model group schema component may appear in the valid XML document in any order. To prove this last point, I've changed the order of some of the elements from the order that they are specified in the XML schema. The result of applying the XML schema to the valid XML document in Listing 4-5 is shown in Figure 4-2.

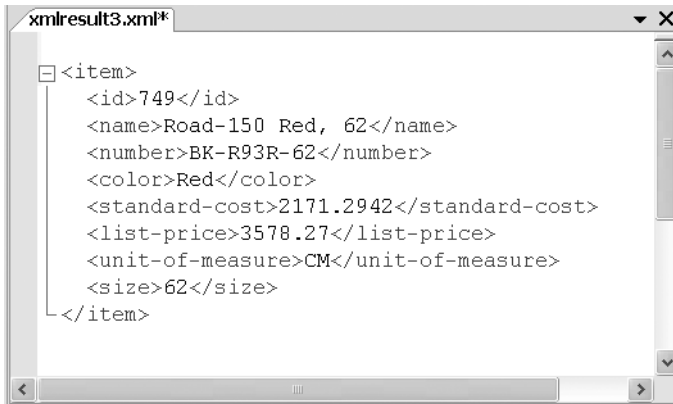


Figure 4-2. Result of applying complex type XML schema to valid XML document

Note The <all> model group schema component can only include <element> and <annotation> elements. It cannot contain groups or other model group schema components.

Listing 4-6 demonstrates use of the <sequence> and <choice> model group schema components.

Listing 4-6. XML Schema and XML Document with Complex Type Using <sequence> and <choice>

```
CREATE XML SCHEMA COLLECTION dbo.ComplexTestSchemaCollection_sequence_choice
AS
N'<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="item">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:choice>
          <xsd:element name="id" />
          <xsd:element name="number" />
        </xsd:choice>
        <xsd:element name="name" />
        <xsd:element name="color" />
        <xsd:choice>
          <xsd:element name="list-price" />
          <xsd:element name="standard-cost" />
        </xsd:choice>
        <xsd:sequence>
          <xsd:element name="size" />
          <xsd:element name="unit-of-measure" />
        </xsd:sequence>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:schema>';
GO

DECLARE @x XML (dbo.ComplexTestSchemaCollection_sequence_choice);

SET @x = N'<?xml version="1.0"?>
<item>
    <id>749</id>
    <name>Road-150 Red, 62</name>
    <color>Red</color>
    <list-price>3578.27</list-price>
    <size>62</size>
    <unit-of-measure>CM</unit-of-measure>
</item>';

SELECT @x;
GO

DROP XML SCHEMA COLLECTION dbo.ComplexTestSchemaCollection_sequence_choice;

```

The XML schema in Listing 4-6 demonstrates nested model group schema components, with a `<sequence>` component that includes `<choice>`, `<sequence>`, and `<element>` child components. A `<choice>` component indicates that only one of its child elements can appear in a valid XML document, as in the following snippet from Listing 4-6.

```

<xsd:choice>
    <xsd:element name="id" />
    <xsd:element name="number" />
</xsd:choice>

```

This `<choice>` component allows you to supply either an `<id>` element or a `<number>` element, but not both, in your XML document. All child elements of the `<sequence>` component must appear in the XML document in the same order they are specified in the XML schema. The following snippet from Listing 4-6 shows the elements `<size>` and `<unit-of-measure>`, which must occur in your valid XML document in that order.

```

<xsd:sequence>
    <xsd:element name="size" />
    <xsd:element name="unit-of-measure" />
</xsd:sequence>

```

Defining Model Groups

Model group definitions are another powerful feature of XSD. Using model group definitions, you can create reusable model groups that can be referenced in your XML schema. Model group definitions are specified with the `<group>` element. Listing 4-7 includes a rework of the XML schema in Listing 4-6 using model group definitions.

Listing 4-7. *Complex XML Schema and XML Document with Model Group Definitions*

```

CREATE XML SCHEMA COLLECTION dbo.ComplexTestSchemaCollection_group
AS
N'<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="item">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref="id-group" />
        <xsd:element name="name" />
        <xsd:element name="color" />
        <xsd:group ref="price-group" />
        <xsd:group ref="size-group" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:group name="id-group">
    <xsd:choice>
      <xsd:element name="id" />
      <xsd:element name="number" />
    </xsd:choice>
  </xsd:group>

  <xsd:group name="price-group">
    <xsd:choice>
      <xsd:element name="list-price" />
      <xsd:element name="standard-cost" />
    </xsd:choice>
  </xsd:group>

  <xsd:group name="size-group">
    <xsd:sequence>
      <xsd:element name="size" />
      <xsd:element name="unit-of-measure" />
    </xsd:sequence>
  </xsd:group>

</xsd:schema>';
GO

DECLARE @x XML (dbo.ComplexTestSchemaCollection_group);

SET @x = N'<?xml version="1.0"?>
<item>
  <id>749</id>
  <name>Road-150 Red, 62</name>

```

```

    <color>Red</color>
    <list-price>3578.27</list-price>
    <size>62</size>
    <unit-of-measure>CM</unit-of-measure>
  </item>';

```

```

SELECT @x;
GO

```

```

DROP XML SCHEMA COLLECTION dbo.ComplexTestSchemaCollection_group;

```

In Listing 4-7, the `<id>` and `<number>` elements are grouped together in a `<choice>` model group component as before. The `<choice>` model group component is then added to the `<id-group>` model group definition. Finally, this model group definition is referenced within the `<complexType>` definition. The example also uses an `id-price` `<choice>` group and a `size-group` `<sequence>` group.

The model group definition is roughly analogous to a *class* in an object-oriented language. The model group definition requires a definition, as in the following code snippet:

```

<xsd:group name="id-group">
  <xsd:choice>
    <xsd:element name="id" />
    <xsd:element name="number" />
  </xsd:choice>
</xsd:group>

```

Model group definitions are always *top-level* schema components, nested just within the `<schema>` component. To continue the object-oriented programming analogy, an *instance* of the model group definition is created via a reference. The reference to the model group definition requires a `<group>` element with a `ref` attribute set to the name of the model group definition, as shown in the following code line:

```

<xsd:group ref="id-group" />

```

As you saw in Listing 4-7, model group definitions are useful. Some of the advantages of using model group definitions include the following:

- **Code reuse.** XSD model group definitions are reusable, which can make your XML schemas more compact.
- **Easier debugging.** XSD model group definitions provide modularity, which can make it easier to locate, fix, and test issues in your XML schemas.
- **Faster development.** XSD model group definitions can make XML schemas more readable and cut down development time, particularly for applications developers and programmers who are used to modular programming constructs.
- **Simplified maintenance.** XSD model group definitions make modeling XML schemas and managing updates and modifications easier.

Adding Attributes

Attributes can be added to your type definitions with the `<attribute>` declaration schema component. I will modify the sample XML schema from Listing 4-7 by eliminating the `<id>` and `<number>` elements and turning them into attributes of the `<item>` element. Listing 4-8 shows the modified XML schema collection.

Listing 4-8. *Complex Schema Collection with Attributes*

```
CREATE XML SCHEMA COLLECTION dbo.ComplexTestSchemaCollection_attribute
AS
N'<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="item">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" />
        <xsd:element name="color" />
        <xsd:group ref="price-group" />
        <xsd:group ref="size-group" />
      </xsd:sequence>
      <xsd:attribute name="id" />
      <xsd:attribute name="number" />
    </xsd:complexType>
  </xsd:element>

  <xsd:group name="price-group">
    <xsd:choice>
      <xsd:element name="list-price" />
      <xsd:element name="standard-cost" />
    </xsd:choice>
  </xsd:group>

  <xsd:group name="size-group">
    <xsd:sequence>
      <xsd:element name="size" />
      <xsd:element name="unit-of-measure" />
    </xsd:sequence>
  </xsd:group>

</xsd:schema>';
GO

DECLARE @x XML (dbo.ComplexTestSchemaCollection_attribute);

SET @x = N'<?xml version="1.0"?>
<item id="749" number="BK-R93R-62">
  <name>Road-150 Red, 62</name>
```

```

    <color>Red</color>
    <list-price>3578.27</list-price>
    <size>62</size>
    <unit-of-measure>CM</unit-of-measure>
  </item>';

```

```

SELECT @x;
GO

```

```

DROP XML SCHEMA COLLECTION dbo.ComplexTestSchemaCollection_attribute;

```

The result of applying this XML schema to the sample XML data in the example is shown in Figure 4-3.

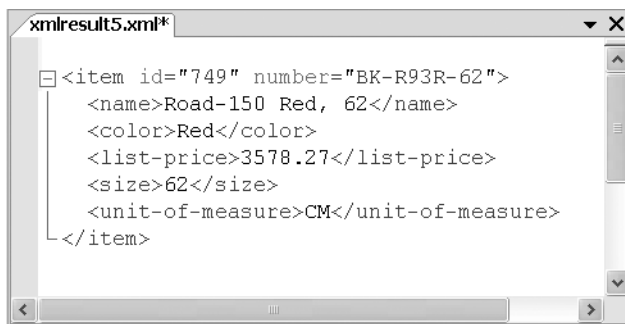


Figure 4-3. Result of sample XML schema with attributes applied to sample data

XSD also provides an attribute-grouping mechanism—the `<attributeGroup>` attribute group definition schema component. The `<attributeGroup>` component can be used to group together attributes for reuse, similarly to the way the `<group>` element can be used to create model group definitions. Listing 4-9 updates Listing 4-8 to use an attribute group definition schema component.

Listing 4-9. Sample XML Schema with Attribute Group Definition Schema Component

```

CREATE XML SCHEMA COLLECTION dbo.ComplexTestSchemaCollection_attrGroup
AS
N'<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="item">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" />
        <xsd:element name="color" />
        <xsd:group ref="price-group" />
        <xsd:group ref="size-group" />
      </xsd:sequence>
      <xsd:attributeGroup ref="id-attr-group" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

```

    </xsd:complexType>
</xsd:element>

<xsd:attributeGroup name="id-attr-group">
    <xsd:attribute name="id" />
    <xsd:attribute name="number" />
</xsd:attributeGroup>

<xsd:group name="price-group">
    <xsd:choice>
        <xsd:element name="list-price" />
        <xsd:element name="standard-cost" />
    </xsd:choice>
</xsd:group>

<xsd:group name="size-group">
    <xsd:sequence>
        <xsd:element name="size" />
        <xsd:element name="unit-of-measure" />
    </xsd:sequence>
</xsd:group>

</xsd:schema>';
GO

DECLARE @x XML (dbo.ComplexTestSchemaCollection_attrGroup);

SET @x = N'<?xml version="1.0"?>
<item id="749" number="BK-R93R-62">
    <name>Road-150 Red, 62</name>
    <color>Red</color>
    <list-price>3578.27</list-price>
    <size>62</size>
    <unit-of-measure>CM</unit-of-measure>
</item>';

SELECT @x;
GO

DROP XML SCHEMA COLLECTION dbo.ComplexTestSchemaCollection_attrGroup;

```

Constraining Occurrences

Many of the XSD elements allow you to define a minimum and maximum number of occurrences using the `minOccurs` and `maxOccurs` attributes. The `minOccurs` and `maxOccurs` attributes constrain elements or groups to a specific number of occurrences within your XML documents. They can also be used to define a *recursive* element, an element that can contain an element

of its same type as a child. This is useful functionality in many applications, such as Bill of Materials (also known as “parts explosions” or BOMs). I discussed generating a recursive BOM for AdventureWorks products with the FOR XML clause in Chapter 2. Here I will build on that example and create an XML schema for the BOM that was generated in Listing 2-23. Listing 4-10 defines the XML schema for the basic recursive BOM. The example shows a partial BOM for an AdventureWorks Road-150 Red, 62 bicycle, product number 749.

Listing 4-10. *Sample Nested BOM with Occurrence Constraints*

```
CREATE XML SCHEMA COLLECTION dbo.ComplexBOMSchema
AS
N'<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="items">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="item">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" minOccurs="1" maxOccurs="1" />
        <xsd:element name="color" minOccurs="0" maxOccurs="1" />
        <xsd:group ref="price-group" minOccurs="1" maxOccurs="1" />
        <xsd:element name="quantity" minOccurs="1" maxOccurs="1" />
        <xsd:group ref="size-group" minOccurs="0" maxOccurs="1" />
        <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
      <xsd:attributeGroup ref="id-attr-group" />
    </xsd:complexType>
  </xsd:element>

  <xsd:attributeGroup name="id-attr-group">
    <xsd:attribute name="id" />
    <xsd:attribute name="number" />
  </xsd:attributeGroup>

  <xsd:group name="price-group">
    <xsd:choice>
      <xsd:element name="list-price" minOccurs="1" maxOccurs="1" />
      <xsd:element name="standard-cost" minOccurs="1" maxOccurs="1" />
    </xsd:choice>
  </xsd:group>
```

```

<xsd:group name="size-group">
  <xsd:sequence>
    <xsd:element name="size" minOccurs="1" maxOccurs="1" />
    <xsd:element name="unit-of-measure" minOccurs="1" maxOccurs="1" />
  </xsd:sequence>
</xsd:group>

</xsd:schema>';
GO

DECLARE @x XML (dbo.ComplexBOMSchema);

SET @x = N'<?xml version="1.0"?>
<items>
  <item id="749" number="BK-R93R-62">
    <name>Road-150 Red, 62</name>
    <color>Red</color>
    <list-price>3578.2700</list-price>
    <quantity>1.00</quantity>
    <size>62</size>
    <unit-of-measure>CM </unit-of-measure>
  <item id="519" number="SA-R522">
    <name>HL Road Seat Assembly</name>
    <list-price>196.9200</list-price>
    <quantity>1.00</quantity>
  </item>
  <item id="717" number="FR-R92R-62">
    <name>HL Road Frame - Red, 62</name>
    <color>Red</color>
    <list-price>1431.5000</list-price>
    <quantity>1.00</quantity>
    <size>62</size>
    <unit-of-measure>CM </unit-of-measure>
  </item>
  <item id="807" number="HS-3479">
    <name>HL Headset</name>
    <list-price>124.7300</list-price>
    <quantity>1.00</quantity>
  </item>
  <item id="813" number="HB-R956">
    <name>HL Road Handlebars</name>
    <list-price>120.2700</list-price>
    <quantity>1.00</quantity>
  </item>
  <item id="820" number="FW-R820">
    <name>HL Road Front Wheel</name>
    <color>Black</color>

```

```
<list-price>330.0600</list-price>
<quantity>1.00</quantity>
</item>
<item id="828" number="RW-R820">
  <name>HL Road Rear Wheel</name>
  <color>Black</color>
  <list-price>357.0600</list-price>
  <quantity>1.00</quantity>
</item>
<item id="894" number="RD-2308">
  <name>Rear Derailleur</name>
  <color>Silver</color>
  <list-price>121.4600</list-price>
  <quantity>1.00</quantity>
</item>
<item id="907" number="RB-9231">
  <name>Rear Brakes</name>
  <color>Silver</color>
  <list-price>106.5000</list-price>
  <quantity>1.00</quantity>
</item>
<item id="940" number="PD-R853">
  <name>HL Road Pedal</name>
  <color>Silver/Black</color>
  <list-price>80.9900</list-price>
  <quantity>1.00</quantity>
</item>
<item id="945" number="FD-2342">
  <name>Front Derailleur</name>
  <color>Silver</color>
  <list-price>91.4900</list-price>
  <quantity>1.00</quantity>
</item>
<item id="948" number="FB-9873">
  <name>Front Brakes</name>
  <color>Silver</color>
  <list-price>106.5000</list-price>
  <quantity>1.00</quantity>
</item>
<item id="951" number="CS-9183">
  <name>HL Crankset</name>
  <color>Black</color>
  <list-price>404.9900</list-price>
  <quantity>1.00</quantity>
</item>
<item id="952" number="CH-0234">
  <name>Chain</name>
```



```

p.Name AS "name",
p.Color AS "color",
p.ListPrice AS "list-price",
c.PerAssemblyQty AS "quantity",
p.Size AS "size",
p.SizeUnitMeasureCode AS "unit-of-measure",
(
  SELECT d.ComponentID AS "@id",
    p.ProductNumber AS "@number",
    p.Name AS "name",
    p.Color AS "color",
    p.ListPrice AS "list-price",
    d.PerAssemblyQty AS "quantity",
    p.Size AS "size",
    p.SizeUnitMeasureCode AS "unit-of-measure",
  (
    SELECT e.ComponentID AS "@id",
      p.ProductNumber AS "@number",
      p.Name AS "name",
      p.Color AS "color",
      p.ListPrice AS "list-price",
      e.PerAssemblyQty AS "quantity",
      p.Size AS "size",
      p.SizeUnitMeasureCode AS "unit-of-measure"
    FROM Production.BillofMaterials e
    INNER JOIN Production.Product p
      ON e.ComponentID = p.ProductID
    WHERE e.ProductAssemblyID = d.ComponentID
    FOR XML PATH (N'item'), TYPE
  )
  FROM Production.BillofMaterials d
  INNER JOIN Production.Product p
    ON d.ComponentID = p.ProductID
  WHERE d.ProductAssemblyID = c.ComponentID
  FOR XML PATH (N'item'), TYPE
)
FROM Production.BillofMaterials c
INNER JOIN Production.Product p
  ON c.ComponentID = p.ProductID
WHERE c.ProductAssemblyID = b.ComponentID
FOR XML PATH (N'item'), TYPE
)
FROM Production.BillofMaterials b
INNER JOIN Production.Product p
  ON b.ComponentID = p.ProductID
WHERE b.ProductAssemblyID = a.ComponentID
FOR XML PATH(N'item'), TYPE

```

```

)
FROM Production.BillOfMaterials a
INNER JOIN Production.Product p
  ON a.ComponentID = p.ProductID
WHERE p.ProductID = @ProductID
FOR XML PATH(N'item'), ROOT(N'items'), TYPE;

```

The `minOccurs` and `maxOccurs` attributes of the XSD document are used on elements and model group definition references to restrict the number of times a given element or group can occur in the XML document. I've added the `minOccurs` and `maxOccurs` attributes to several elements in the sample XML schema, affecting the structure of the XML documents that it can validate.

Note The `<element>`, `<group>`, and `<any>` elements allow `minOccurs` and `maxOccurs` attributes if they are not directly below the `<schema>` element.

The first thing to notice is that the `<name>` element is defined with `minOccurs="1"` and `maxOccurs="1"`, which means it must occur in the XML document exactly once per `<item>`. The `<color>` element, on the other hand, is defined with `minOccurs="0"` and `maxOccurs="1"`. This makes the `<color>` element optional, but if it is used, it can only be included once per `<item>` element.

The `<price-group>` model group definition reference is defined with `minOccurs="1"` and `maxOccurs="1"`, so this group must occur exactly once per `<item>` also. The elements that make up the `<price-group>`, `<list-price>` and `<standard-cost>`, are both defined with `minOccurs="1"` and `maxOccurs="1"`. Because this is a `<choice>` group, exactly one of these two items must occur exactly once per `<item>` element.

The `<size-group>` model group definition reference is defined with `minOccurs="0"` and `maxOccurs="1"`, making a reference to this model group in your XML document optional. The elements that make up this group, `<size>` and `<unit-of-measure>`, are defined with `minOccurs="1"` and `maxOccurs="1"`. The net result of these occurrence constraints is that the `<size>` and `<unit-of-measure>` elements are optional, but if you include one, you must include both in the specified order.

One additional subelement, an `<item>` element reference, has been added within the `<item>` element. This subelement has been set to `minOccurs="0"` and `maxOccurs="unbounded"`. The result is that the `<item>` element can now contain any number of `<item>` subelements, nested to any depth. This nesting of `<item>` elements within `<item>` elements makes the `<item>` element recursive.

Caution It's important to set `minOccurs="0"` when defining a recursive element so that the recursion depth can reach an end. If `minOccurs` is greater than 0 in a recursive element definition, the recursion can never end and your XML schema will ultimately error out.

The result of adding these additional occurrence constraints to validate a recursive BOM in XML format is shown in Figure 4-4.



Figure 4-4. Result of recursive XSD applied to Bill Of Materials

Extending XML Schemas with Wildcards

XSD supplies the `<any>` and `<anyAttribute>` wildcard schema components that let you define the undefined. These two schema components provide additional flexibility by allowing you to include elements and attributes in your XML documents that are not defined by the local XML schema. Listing 4-12 extends the previous listing to include an `<any>` schema component.

Listing 4-12. Sample XML Schema with `<any>` Schema Component

```
USE AdventureWorks;
GO
CREATE XML SCHEMA COLLECTION dbo.ComplexBOMSchema_Wildcard
AS
N'<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="items">
    <xsd:complexType>
```

```

        <xsd:sequence>
            <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="item">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="name" minOccurs="1" maxOccurs="1" />
            <xsd:element name="color" minOccurs="0" maxOccurs="1" />
            <xsd:group ref="price-group" minOccurs="1" maxOccurs="1" />
            <xsd:element name="quantity" minOccurs="1" maxOccurs="1" />
            <xsd:group ref="size-group" minOccurs="0" maxOccurs="1" />
            <xsd:any processContents="lax" namespace="##other" minOccurs="0"
                maxOccurs="unbounded" />
            <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attributeGroup ref="id-attr-group" />
    </xsd:complexType>
</xsd:element>

<xsd:attributeGroup name="id-attr-group">
    <xsd:attribute name="id" />
    <xsd:attribute name="number" />
</xsd:attributeGroup>

<xsd:group name="price-group">
    <xsd:choice>
        <xsd:element name="list-price" minOccurs="1" maxOccurs="1" />
        <xsd:element name="standard-cost" minOccurs="1" maxOccurs="1" />
    </xsd:choice>
</xsd:group>

<xsd:group name="size-group">
    <xsd:sequence>
        <xsd:element name="size" minOccurs="1" maxOccurs="1" />
        <xsd:element name="unit-of-measure" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
</xsd:group>

</xsd:schema>';
GO

DECLARE @ProductID int;
SET @ProductID = 749;

```

```

DECLARE @x XML (dbo.ComplexBOMSchema_Wildcard);

WITH XMLNAMESPACES ('urn:apress:bom-misc' AS misc)
SELECT @x =
(
    SELECT a.ComponentID AS "@id",
    p.ProductNumber AS "@number",
    p.Name AS "name",
    p.Color AS "color",
    p.ListPrice AS "list-price",
    a.PerAssemblyQty AS "quantity",
    p.Size AS "size",
    p.SizeUnitMeasureCode AS "unit-of-measure",
    (
        SELECT RTRIM(pmpdc.CultureID) AS "misc:description/@xml:lang",
        pd.Description AS "misc:description"
        FROM Production.Product p1
        INNER JOIN Production.ProductModel pm
            ON p1.ProductModelID = pm.ProductModelID
        INNER JOIN Production.ProductModelProductDescriptionCulture pmpdc
            ON pm.ProductModelID = pmpdc.ProductModelID
        INNER JOIN Production.ProductDescription pd
            ON pmpdc.ProductDescriptionID = pd.ProductDescriptionID
        WHERE p1.ProductID = a.ComponentID
        FOR XML PATH (''), ROOT('misc:product-info'), TYPE
    ),
),
(
    SELECT b.ComponentID AS "@id",
    p.ProductNumber AS "@number",
    p.Name AS "name",
    p.Color AS "color",
    p.ListPrice AS "list-price",
    b.PerAssemblyQty AS "quantity",
    p.Size AS "size",
    p.SizeUnitMeasureCode AS "unit-of-measure",
    (
        SELECT c.ComponentID AS "@id",
        p.ProductNumber AS "@number",
        p.Name AS "name",
        p.Color AS "color",
        p.ListPrice AS "list-price",
        c.PerAssemblyQty AS "quantity",
        p.Size AS "size",
        p.SizeUnitMeasureCode AS "unit-of-measure",
        (
            SELECT d.ComponentID AS "@id",
            p.ProductNumber AS "@number",

```

```

    p.Name AS "name",
    p.Color AS "color",
    p.ListPrice AS "list-price",
    d.PerAssemblyQty AS "quantity",
    p.Size AS "size",
    p.SizeUnitMeasureCode AS "unit-of-measure",
    (
        SELECT e.ComponentID AS "@id",
            p.ProductNumber AS "@number",
            p.Name AS "name",
            p.Color AS "color",
            p.ListPrice AS "list-price",
            e.PerAssemblyQty AS "quantity",
            p.Size AS "size",
            p.SizeUnitMeasureCode AS "unit-of-measure"
        FROM Production.BillofMaterials e
        INNER JOIN Production.Product p
            ON e.ComponentID = p.ProductID
        WHERE e.ProductAssemblyID = d.ComponentID
            AND e.EndDate IS NULL
        FOR XML PATH (N'item'), TYPE
    )
    FROM Production.BillofMaterials d
    INNER JOIN Production.Product p
        ON d.ComponentID = p.ProductID
    WHERE d.ProductAssemblyID = c.ComponentID
        AND d.EndDate IS NULL
    FOR XML PATH (N'item'), TYPE
)
FROM Production.BillofMaterials c
INNER JOIN Production.Product p
    ON c.ComponentID = p.ProductID
WHERE c.ProductAssemblyID = b.ComponentID
    AND c.EndDate IS NULL
FOR XML PATH (N'item'), TYPE
)
FROM Production.BillofMaterials b
INNER JOIN Production.Product p
    ON b.ComponentID = p.ProductID
WHERE b.ProductAssemblyID = a.ComponentID
    AND b.EndDate IS NULL
FOR XML PATH(N'item'), TYPE
)
FROM Production.BillofMaterials a
INNER JOIN Production.Product p
    ON a.ComponentID = p.ProductID
WHERE p.ProductID = @ProductID

```

```

AND a.EndDate IS NULL
FOR XML PATH(N'item'), ROOT(N'items'), TYPE
);

SELECT @x;
GO

DROP XML SCHEMA COLLECTION dbo.ComplexBOMSchema_Wildcard;

```

The `<any>` schema component declaration allows for elements in your XML document that do not normally belong, since they are not explicitly defined in the local XML schema. In the example given, the `<product-info>` element and its subelements were added to the XML document with a namespace different from the local default namespace. In the example, I added a new subquery (also highlighted in the listing) to retrieve additional descriptions for top-level components in all languages for which they are available. A partial result of the example is shown in Figure 4-5.

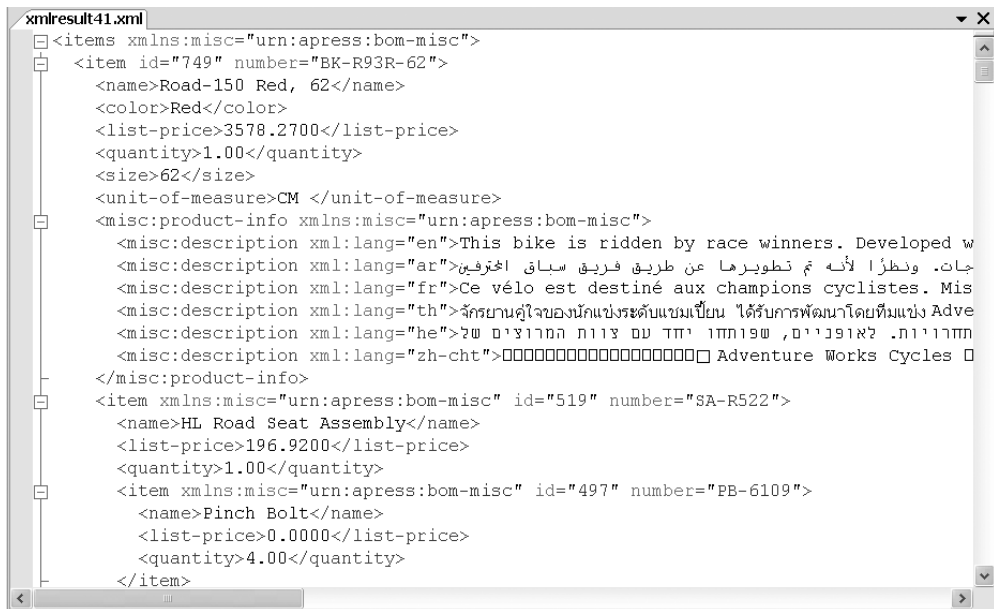


Figure 4-5. Result of XML schema with `<any>` schema component

UNIQUE PARTICLE ATTRIBUTION CONSTRAINTS

XML schemas have to meet a set of *unique particle attribution constraints*. These constraints are a set of rules specifying that XML schemas must be unambiguous. Unambiguous means the XML schema must be able to validate any given XML document in such a way that every element in the XML document maps to no more than one element defined in the XML schema. Additionally, XML schemas must be able to validate XML documents without any “look-ahead” algorithms. This makes sense when you consider that XML can be received as a stream, denying your applications access to look-ahead characters during transmission.

When you use the wildcard schema components `<any>` and `<anyAttribute>`, take special care to avoid ambiguity. Improper or incomplete attribute specifications on these schema components can cause errors that will keep SQL Server from registering XML schema collections based on your XSD.

The `<any>` schema component declaration in the sample XML schema is shown here:

```
<xsd:any processContents="lax" namespace="##other" minOccurs="0"
maxOccurs="unbounded" />
```

The `processContents` attribute of the `<any>` schema component specifies lax content processing, meaning the content is checked for well-formedness but only those items that can be validated against a schema are. Items that cannot be immediately validated against a schema are not validated. The other possible values for `processContents` include `strict` for strict validation of content and `skip`, which skips validation and simply checks for XML well-formedness. The default `processContents` value is `strict`.

Tip The `processContents` attribute cannot be set to `lax` on SQL Server 2005; support for this setting is new to SQL Server 2008. For SQL Server 2005, or XML schemas that must maintain backward-compatibility, use the `skip` setting instead.

The `namespace` attribute specifies the valid namespaces that may be used to prefix the elements in the `<any>` schema component. The value used in the example is `##other`, indicating that any namespace other than the default namespace for the document can be used. Other possible namespace values include `##any`, which stands for any namespace or a list of space-delimited namespace URI strings. The identifiers `##local` and `##targetNamespace` can be used in place of proper URI strings in the list.

In the sample, the namespace `##other` is specified to prevent ambiguity in the XML schema. If `##any` is used, for instance, SQL Server responds with an error message similar to the following result, indicating that the XML schema is ambiguous:

```
Msg 6916, Level 16, State 2, Line 1
XML Validation: The content model of type or model group
'xs:nun(/item/complexType())' is ambiguous and thus violates the unique
particle attribution constraint. Consult SQL Server Books Online for more
information.
```

Typing XML

XML schemas are used to validate and constrain XML document structure, but they also serve another very important function. XML schemas provide the ability to apply data type information to your XML documents. The XML Schema recommendation supports two basic classes of data types:

- Simple types define the *simple content* of an element or the value of an attribute.
- Complex types define the structure of an element, including valid attributes, content, and child nodes of an element.

Simple types include primitive types, which are built-in data types that cannot be derived from other types. These are the basic data types, like boolean, float, and decimal, many of which will be familiar to programmers. All simple types are considered to be derived from the base `anySimpleType` data type.

Derived built-in types are derived from the primitive types, including types like integer, unsignedLong, and token. You can type the BOM XML schema by adding data type information to its elements and attributes, as demonstrated in Listing 4-13.

Listing 4-13. Typed BOM Schema Collection

```
CREATE XML SCHEMA COLLECTION dbo.ComplexBOMSchema_Typed
AS
N'<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="items">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="item">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" minOccurs="1" maxOccurs="1" type="xsd:string" />
        <xsd:element name="color" minOccurs="0" maxOccurs="1" type="xsd:string" />
        <xsd:group ref="price-group" minOccurs="1" maxOccurs="1" />
        <xsd:element name="quantity" minOccurs="1" maxOccurs="1"
          type="xsd:decimal" />
        <xsd:group ref="size-group" minOccurs="0" maxOccurs="1" />
        <xsd:any processContents="lax" namespace="##other" minOccurs="0"
          maxOccurs="unbounded" />
        <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
      <xsd:attributeGroup ref="id-attr-group" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

    </xsd:complexType>
  </xsd:element>

  <xsd:attributeGroup name="id-attr-group">
    <xsd:attribute name="id" type="xsd:integer" />
    <xsd:attribute name="number" type="xsd:string" />
  </xsd:attributeGroup>

  <xsd:group name="price-group">
    <xsd:choice>
      <xsd:element name="list-price" minOccurs="1" maxOccurs="1"
        type="xsd:decimal" />
      <xsd:element name="standard-cost" minOccurs="1" maxOccurs="1"
        type="xsd:decimal" />
    </xsd:choice>
  </xsd:group>

  <xsd:group name="size-group">
    <xsd:sequence>
      <xsd:element name="size" minOccurs="1" maxOccurs="1"
        type="xsd:string" />
      <xsd:element name="unit-of-measure" minOccurs="1" maxOccurs="1"
        type="xsd:string" />
    </xsd:sequence>
  </xsd:group>

</xsd:schema>';
GO

```

The type attribute of the <element> and <attribute> schema components restricts the type of their values to the specified data type. The data types I used in the sample BOM XML schema are decimal (which includes all decimal numbers), integer (which includes all integer values—that is, numbers with no decimal places), and string (which encompasses all string values).

XSD also supports additional built-in data types, including familiar data types like boolean, float, double, date, time, and dateTime. There are also several additional self-explanatory data types, including types like positiveInteger, negativeInteger, and unsignedLong. XSD also supports other data types you might not immediately recognize, like normalizedString, which is a whitespace-normalized string value—that is, a string with no carriage returns (#xD;), line-feeds (#xA;), or tab (#x9;) characters.

Tip Appendix B provides a complete reference to the built-in XML Schema and XQuery Data Model data types available in SQL Server 2008.

Every data type is defined by three sets of items:

- A value space, which is the set of all distinct values that can be represented by the data type
- A lexical space, which is the set of valid literals, or character-based representations, for values
- A set of facets, which characterize properties of the data type's values

User-defined simple data types, based on the built-in simple data types, can be created with the `<simpleType>` schema component. User-defined simple data types can be derived by restriction, list, or union.

Complex types are user-defined types that describe elements with one or more child elements or attributes. Complex types are like definitions for XML document subtrees. You can also restrict or extend built-in types or user-defined types to create user-defined derived types.

To derive by restriction, you simply define a `simpleType` and apply the `<restriction>` schema component with applicable attributes to narrow down the type's value space, lexical space, or facets. You can use the `<restriction>` schema component to apply any of several validation criteria, including the following:

- Defining a pattern for validation via regular expression
- Setting minimum and maximum allowable values for the data type
- Specifying white-space handling
- Setting minimum and maximum length for allowable values
- Specifying other type-specific properties for your user-defined data type

Listing 4-14 builds on the BOM example to add user-defined simple types to validate the product ID and product number attributes.

Listing 4-14. *BOM Example with User-Defined Simple Types*

```
CREATE XML SCHEMA COLLECTION dbo.ComplexBOMSchema_UserTypes
AS
N'<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="items">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="item">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" minOccurs="1" maxOccurs="1"
          type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```

    <xsd:element name="color" minOccurs="0" maxOccurs="1"
      type="xsd:string" />
    <xsd:group ref="price-group" minOccurs="1" maxOccurs="1" />
    <xsd:element name="quantity" minOccurs="1" maxOccurs="1"
      type="xsd:decimal" />
    <xsd:group ref="size-group" minOccurs="0" maxOccurs="1" />
    <xsd:any processContents="lax" namespace="##other" minOccurs="0"
      maxOccurs="unbounded" />
    <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attributeGroup ref="id-attr-group" />
</xsd:complexType>
</xsd:element>

<xsd:simpleType name="product-number-type">
  <xsd:restriction base="xsd:string">
    <xsd:pattern
      value="[A-Z]{2}-[0-9A-Z][0-9]{2}[0-9A-Z](-([0-9]{2}|[BLMRSX]))?" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="product-id-type">
  <xsd:restriction base="xsd:nonNegativeInteger">
    <xsd:minInclusive value="1" />
    <xsd:maxInclusive value="999" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="unit-of-measure-type">
  <xsd:list itemType="xsd:normalizedString" />
</xsd:simpleType>

<xsd:attributeGroup name="id-attr-group">
  <xsd:attribute name="id" type="product-id-type" />
  <xsd:attribute name="number" type="product-number-type" />
</xsd:attributeGroup>

<xsd:group name="price-group">
  <xsd:choice>
    <xsd:element name="list-price" minOccurs="1" maxOccurs="1"
      type="xsd:decimal" />
    <xsd:element name="standard-cost" minOccurs="1" maxOccurs="1"
      type="xsd:decimal" />
  </xsd:choice>
</xsd:group>

<xsd:group name="size-group">
  <xsd:sequence>

```

```

    <xsd:element name="size" minOccurs="1" maxOccurs="1" type="xsd:string" />
    <xsd:element name="unit-of-measure" minOccurs="1" maxOccurs="1"
      type="xsd:string" />
  </xsd:sequence>
</xsd:group>

</xsd:schema>';
GO

```

The product-number-type user-defined type uses a `<pattern>` schema element with a regular expression to restrict the pattern of a string. The regular expression used for this purpose is as follows:

```
[A-Z]{2}-[0-9A-Z][0-9]{2}[0-9A-Z](-([0-9]{2}|[BLMRSX]))?
```

Tip In-depth coverage of regular expression syntax is beyond the scope of this book. There are many excellent books on the subject, however, including *Regular Expression Recipes for Windows Developers: A Problem-Solution Approach* by Nathan A. Good (Apress, 2005). If you plan to create user-defined types by restricting patterns, I highly recommend you pick up a book that details regular expression syntax for Windows, which differs in some respects from other platforms.

This regular expression ensures that only strings containing the defined characters in the specified order pass the validation test. The product-id-type user-defined data type uses the `minInclusive` and `maxInclusive` attributes to restrict the `nonNegativeInteger` data type to include only integer values between 1 and 999, inclusive.

While most user data types are simple types created by restriction, as I demonstrated previously in this section, you can also use XSD to define list and union data types. A union data type is one in which the value spaces and lexical spaces of one or more simple data types are unioned together. The example in Listing 4-15 demonstrates a simple union data type whose value space contains all nonzero integers. It achieves this by combining the lexical and value spaces of the `negativeInteger` and `positiveInteger` data types, both of which exclude the number zero from their value spaces.

Listing 4-15. Simple union Data Type Example

```

CREATE XML SCHEMA COLLECTION UnionSchemaCollection
AS
N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="nonZeroInteger">
    <xsd:simpleType>
      <xsd:union>
        <xsd:simpleType>
          <xsd:restriction base = "xsd:negativeInteger"/>
        </xsd:simpleType>
        <xsd:simpleType>
          <xsd:restriction base = "xsd:positiveInteger"/>

```

```

        </xsd:simpleType>
    </xsd:union>
</xsd:simpleType>
</xsd:element>
</xsd:schema>';
GO

```

```

DECLARE @x xml (UnionSchemaCollection);
SET @x = N'<nonZeroInteger>1</nonZeroInteger>';
SELECT @x;
GO

```

The list data type defines space-separated lists of simple data type items. Borrowing from the example in Listing 4-14 we can create a list data type that allows lists of AdventureWorks product numbers, as shown in Listing 4-16.

Listing 4-16. *Simple list Data Type Example*

```

CREATE XML SCHEMA COLLECTION ListSchemaCollection
AS
N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ProductNumberList">
    <xsd:simpleType>
      <xsd:list>
        <xsd:simpleType>
          <xsd:restriction base = "xsd:string">
            <xsd:pattern
              value = "[A-Z]{2}-[0-9A-Z][0-9]{2}[0-9A-Z](-([0-9]{2}|[BLMRSX]))?" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:list>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>';
GO

```

```

DECLARE @x xml (ListSchemaCollection);

SET @x = N'<ProductNumberList>RM-M823 FR-R92B-58
  CB-2903 FR-M94B-42</ProductNumberList>';

SELECT @x;
GO

```

In this section, I gave an overview of XML schema data types. I will cover these data types and others included in the XDM in greater depth in Chapter 5. I have also provided a quick reference to the XDM in Appendix B.

Summary

The W3C XML Schema recommendation defines a language for defining the structure of, and constraining the content of, XML data. XML Schema provides a more flexible and powerful method of validating your XML documents than the older method of using DTDs. In this chapter, I introduced XML schema collections and how they are used by SQL Server to validate the structure and content of your XML data, per the W3C XML Schema recommendation. Throughout this chapter, I used XSD to build a complex XML schema collection, which can be used to validate recursive nested BOMs in XML format.

Along the way, I demonstrated several key XML schema concepts, including the basics of how to define elements and attributes of an XML document. I also talked about defining model groups, constraining model and group occurrences, and using built-in data types to constrain XML content.

I also discussed avoiding ambiguity in XML schemas, documenting with annotations, and defining user-defined data types through restriction. In the next chapter, I will discuss the SQL Server 2008 implementation of the W3C XML Query (XQuery) language, including an in-depth discussion of XDM.



XQuery

The W3C defines the powerful XML Query (XQuery) language for querying XML data. SQL Server 2008 provides built-in support for XQuery via its `xml` data type methods. In this chapter, I will examine SQL Server's XQuery implementation and support, including keywords, syntax, and the XQuery/XPath Data Model (XDM). As always, I'll provide working examples to illustrate and highlight XQuery functionality along the way.

SQL Server implements XQuery through its `xml` data type methods: `query()`, `value()`, `nodes()`, `exists()`, and `modify()`. I introduced these methods in Chapter 3. I will begin this chapter with an introduction to the XQuery language and the XDM, finishing up with tips for maximizing SQL Server XQuery performance.

Introducing the XQuery Language

The W3C XQuery recommendation defines a declarative, typed, functional language for querying XML data. The XQuery language borrows heavily from several prior XML query languages including Quilt, XPath 1.0, XQL, and XML-QL, as well as database query languages like SQL and OQL. Technically XQuery 1.0 is considered an extension of XPath 2.0. These two languages have consistent language descriptions and grammars, they share a common data model (the XDM), and any valid XPath 2.0 expression is guaranteed to return the same result in XQuery.

XQuery's underlying data model, the XDM, defines an abstract model and type system that XQuery uses to query and perform operations on XML data. The W3C specification for the XDM extends the XML Schema data type system by adding additional data types of its own, including `untyped`, `untypedAtomic`, `anyAtomicType`, `dayTimeDuration`, and `yearMonthDuration`. SQL Server, however, does not support the `dayTimeDuration` and `yearMonthDuration` fully ordered subtypes of the duration data type. Using the XDM as the underlying data model, XQuery can query typed or untyped XML documents or fragments. An XDM instance can be logically viewed as a tree structure of your XML data with additional type information added. Figure 5-1 shows a simplified XDM instance for a short-typed XML document.

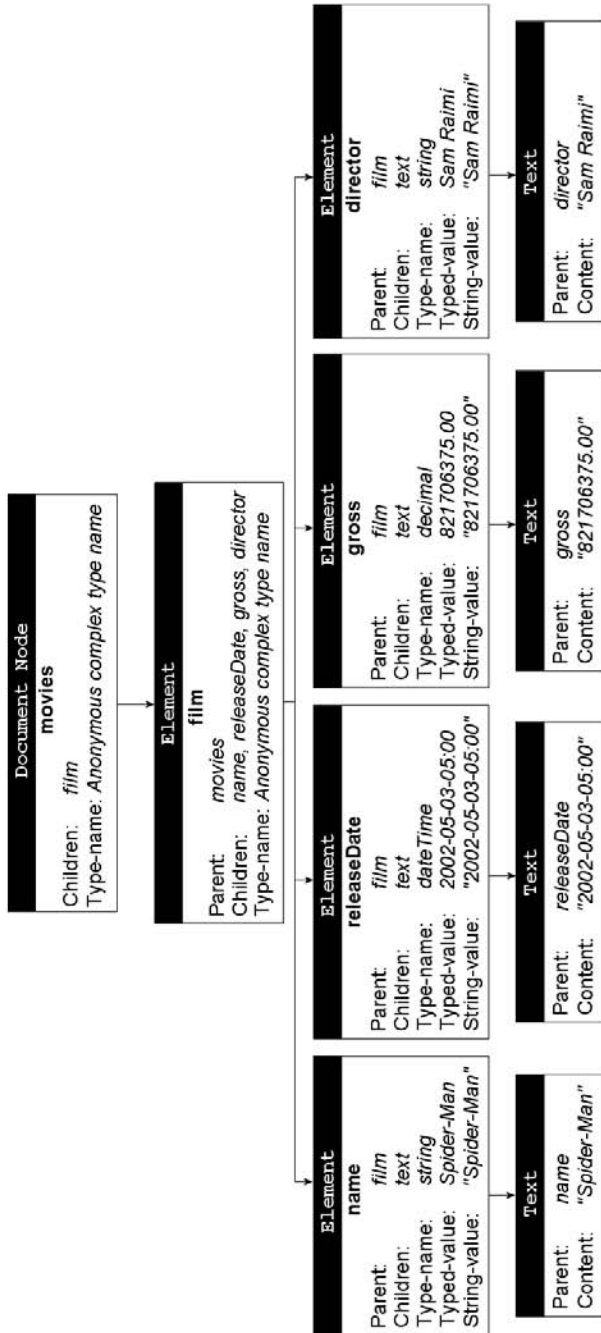


Figure 5-1. Simple XML document as an XDM instance

In the figure, each box represents an XDM node stored in a hierarchical structure with the root document node at the top. Each node is assigned XDM type and value information.

UNTYPED XML AND THE XDM

The term *untyped* can be a bit misleading when referring to SQL Server XQuery support and the XDM. Recall that a *typed xml* instance has been associated with an XML schema collection, while an untyped instance has not. When you store untyped xml, SQL Server converts it to an XDM instance with all nodes stored using the untyped data type and all attributes stored using the untypedAtomic data type. This is not as efficient as using typed xml instances, however. For best performance and static type-checking, the best practice is to associate your xml instances with an XML schema collection if you plan to query it via XQuery.

The XDM is designed to provide a powerful abstract representation of your XML data while providing facilities to support static typing and to make XQuery queries more efficient than querying raw textual XML documents. The XDM maps XML content to a hierarchical structure similar to the example shown in Figure 5-1. The XDM can map seven kinds of nodes:

- **Document nodes** encapsulate XML documents. Document nodes can contain only element, processing instruction, comment, and text nodes as children. Document nodes cannot have parent nodes. The document node is roughly analogous to the “root node” of XML. The main difference is that XQuery can operate on XML data that is composed of sequences of nodes that do not follow all the rules for well-formedness (e.g., no single root node). XML data that is not well-formed can have multiple document nodes.
- **Element nodes** represent XML elements. Element nodes, like document nodes, can contain element, processing instruction, comment, and text nodes as children. Unlike document nodes, an element node can have a parent node.
- **Attribute nodes** represent XML attributes. Attributes generally have an associated element node, although they can also have no associated element node and no parent node.
- **Namespace nodes** represent the namespace URI to namespace prefix/default namespace bindings.
- **Processing instruction nodes** represent XML processing instructions.
- **Comment nodes** represent XML comments.
- **Text nodes** represent XML character content.

The XDM also maps XML content to singleton atomic values. *Singleton atomic values* are indivisible values that fall into the value space of one of the basic XDM data types. Examples of atomic values include numeric constants like 3.141592 and string values like "George". The XDM type system, shown in Figure 5-2, builds on the XML Schema type system to provide a rich set of data types.

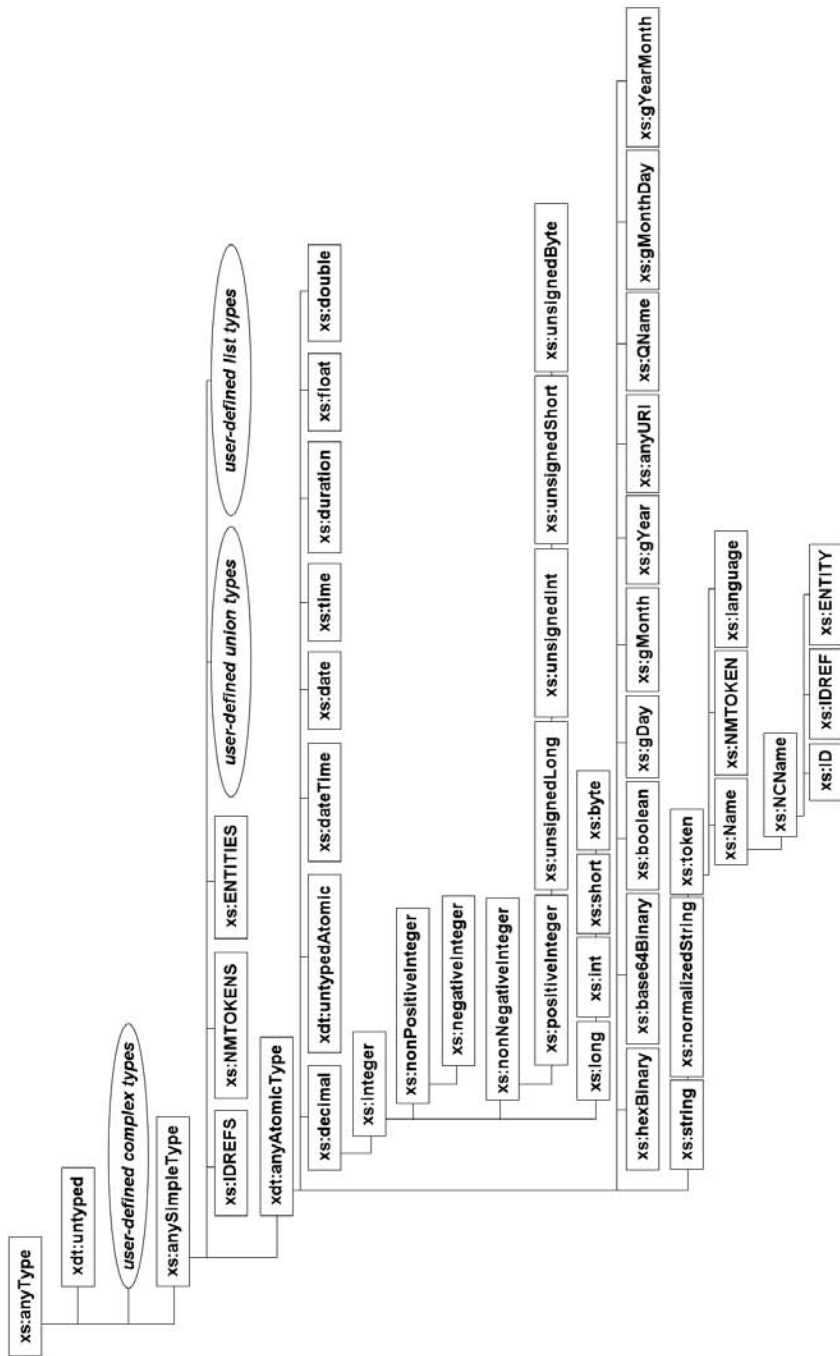


Figure 5-2. XDM data type system

As mentioned previously, SQL Server stores an abstract logical representation of your XML data in `xml` data type instances. In other words, SQL Server does not store the exact character-for-character string representation of an `xml` instance. Instead it converts the literal XML data it is given into an internal XDM-style format. The XDM provides a hierarchical structure that stores only the important information about XML nodes. Some of the metadata, however, is lost in the conversion process. Items that are lost include CDATA specifications and character and Document Type Definition (DTD) references, which are expanded during processing. Also, items such as numeric values might be stored in a nonliteral format. The numeric value 10.000 can be stored as the integer value 10 or a floating point value like 1E+1, depending on which data type it is assigned to. In these cases, the value and meaning of the numeric literal have not changed, but the exact representation has been altered for storage purposes.

NONLITERAL STORAGE

The W3C standard gives vendors a choice when they decide how to store numeric and other values. The basic idea is that the system can store a value in any format chosen by the vendor, so long as the meaning of the value is not changed. In other words, SQL Server is free to store atomic XQuery values in any format it wants, so long as the meaning is not altered. For instance, the numeric value 00001 can be stored as 1 or 1.0. Though these are different literal representations, they all refer to the exact same value, namely the value of the number one.

Creating XQuery Queries

XQuery queries are composed of *expressions*. The most basic expressions in XQuery are *primary expressions*, which include literals, variable references, function calls, constructors, and context item expressions. More complex expressions, like mathematical expressions, can be built from primary expressions. Listing 5-1 shows both types of expressions in XQuery.

Listing 5-1. Simple XQuery Queries

```
DECLARE @x XML;  
SET @x = '';  
SELECT @x.query('42'), @x.query('(2+3)*4');
```

A very simple primary expression, 42, is shown in the example. The slightly more complex mathematical expression $(2 + 3) * 4$ is built from literal values. I will discuss some of the other items that can be used as primary expressions, like function calls and constructors, in detail in the next chapter.

Another important concept in XQuery is the *sequence*. A sequence is an ordered collection of items: either nodes or atomic values. The word “ordered,” in this sense, does not necessarily mean alphabetical or numeric sort order, but rather *document order*. Because the order of nodes in XML data is important, XQuery returns nodes in document order—the order the nodes appear in your XML data—by default. Sequences have some special properties, which I describe in the “Sets and Sequences” sidebar.

SETS AND SEQUENCES

Order is an important property of sequences. In XPath 1.0, which was the predecessor to XPath 2.0 and XQuery 1.0, results were returned as node sets. XPath 1.0 node sets did not guarantee order. XQuery places new emphasis on node order in sequences, aligning it with the emphasis placed on element order by other XML recommendations. Another important difference between XPath 1.0 node sets and XQuery node sequences is that XPath 1.0 node sets do not allow duplicate nodes, while XQuery node sequences do.

Sequences in XQuery can be defined using parentheses and the *comma operator*. The comma operator evaluates each of its operands and concatenates the results into a sequence. A very simple sequence is shown in Listing 5-2, with the result shown in Figure 5-3.

Listing 5-2. Simple Sequence Example

```
DECLARE @x xml;  
SELECT @x = N'';  
SELECT @x.query('(3.141592, 1.414214, 1.0079)');
```

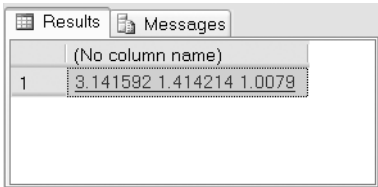


Figure 5-3. Result of simple sequence query

Empty sequences can be declared with the empty sequence notation: (). Sequences may contain duplicate values, although a sequence cannot contain other sequences. If a sequence declaration contains other sequences, those subsequences are “flattened out,” and empty sequences are removed. Listing 5-3 demonstrates a more complex sequence creation.

Listing 5-3. Sequence with Empty Sequences, Child Sequences, and Duplicate Values

```
DECLARE @x xml;  
SELECT @x = N'';  
SELECT @x.query('(1, 1, 2, (), 3, 4, 5, 10, (9, 8), 6, 7)');
```

This example demonstrates several aspects of sequence creation. There are duplicate values in this sequence, the sequence is created in document order (not numeric or alphabetical order), the empty sequence is removed completely, and the subsequence is flattened out into its component values. This is all shown in the result in Figure 5-4.

	(No column name)
1	1 1 2 3 4 5 10 9 8 8 6 7

Figure 5-4. Result of more complex sequence query

Another important property of sequences is that a sequence consisting of one singleton atomic value is equivalent to that singleton atomic value. This means that an XQuery comparison like `(4) eq 4` returns true.

While you are free to mix and match singleton atomic values of different data types in a single sequence, SQL Server does not allow you to mix singleton atomic values and nodes in a sequence. The following sequences are perfectly valid in the SQL Server XQuery implementation:

```
("One", 2, 3.0, "IV")
(<Currency>USD</Currency>, <Currency>EUR</Currency>, <Currency>GBP</Currency>)
```

But a heterogeneous sequence that combines nodes with singleton atomic values, like the following, will generate an error:

```
(<Four>4</Four>, 5.0, 6E+0, 7, "VIII")
```

Sequences are particularly important to XQuery, as they are the building blocks for XQuery results. XQuery results are always returned as sequences of nodes.

Cross-Platform Tip SQL Server XQuery does not support the W3C-defined union, intersect, and except sequence operators.

Defining the XQuery Prolog

Every XQuery query consists of two parts, the prolog and the body. The body of the query contains the path, FLWOR (for-let-where-order by-return), and other expressions. The prolog consists of a series of definitions and declarations that configure the static context for XQuery processing. The static context in XQuery is analogous to the preprocessing environment provided by many compiled languages, like C and C++. The prolog appears before the body, and each declaration ends with a semicolon (;) separator character.

SQL Server provides limited support for XQuery prologs, including the following declarations:

- The version declaration declares the XQuery version. The only currently valid version is "1.0", which is the default version if the version declaration is omitted. A version declaration takes the following format:

```
xquery version "1.0";
```

- The namespace declaration allows you to assign a namespace prefix to a URI. The namespace declaration looks like the following example:

```
declare namespace AWMI="http://tempuri.org/default";
```

- The default element namespace declaration defines the default namespace for unprefix elements and types in the XQuery query. This declaration looks like the following example:

```
declare default element namespace "http://tempuri.org/elements";
```

- The default function namespace declaration defines the default namespace for function names in function calls. These declarations look like the following example:

```
declare default function namespace "http://tempuri.org/functions";
```

The XQuery recommendation defines several additional prolog declarations, which allow variable and function declarations, module imports, and control various settings like white-space handling and collation. The current version of SQL Server does not support these declarations in the prolog.

XQUERY PREDEFINED NAMESPACES

For convenience, XQuery specifies some predefined namespaces. These namespaces include the following:

- `fn` is the XPath functions namespace with a URI of `http://www.w3.org/2005/xpath-functions`
- `xdtd` is the XPath/XQuery data types namespace with a URI of `http://www.w3.org/2003/05/xpath-datatypes`
- `xml` is the XML recommendation namespace with a URI of `http://www.w3.org/XML/1998/namespace`
- `xs` is the XML Schema namespace with a URI of `http://www.w3.org/2001/XMLSchema`
- `xsi` is the XML Schema instance with a URI of `http://www.w3.org/2001/XMLSchema-instance`

You don't need to declare these predefined namespaces yourself in the prolog to use them in your XQuery queries. If you choose to, however, you can redeclare the predefined namespaces yourself with the exceptions of the `xml` and `xmlns` namespaces. Another XQuery standard predefined namespace is the `local` namespace, which has a URI of `http://www.w3.org/2005/xquery-local-functions`. SQL Server does not predeclare this namespace, so you will need to define it yourself to use it.

Building Path Expressions

In addition to primary expressions, XQuery also supports *path expressions*. Path expressions are based on the XPath concept of a path expression. They are composed of *steps* that indicate the hierarchical path from which your XML data can be retrieved. The sample path expression shown in Listing 5-4 retrieves a sequence of nodes from sample geolocation data in XML format.

Listing 5-4. Simple Path Expression

```
DECLARE @x xml;

SELECT @x = N'
<Geocode-Results>
  <Result Name = "Microsoft Corp.">
    <Address>Microsoft Way</Address>
    <City>Redmond</City>
    <State>WA</State>
    <Zip>98052</Zip>
    <Latitude>47.643727</Latitude>
    <Longitude>-122.130474</Longitude>
  </Result>
  <Result Name = "Apple Inc.">
    <Address>1 Infinite Loop</Address>
    <City>Cupertino</City>
    <State>CA</State>
    <Zip>95014</Zip>
    <Latitude>37.332315</Latitude>
    <Longitude>-122.030749</Longitude>
  </Result>
</Geocode-Results>';

SELECT @x.query('/Geocode-Results/Result');
```

FROM XPATH 1.0 TO XQUERY

While XQuery syntax can be considered an extension to XPath 1.0, there are a number of significant differences between the two. One of the biggest advantages that XQuery provides over XPath 1.0 is the XDM type system. While XPath 1.0 only supports three basic data types (number, boolean, and string), XQuery supports over 50 built-in types in the extensive XDM type system.

XPath 1.0 and XQuery also differ in how results are returned. XPath 1.0 returns results in the form of *node sets*, which contain no duplicate nodes and assign no significance to ordering. XQuery, on the other hand, returns results as *node sequences*, which can contain duplicate nodes and are returned in a specified order (generally, results are returned in *document order* by default). The change from node sets to node sequences and this behavioral change in XQuery support the importance of node order in XML and XQuery.

XQuery also supports more types of comparisons than XPath 1.0, including general comparison operators, value comparison operators, node comparisons, and quantifiers. XQuery also supports *if. . . then. . . else* expressions, powerful FLWOR expressions, and a wide variety of built-in functions and operators I'll discuss later in this chapter.

The sample path expression `/Geocode-Results/Result` tells XQuery to perform the following query:

- The leading forward slash (`/`) tells the XQuery processor to go to the root of the XML document. The leading forward slash (`/`) technically indicates an implied root node, which is the parent of your XML data; this is particularly important for XML data that is not well-formed.
- The `Geocode-Results` step tells the XQuery processor to locate and visit each of the `Geocode-Results` node(s) located at the top level of your XML data and make them the *context node* in turn.
- The next forward slash (`/`) is a separator between steps.
- The final `Result` step tells the XQuery processor to locate all `Result` nodes below the `Geocode-Results` nodes and make them the context node in turn.
- Because there is no predicate included to limit the results of the path expression, XQuery will return a node sequence composed of all the context nodes that match the full path expression.

Tip The context node is a very important concept in XQuery. The context node is the node that is currently being processed by the XQuery processor. Path expression steps are indicated relative to the current context node. Another important concept is the *node test*. The node tests supported by SQL Server include both name node tests that match by name alone and node kind tests that match specific kinds of nodes.

The results of the sample XQuery query in Listing 5-4 are shown in Figure 5-5.

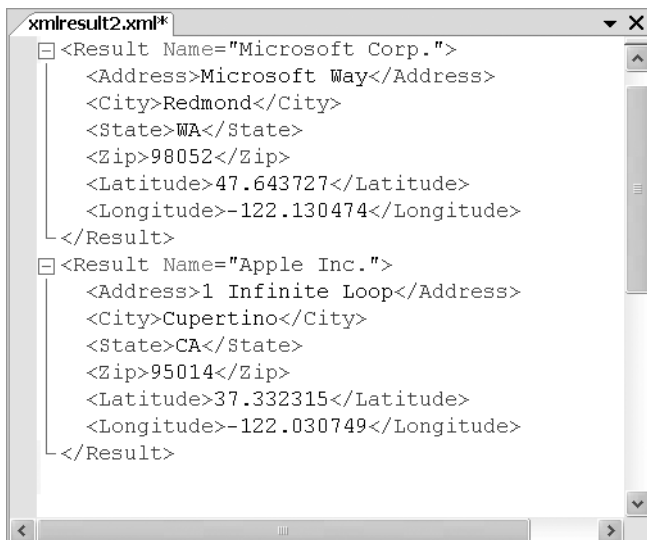


Figure 5-5. Results of simple XQuery path expression query

More complex path expressions can be created by using combinations of *axis steps*, which provide a “direction of movement” for each step in a path expression, node tests, and predicate filters to narrow down the results. The simple path expression given in Listing 5-4, and indeed most simple path expressions, can be read from left to right and have an implied direction of movement: next sibling and next child. If you indent your nested XML data as I’ve done in Listing 5-4, it’s easiest to visualize this implied movement as down and to the right. An axis step is a step indicating an explicit direction or axis as well as a node test. The explicit child axis step is used in Listing 5-5 to achieve the same results as the example in Listing 5-4.

Listing 5-5. *Using the child Axis Step*

```
DECLARE @x xml;

SELECT @x = N'
<Geocode-Results>
  <Result Name = "Microsoft Corp.">
    <Address>Microsoft Way</Address>
    <City>Redmond</City>
    <State>WA</State>
    <Zip>98052</Zip>
    <Latitude>47.643727</Latitude>
    <Longitude>-122.130474</Longitude>
  </Result>
  <Result Name = "Apple Inc.">
    <Address>1 Infinite Loop</Address>
    <City>Cupertino</City>
    <State>CA</State>
    <Zip>95014</Zip>
    <Latitude>37.332315</Latitude>
    <Longitude>-122.030749</Longitude>
  </Result>
</Geocode-Results>';

SELECT @x.query('/Geocode-Results/child::Result');
```

The axes supported by SQL Server 2008 are listed in Table 5-1.

Table 5-1. *SQL Server 2008 Support Axes*

Axis	Direction	Description
attribute	Forward	The attribute axis represents the attributes of the context node. The abbreviation for the attribute axis is the at sign (@).
child	Forward	The child axis navigates to the direct child node of the context node. The child axis is the default.
descendant	Forward	The descendant axis represents the transitive closure of the context node. The descendant axis contains the children of the context node, the children's children, and so on.
descendant-or-self	Forward	The descendant-or-self axis contains the context node and its descendants. The descendant-or-self axis can be abbreviated with the double forward slash (/).
parent	Reverse	The parent axis navigates to the parent of the context node. The parent axis can be abbreviated with two periods (..).
self	Forward	The self axis represents the current context node. The self axis can be abbreviated with a period (..).

The examples I've given you so far begin with a single slash (/) character, indicating the path should begin at the root of the XML data. The double slash (//) axis step can be used to start a path as well, indicating a match anywhere within the XML data. The path //Result returns the same result as the path /Geocode-Results/Result with the sample data in Listing 5-4. Additionally, you can use the // operator within a path, like /Geocode-Results//Result, to indicate a match anywhere below the matches returned by the first part of the path. Although the path expressions here all return the same result for the sample data from Listing 5-4, it's important to note that with different data these paths could return completely different results. For instance, if there were another level of Result nodes somewhere else in the document, the //Result path expression would return those nodes in addition to the nodes that match /Geocode-Results/Result that we are actually looking for.

Cross-Platform Tip The XQuery recommendation specifies an additional type of expression, known as a *range expression*. A range expression is a shorthand method of expressing a sequence that contains a range of scalar values. For instance, the range expression (1 to 5) expands out to (1, 2, 3, 4, 5). SQL Server XQuery does not implement range expressions, so keep this in mind if you are porting XQuery code from another platform.

Limiting Results with Predicates

In addition to axis steps, *predicates* can be added to filter results. Predicates are included in hard brackets ([]) in the path. A predicate can appear after any axis step, although they are often included at the very end. XQuery supports several different types of predicates and a full complement of comparison operators. XQuery uses predicates to filter results by keeping

results for which the predicate evaluates to true and discarding those for which the predicate is false.

The simplest predicates are numeric singleton atomic values, which return true if they are equal to the current context node position. The current context node position is analogous to an index number for each node, and numbering begins with 1. This is demonstrated in Listing 5-6, which shows modifications to the XQuery query in Listing 5-4 to filter using numeric predicates to limit the results to nodes at specific context positions.

Listing 5-6. *Filtering Using Numeric Predicates*

```
DECLARE @x xml;

SELECT @x = N'
<Geocode-Results>
  <Result Name = "Microsoft Corp.">
    <Address>Microsoft Way</Address>
    <City>Redmond</City>
    <State>WA</State>
    <Zip>98052</Zip>
    <Latitude>47.643727</Latitude>
    <Longitude>-122.130474</Longitude>
  </Result>
  <Result Name = "Apple Inc.">
    <Address>1 Infinite Loop</Address>
    <City>Cupertino</City>
    <State>CA</State>
    <Zip>95014</Zip>
    <Latitude>37.332315</Latitude>
    <Longitude>-122.030749</Longitude>
  </Result>
</Geocode-Results>';

SELECT @x.query('/Geocode-Results/Result[2]');
SELECT @x.query('/Geocode-Results[1]/Result[2]');
SELECT @x.query('(//Geocode-Results/Result)[2]');
```

Note The [1] numeric predicate after a path expression is by far the most common numeric predicate, particularly when a singleton atomic value or a single node is required. The [1] numeric predicate returns the single node at context position 1, ensuring that only a single node or singleton atomic value is ever returned. This is very important for static type checking to work—even if you are sure a single node will be returned because only one node matches your path expression. XQuery takes a more pessimistic, less data-centric view. XQuery processors are required to assume your data may change, and that those changes may cause the same query to return multiple nodes in the future. Because of this, XQuery demands that you use [1] in many cases to ensure that only a single node is ever returned.

All three of the queries in Listing 5-6 return the same result, shown in Figure 5-6.

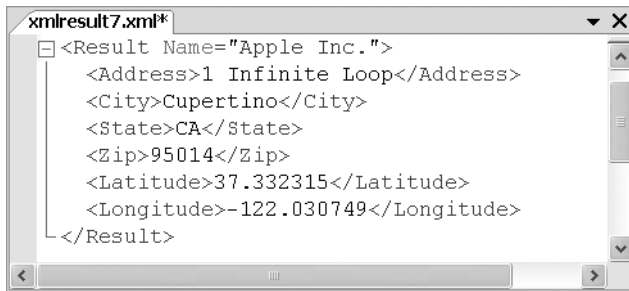


Figure 5-6. Result of XQuery queries with numeric predicates

Although all three of the queries in Listing 5-6 return the same results for the sample data, there are differences in how they achieve that result. The first path expression, shown in the following code line, retrieves the second Result node from every Geocode-Results parent node match:

```
/Geocode-Results/Result[2]
```

The second path expression in the example retrieves the second Result node from only the first Geocode-Results parent node match, shown in the following path expression:

```
/Geocode-Results[1]/Result[2]
```

The third path expression retrieves the second match from all nodes that match the /Geocode-Results/Result path, shown in the following path expression:

```
(/Geocode-Results/Result)[2]
```

Despite returning the same results in the example, the differences in behavior and results between the three path expressions can be profound with different sample data, if there were two top-level Geocode-Results nodes, for instance.

STATIC ANALYSIS

The W3C recommends that XQuery query processors implement a mode of evaluation known as *static analysis*. Static analysis is used to validate XQuery expressions before execution. The static analysis phase of XQuery query processing produces pre-runtime metadata known as the *static context*. The static context is used to capture several errors prior to the execution phase, including various type mismatches, use of unsupported features, XML schema validation errors, and various other exceptions.

One of the most common static errors you are likely to encounter is when a query that returns a singleton value is expected, but a query that could return multiple nodes is encountered. The `value()` method, for example, expects a query that returns a singleton value. XQuery uses a very conservative mode of static analysis known as *pessimistic validation*. One of the results of this model is that if a query could conceivably, under any circumstances, return multiple nodes where a query that returns a singleton value is expected, a static error is raised. The actual data in the target XML instance that is being queried is not considered

during the static analysis phase. If you have a typed `xml` instance, however, the associated XML schema collection is used during static analysis to validate your XQuery queries.

The XQuery static analysis phase can be a little daunting when you first start working with XQuery. The pessimistic nature of XQuery prevents automatic type conversions that programmers also take for granted in other languages. An `xs:string` value of "9.4" will not be automatically converted to a numeric type in a math expression, for instance. You have to explicitly cast values between types, with a few exceptions.

The error messages are also sometimes a bit cryptic. For instance, "XQuery [value()]: 'value()' requires a singleton (or empty sequence), found operand of type 'xdt:untypedAtomic *'" could simply mean you forgot to follow the query with a `[1]` predicate, or it could be a sign of a more serious flaw in your query.

Once your query passes the static analysis phase, additional error checking is provided at query runtime by the dynamic analysis phase. The dynamic analysis phase produces metadata containing a wide range of runtime information, known as the *dynamic context*. This "belt-and-suspenders" approach quickly eliminates a lot of errors. Unfortunately there's little that can be done by the XQuery processor to resolve logic errors, which also happen to be the hardest type of error to troubleshoot (e.g., a path expression that evaluates to the wrong node).

You can create more complex XQuery predicates using the built-in comparison operators. XQuery includes three types of comparison operators, shown in Figure 5-7.

Value Comparison Operators		General Comparison Operators	
<code>eq</code>	Equal to	<code>=</code>	Equal to
<code>ne</code>	Not equal to	<code>!=</code>	Not equal to
<code>gt</code>	Greater than	<code>></code>	Greater than
<code>ge</code>	Greater than or equal to	<code>>=</code>	Greater than or equal to
<code>lt</code>	Less than	<code><</code>	Less than
<code>le</code>	Less than or equal to	<code><=</code>	Less than or equal to
Node Comparison Operators			
<code>is</code>	Node identity equality		
<code>>></code>	Left node precedes right node		
<code><<</code>	Left node follows right node		

Figure 5-7. XQuery comparison operators

The XQuery *value comparison operators* act like basic scalar comparison operators in other languages, comparing singleton atomic values to one another. Listing 5-7 modifies Listing 5-4 to demonstrate the use of a value comparison operator that limits the result to only those `Result` nodes with a `Name` attribute equal to "Apple Inc.".

Listing 5-7. *Value Comparison Operator Example*

```

DECLARE @x xml;

SELECT @x = N'
<Geocode-Results>
  <Result Name = "Microsoft Corp.">
    <Address>Microsoft Way</Address>
    <City>Redmond</City>
    <State>WA</State>
    <Zip>98052</Zip>
    <Latitude>47.643727</Latitude>
    <Longitude>-122.130474</Longitude>
  </Result>
  <Result Name = "Apple Inc.">
    <Address>1 Infinite Loop</Address>
    <City>Cupertino</City>
    <State>CA</State>
    <Zip>95014</Zip>
    <Latitude>37.332315</Latitude>
    <Longitude>-122.030749</Longitude>
  </Result>
</Geocode-Results>';

SELECT @x.query('/Geocode-Results/Result[@Name eq "Apple Inc."]);

```

XQuery *general comparison operators* are described as “existential” operators that are used to compare sequences. The general comparison operators return true if any of the singleton atomic values from the sequence on the left-hand side of the operator fulfills the operator comparison with any of the singleton atomic values from the right-hand sequence. In other words, the following general comparison returns true:

$$(1, 2, 3) = (3, 4, 5)$$

This general comparison is true because the number 3 appears in both the left-hand and right-hand sequences, and 3 equals 3 is true. However, the following general comparison returns false because there are no singleton atomic values from the left-hand sequence that fulfill the less-than comparison with any singleton atomic values from the right-hand sequence.

$$(3, 4, 5) < (1, 2, 3)$$

Finally, the node comparison operators compare two nodes to determine if a node precedes another node (the << operator) or follows another node (the >> operator) in document order. The `is` node comparison operator checks for node identity equality. XQuery assigns every node a unique identifier; a node identity equality comparison checks that the two nodes being compared are the same node. It's important to note that two nodes with the same structure and content will return false when compared using `is` if both nodes are not the exact same node with the same unique identifier. In other words, the `is` operator does not compare node structure and content, but rather it returns true when the two nodes being compared are the exact same node. Listing 5-8 adds two node comparisons to Listing 5-4 to demonstrate these operators.

Listing 5-8. *Node Comparison Operator Example*

```

DECLARE @x xml;

SELECT @x = N'
<Geocode-Results>
  <Result Name = "Microsoft Corp.">
    <Address>Microsoft Way</Address>
    <City>Redmond</City>
    <State>WA</State>
    <Zip>98052</Zip>
    <Latitude>47.643727</Latitude>
    <Longitude>-122.130474</Longitude>
  </Result>
  <Result Name = "Apple Inc.">
    <Address>1 Infinite Loop</Address>
    <City>Cupertino</City>
    <State>CA</State>
    <Zip>95014</Zip>
    <Latitude>37.332315</Latitude>
    <Longitude>-122.030749</Longitude>
  </Result>
</Geocode-Results>';

SELECT @x.query('/Geocode-Results/Result[@Name = "Apple Inc."])[1]
  >> (/Geocode-Results/Result[@Name eq "Microsoft Corp."])[1]');

SELECT @x.query('/Geocode-Results/Result[@Name = "Apple Inc."])[1]
  is (/Geocode-Results/Result[@Name eq "Microsoft Corp."])[1]');

```

The first query uses the >> node comparison operator to compare the document order of two Result nodes. The first query returns true, since the first node with its Name attribute set to "Apple Inc." appears after the first node with its Name attribute set to "Microsoft Corp." in document order. The second query returns false, since the two nodes compared are not one and the same node.

XQuery supports the and and or keywords to combine multiple logical expressions in a compound predicate. XQuery also supports the fn:not() function to return the inverse of a logical expression. Listing 5-9 modifies Listing 5-4 to demonstrate a compound predicate in a path expression.

Listing 5-9. *Compound Predicate Example*

```

DECLARE @x xml;

SELECT @x = N'
<Geocode-Results>
  <Result Name = "Microsoft Corp.">
    <Address>Microsoft Way</Address>

```

```

    <City>Redmond</City>
    <State>WA</State>
    <Zip>98052</Zip>
    <Latitude>47.643727</Latitude>
    <Longitude>-122.130474</Longitude>
  </Result>
  <Result Name = "Apple Inc.">
    <Address>1 Infinite Loop</Address>
    <City>Cupertino</City>
    <State>CA</State>
    <Zip>95014</Zip>
    <Latitude>37.332315</Latitude>
    <Longitude>-122.030749</Longitude>
  </Result>
</Geocode-Results>;

```

```

SELECT @x.query('/Geocode-Results/Result[fn:not(@Name eq "Apple Inc.")
and @Name eq "Microsoft Corp."]);

```

This compound predicate tests matching nodes to ensure that the Name attribute is not equal to "Apple Inc.", but it also tests that the name is equal to "Microsoft Corp.".

Tip The `fn` namespace is a predeclared namespace that contains definitions for several useful XQuery functions. I will discuss the `fn` namespace in more detail in Chapter 6.

Using Quantified Expressions

XQuery supports the *quantified expressions* *every* and *some*. The quantified expressions accept one or more in clauses that bind variables to values. The quantified expressions also have a *satisfies* clause that contains an expression. The *every* expression is a *universal quantifier*, meaning all of the bound values must satisfy the condition for the quantified expression to return true. The *some* expression is an *existential quantifier* returning true if any of the bound values satisfy the *satisfies* clause. Listing 5-10 demonstrates simple use of quantified expressions.

Listing 5-10. Quantified Expression Examples

```

DECLARE @x xml;

SET @x = '';

SELECT @x.query ('some $x in (1, 2, 3)
  satisfies $x * $x = 9');

SELECT @x.query ('every $x in (1, 2, 3)
  satisfies $x * $x = 9');

```

In both queries, the variable `$x` is bound to each of the values 1, 2, and 3 in turn. The `satisfies` clause contains the expression `$x * $x = 9`. For the `some` expression to return `true`, at least one of the values in the sequence must evaluate to `true` in the `satisfies` clause. For the `every` expression to return `true`, each and every value of `$x` must return `true`. In this example, the `some` clause returns `true`, and the `every` clause returns `false`.

DOCUMENTATION, DOCUMENTATION, DOCUMENTATION

Most organizations have policies concerning documentation in computer programs generated internally or by third parties (when source code is supplied). And all modern programming languages provide some facility for adding inline documentation to your source code. XQuery is no different, and it supplies the so-called “emoticon remark” symbols that delimit inline documentation. The `(: and :)` delimiters denote the beginning and end of a block of XQuery comments. The XQuery processor ignores anything between these symbols. XQuery comments are not accessible outside of these delimiters, and they cause no side effects within your queries. Although I have intentionally left inline documentation out of this book in an effort to pack as much content as possible into the limited number of pages allowed, I highly recommend making extensive use of this feature in your production code.

Using FLWOR Expressions

XQuery offers a powerful query expression mechanism known as FLWOR expressions. FLWOR is an acronym for the XQuery keywords `for`, `let`, `where`, `order by`, and `return`. The FLWOR expression syntax serves as an approximation for the SQL `SELECT` query, and SQL programmers will find it familiar. A FLWOR expression requires, at minimum, a `for` and `return` clause, as shown in Listing 5-11.

Listing 5-11. Simple FLWOR Expression

```
SELECT Resume.query('declare default element namespace
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
    for $i in (/Resume/Employment)
    return ($i/Emp.JobTitle)')
FROM HumanResources.JobCandidate
WHERE JobCandidateId = 1;
```

Tip The `Resume` column of the `HumanResources.JobCandidate` table declares an element namespace with a URI of `http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume`. The `declare default element namespace` statement is used to declare this namespace for the query in Listing 5-11.

The AdventureWorks sample database has an xml data type Resume column in the HumanResources.JobCandidate table. This query uses this sample AdventureWorks data to retrieve the previous job titles of the job candidate with Id 1. The for clause in this instance generates a *tuple stream*. A tuple is a binding of a variable to a sequence of values/nodes, and a tuple stream is a sequence of tuples that will each be considered in turn. For this simple example, the tuple stream consists of the /Resume/Employment nodes of the XML data bound to the \$i variable. The return clause returns the Emp.JobTitle child node of each tuple. The results are shown in Figure 5-8.

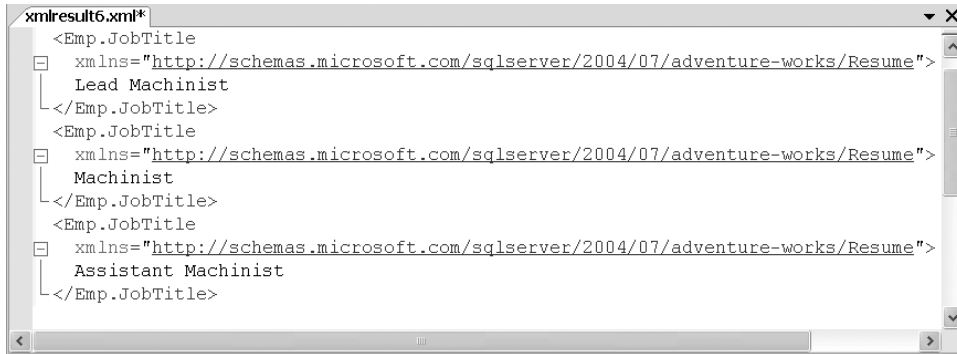


Figure 5-8. Results of simple FLWOR expression

The let clause of the FLWOR expression binds values to variables, but without performing iterations like the for clause. This feature was not available in SQL Server 2005, but it is implemented in SQL Server 2008. Listing 5-12 uses the let clause to assign the Employment node of each Resume node to the \$j variable. Notice that the let clause uses the := symbol to assign/bind tuple streams to variables. The return clause then returns the Emp.JobTitle child node of the \$j variable. The end result of this query is the same as the result of Listing 5-11.

Listing 5-12. FLWOR with let Clause

```
SELECT Resume.query('declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
for $i in (/Resume)
let $j := $i/Employment
return ($j/Emp.JobTitle)')
FROM HumanResources.JobCandidate
WHERE JobCandidateId = 1;
```

Caution The variable scope rules laid out by the W3C specify that variables declared in a for clause can be redeclared in a let clause. Essentially, doing this creates a new instance of the variable but prevents later FLWOR expression clauses from accessing the original variable. Be careful not to accidentally redeclare a variable in a let clause.

The where clause is analogous to the SQL WHERE clause. Like the SQL WHERE clause, the XQuery where clause keeps nodes for which the expression evaluates to true and eliminates nodes that evaluate to false. Listing 5-13 keeps only nodes from the previous results where the Emp.JobTitle is not equal to "Machinist".

Listing 5-13. *FLWOR with where Clause*

```
SELECT Resume.query('declare default element namespace
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
    for $i in (/Resume/Employment)
    where ($i/Emp.JobTitle ne "Machinist")
    return ($i/Emp.JobTitle)')
FROM HumanResources.JobCandidate
WHERE JobCandidateId = 1;
```

The order by clause is analogous to the SQL ORDER BY clause. It is used to reorder tuples in a tuple stream generated by the for and let clauses. The order by clause accepts one or more order specifications, each of which consists of an expression and an ordering modifier (ascending or descending). Listing 5-14 modifies Listing 5-12 to order the results by job title in descending order.

Listing 5-14. *FLWOR with order by Clause*

```
SELECT Resume.query('declare default element namespace
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
    for $i in (/Resume/Employment)
    let $j := ($i/Emp.JobTitle)
    order by fn:string($j) descending
    return ($j)')
FROM HumanResources.JobCandidate
WHERE JobCandidateId = 1;
```

Because the order by clause requires singleton instances, I've used the fn:string function to order by the string value of each Emp.JobTitle.

Cross-Platform Tip SQL Server XQuery does not support the `ordered` and `unordered` keywords, which set the ordering mode for an expression in a static context. The workaround is to convert expressions that use `ordered` and `unordered` to FLWOR expressions and use the `order by` clause to explicitly order the results.

CROSS-PRODUCTS

XQuery can produce cross-products by putting two or more `in` expressions side by side, separated by commas. The following example demonstrates a simple cross-product generation with a FLWOR expression.

```
DECLARE @x xml;
SET @x = '';
SELECT @x.query('for $i in (1, 2, 3), $j in (4, 5, 6)
  return ($i * $j)');
```

This simple result binds the variable `$i` to the values 1, 2, 3, and the variable `$j` to the values 4, 5, 6. Because the two `in` expressions binding the variables are separated by a comma, XQuery generates the cross-product. To finish the example, the `return` clause returns the value of `($i * $j)` for each pair generated in the cross-product. Following is the resulting sequence.

```
4 5 6 8 10 12 12 15 18
```

The cross-product generation feature is the XQuery equivalent of the SQL `CROSS JOIN`. Like the `CROSS JOIN`, it's not particularly useful in most cases unless it is used with a `where` clause, in which case this construct behaves similarly to the SQL `INNER JOIN`.

Constructing XML with XQuery

XQuery offers powerful expressive node construction functionality. XQuery offers both *direct* and *computed element constructors*. A direct element constructor is just what the name implies—XQuery constructs XML directly from your specification in standard XML notation. A simple example of an XQuery direct element constructor is shown in Listing 5-15.

Listing 5-15. Simple XQuery Direct Element Constructor

```
DECLARE @x XML;
SET @x = N'';
SELECT @x.query ('
<Measurement>
  One dozen is equal to { 5 + 7 }
</Measurement>');
```

This simple example demonstrates a direct element constructor with an *enclosed expression*. The enclosed expression is enclosed with curly braces, indicating to XQuery that the contents should be evaluated as an expression. In this example, the expression `5 + 7` is evaluated during the element construction to produce the result shown in Figure 5-9.

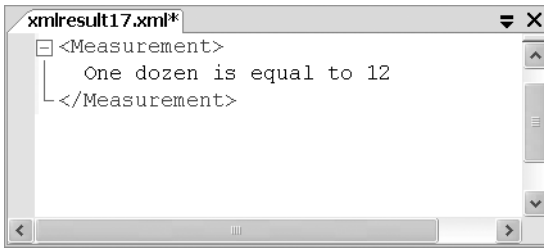


Figure 5-9. Simple direct element construction

The true power of direct element construction is its ability to use more complex enclosed expressions to create complex XML. Listing 5-16 retrieves the education level information (degrees and graduation dates) for an AdventureWorks job applicant and uses it to demonstrate complex direct XML construction with a FLWOR expression.

Listing 5-16. *FLWOR Expression Direct Element Construction*

```
SELECT Resume.query ('declare namespace
    ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
<Education.History>
{
  for $i in (/ns:Resume/ns:Education)
  return
  (
    <Level>
      <Degree>
        { fn:data($i/ns:Edu.Level) }
      </Degree>
      <Date>
        { fn:data($i/ns:Edu.EndDate) }
      </Date>
    </Level>
  )
}
</Education.History>')
FROM HumanResources.JobCandidate
WHERE JobCandidateId = 2;
```

Tip If you want to include curly braces in your XML output, you can escape them by doubling them up like this: {{ and }}. The escaped braces are converted to literal single braces in your constructed XML.

The result of this more complex direct element constructor including a FLWOR expression is shown in Figure 5-10.

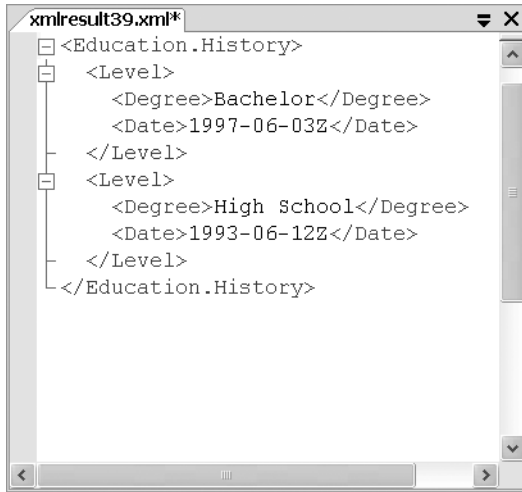


Figure 5-10. *Result of complex direct element constructor*

Direct element construction is a powerful tool, but XQuery also supports computed element constructors. Computed element constructors consist of a node type keyword, a name (except when constructing a text node), and an expression that generates the node content. SQL Server XQuery supports the element, attribute, and text node type keywords to construct the specified node types. Listing 5-17 expands the sample in Listing 5-16 to construct XML using computed element constructors.

Cross-Platform Tip SQL Server XQuery does not support the document node type keyword. You can use the element keyword to create a top-level node as a workaround to this limitation.

Listing 5-17. *Computed Element Constructors Example*

```
SELECT Resume.query ('declare namespace
ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
for $i in (/ns:Resume/ns:Education)
return (
  element Education.History
  {
    element Level
    {
      attribute School { fn:data($i/ns:Edu.School) },
      element Degree { fn:data($i/ns:Edu.Level) },
      element Date { fn:data($i/ns:Edu.EndDate) }
```



```

    }
  }
)')
FROM HumanResources.JobCandidate
WHERE JobCandidateId = 2;

```

The result of this computed element construction query is shown in Figure 5-11.

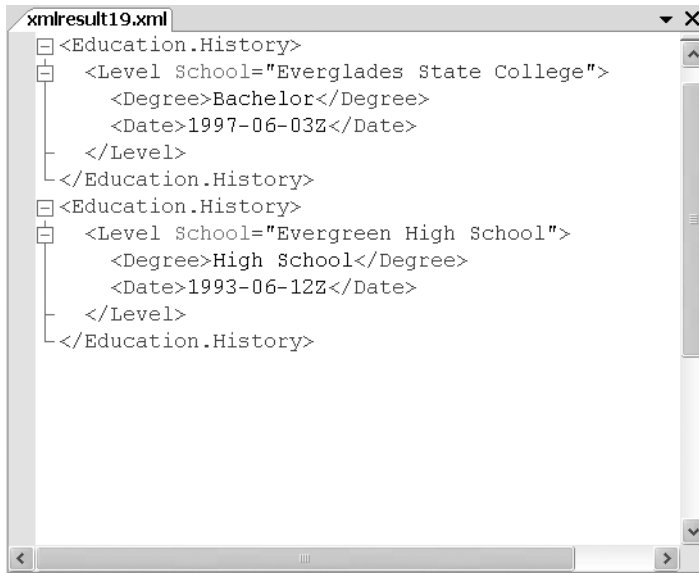


Figure 5-11. *Computed element construction result*

Though XML construction is a powerful and useful XQuery feature, it is not as efficient as the FOR XML clause when converting large amounts of relational data to XML format. If your goal is to convert a lot of relational data to XML, consider using SELECT with the FOR XML clause instead.

Using the SQL Server xml Methods

The SQL Server xml data type has four methods that utilize XQuery. So far I've covered several examples of the query() method, so you should be fairly familiar with it at this point. In this section, I'll describe each of the four xml data type methods in greater detail and explore their inner workings.

Querying with query()

The xml data type query() method is the primary method for querying your XML data. The query() method accepts an XQuery query as a parameter and returns a sequence of nodes as an untyped xml instance. The examples so far in this chapter have all used the query() method to perform queries against xml data type instances and to evaluate stand-alone XQuery expressions.

Tip Using the `query()` method on an `xml` variable containing an empty string is a useful method for quickly evaluating stand-alone XQuery expressions and performing quick one-off tests. You’ve seen this method used several times in the book so far, and I will use it again. Be sure to assign an empty string to your `xml` data type instance, since an XQuery query against a `NULL` `xml` instance always returns `NULL`.

I’ve covered several examples that highlight the functionality of the `query()` method in this chapter, and you’ll see many more before we’re done, so I will forego yet another example just to demonstrate the `query()` method here. Instead, I’ll focus on the underlying operations performed by the XQuery methods like `query()`.

The SQL Server team had to make several design decisions when adding the `xml` data type and XQuery support to SQL Server. One of the most important decisions they made was to leverage the power of the existing SQL Server engine. Using the `query()` method generates a Table Valued Function [XML Reader with XPath filter] operation in your query plan. This is easily verified by viewing the query plan for a query that uses the `query()` method. The Table Valued Function [XML Reader with XPath filter] operation shreds your input XML into a relational format that can be used by the SQL Server engine to fulfill your query. Figure 5-12 shows the Table Valued Function [XML Reader with XPath filter] operation produced by a query plan.

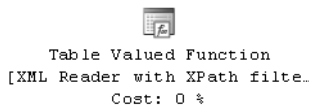


Figure 5-12. Table Valued Function [XML Reader with XPath filter] operation

A simplified example of the relational data produced by this operator is shown in Figure 5-13.

ORDPATH	TAG	NODE	NODETYPE	VALUE	PATH_ID
1	1 (Resume)	Element	12 (ResumeT)	null	#Resume
1.1	2 (Name)	Element	13 (NameT)	null	#Name#Resume
1.1.1	3 (Prefix)	Element	2 (xs:string)		#Prefix#Name#Resume
1.1.3	4 (First)	Element	2 (xs:string)	Shai	#First#Name#Resume
1.1.5	5 (Middle)	Element	2 (xs:string)		#Middle#Name#Resume
1.1.7	6 (Last)	Element	2 (xs:string)	Bassli	#Last#Name#Resume
1.1.9	7 (Suffix)	Element	2 (xs:string)		#Suffix#Name#Resume

Figure 5-13. Relational data produced by [XML Reader with XPath filter]

Once the XML data is shredded into relational form, the SQL Server engine can query and manipulate it just like any other relational data. The cost for dynamically shredding many rows of XML data stored in a table to fulfill a query can be high, however. Creating a primary XML index on an xml data type column serializes a preshredded representation of your XML data, eliminating the cost of shredding at query time. The tradeoff, of course, is greater storage requirements for the XML index. We will discuss XML indexes in detail in Chapter 7.

Retrieving Scalar Values with value()

The value() method retrieves a scalar value from your xml data type instance and casts it to a SQL Server data type instance. The value() method is particularly useful when you need to retrieve a singleton atomic value from your xml data type instance or for shredding your xml instances when used in conjunction with the nodes() method. I'll discuss the shredding aspects in the "Shredding XML with nodes()" section of this chapter.

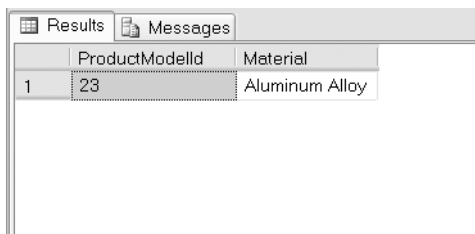
The value() method requires two parameters, an XQuery query that returns a singleton atomic value and a SQL Server data type. The SQL Server data type name must be passed in as a string. Listing 5-18 uses the value() method to retrieve the Material specification for AdventureWorks product model #23.

Listing 5-18. Using the value() Method

```
SELECT ProductModelId, CatalogDescription.value ('declare namespace ns =
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/"
ProductModelDescription";
    (/ns:ProductDescription/ns:Specifications/Material/text())[1]',
    'VARCHAR(100)')
AS Material
FROM Production.ProductModel
WHERE ProductModelId = 23;
```

Tip It's important that the XQuery query passed to the value() method return a singleton atomic value. If there is any possibility that the query can return more than one value, a static type error is thrown.

The result of this simple value() method query is shown in Figure 5-14.



	ProductModelId	Material
1	23	Aluminum Alloy

Figure 5-14. Result of value() method

Checking for Node Existence with exist()

The `exist()` method checks for the existence of a node and returns a bit value indicating whether the node exists or not. The `exist()` method accepts a single XQuery query as a parameter and returns a result based on the return value of the query. If the query returns SQL NULL, the `exist()` method returns NULL; if the query returns an empty sequence, the `exist()` method returns 0; and if the query returns a sequence of one or more nodes, the `exist()` method returns 1.

The sample query in Listing 5-19 uses the `exist()` method to query the `CatalogDescription` xml column of the `ProductModel` table. The query returns information for all product models that are made from an alloy material.

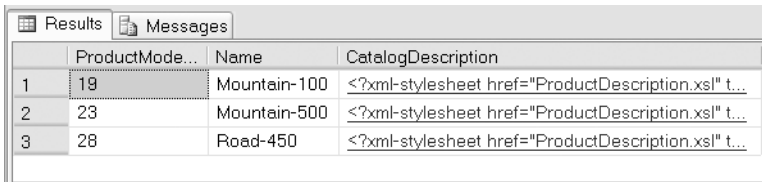
Listing 5-19. Using `exist()` Method

```
DECLARE @material VARCHAR(50);

SET @material = 'Alloy';

SELECT pm.ProductModelId, pm.Name, pm.CatalogDescription
FROM Production.ProductModel pm
WHERE pm.CatalogDescription.exist('declare namespace ns =
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
    /ns:ProductDescription/ns:Specifications/Material
    [fn:contains( . , sql:variable("@material") ) ]') = 1;
```

Three rows meet the criteria, as shown in Figure 5-15.



	ProductModelId	Name	CatalogDescription
1	19	Mountain-100	<?xml-stylesheet href="ProductDescription.xsl" t...
2	23	Mountain-500	<?xml-stylesheet href="ProductDescription.xsl" t...
3	28	Road-450	<?xml-stylesheet href="ProductDescription.xsl" t...

Figure 5-15. Results of `exist()` method sample

The sample XQuery in Listing 5-19 makes use of the XQuery `fn:contains` function and the `sql:variable` function. The `fn:contains` function is roughly analogous to the T-SQL `CHARINDEX` function. It accepts two strings as parameters and returns true if the second string exists within the first string; otherwise, it returns false. The `sql:variable` function passes the value of a T-SQL variable into your XQuery query. In this case, the T-SQL variable `@material` contains the string `Alloy`. I will discuss the `fn:contains` and `sql:variable` functions in greater detail in Chapter 6.

Shredding XML with `nodes()`

Traditionally, the main method for shredding XML (converting XML data to relational format) has been the `OPENXML` rowset provider. The `xml` data type now provides a much more elegant solution with its `nodes()` method. The `nodes()` method is very flexible and can be used with

the other xml data type methods to give you very fine-grained control over the shredding process. Listing 5-20 demonstrates using the `nodes()` method to query the XML query plans in the SQL Server cache.

Listing 5-20. *Querying the Cached XML Query Plans*

```
WITH Plans(nodeid, physicalop, estimated_cost, plan_handle,
           text, query_plan, cacheobjtype, objtype)
AS
(
    SELECT RelOp.op.value('declare default element namespace
        "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
        @NodeId', 'int'),
        RelOp.op.value('declare default element namespace
        "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
        @PhysicalOp', 'varchar(50)'),
        RelOp.op.value('declare default element namespace
        "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
        @EstimatedTotalSubtreeCost ', 'float'),
        cp.plan_handle,
        st.text,
        qp.query_plan,
        cp.cacheobjtype,
        cp.objtype
    FROM sys.dm_exec_cached_plans cp
    CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
    CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle) qp
    CROSS APPLY qp.query_plan.nodes('declare default element namespace
        "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
        //RelOp') RelOp (op)
)
SELECT ROW_NUMBER() OVER (PARTITION BY p.plan_handle ORDER BY p.NodeId)
    AS Operation_Num,
    p.physicalop,
    p.text,
    p.cacheobjtype,
    p.objtype,
    p.estimated_cost
FROM Plans p
WHERE p.cacheobjtype = 'Compiled Plan';
```

The core of this sample is the Common Table Expression (CTE) that uses execution-related dynamic management views and functions to retrieve the query cache information. The CTE retrieves all plan handles from the `sys.dm_exec_cached_plans` dynamic management view. It uses `CROSS APPLY` to pass the plan handles into the `sys.dm_exec_sql_text` and `sys.dm_exec_query_plan` dynamic management functions. Finally it uses `CROSS APPLY` to shred the XML query plan retrieved with the `nodes()` method.

```

FROM sys.dm_exec_cached_plans cp
CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle) qp
CROSS APPLY qp.query_plan.nodes('declare default element namespace
    "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
    //RelOp') RelOp (op)

```

Note The XML namespace URI for SQL Server XML query plans is <http://schemas.microsoft.com/sqlserver/2004/07/showplan>. The actual XML schema for query plans is available at this URL on the Web if you want to view it.

The `nodes()` method specifies the XQuery query `//RelOp`, which retrieves all nodes named `RelOp` anywhere they occur in the XML query plan. The first step of the `nodes()` shredding process is to return each `RelOp` node as a single row in the `op` column of a virtual table named `RelOp`.

Tip You specify the virtual table name and column in the format *virtual_table (column)* as an alias after the `nodes()` method call. You can see this in the sample code where I've specified a virtual table name of `RelOp` and a column name of `op`.

The second step in the shredding process is to pick apart the XML in the rows returned by the `nodes()` method. You can do this with the `query()` or `value()` nodes methods, in the `SELECT` clause of the CTE from Listing 5-20, as shown in the following.

```

SELECT RelOp.op.value('declare default element namespace
    "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
    @NodeId', 'int'),
    RelOp.op.value('declare default element namespace
    "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
    @PhysicalOp', 'varchar(50)'),
    RelOp.op.value('declare default element namespace
    "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
    @EstimatedTotalSubtreeCost ', 'float'),

```

In this example, the `value()` method is used to retrieve the values of the `@NodeId`, `@PhysicalOp`, and `@EstimatedTotalSubtreeCost` attributes of each `RelOp` node. The final result is shown in Figure 5-16.

	Operat...	physicalop	text	cacheobjtype	objtype	estimated_c...
1	1	Sequence Project	WITH Plans(nodeid, physicalop, estimated_cost, p...	Compiled Plan	Adhoc	686.416
2	2	Segment	WITH Plans(nodeid, physicalop, estimated_cost, p...	Compiled Plan	Adhoc	686.416
3	3	Compute Scalar	WITH Plans(nodeid, physicalop, estimated_cost, p...	Compiled Plan	Adhoc	686.416
4	4	Sort	WITH Plans(nodeid, physicalop, estimated_cost, p...	Compiled Plan	Adhoc	686.416
5	5	Nested Loops	WITH Plans(nodeid, physicalop, estimated_cost, p...	Compiled Plan	Adhoc	686.402
6	6	Nested Loops	WITH Plans(nodeid, physicalop, estimated_cost, p...	Compiled Plan	Adhoc	458.364
7	7	Nested Loops	WITH Plans(nodeid, physicalop, estimated_cost, p...	Compiled Plan	Adhoc	230.327
8	8	Nested Loops	WITH Plans(nodeid, physicalop, estimated_cost, p...	Compiled Plan	Adhoc	2.28983
9	9	Nested Loops	WITH Plans(nodeid, physicalop, estimated_cost, p...	Compiled Plan	Adhoc	2.85267E-05
10	10	Nested Loops	WITH Plans(nodeid, physicalop, estimated_cost, p...	Compiled Plan	Adhoc	1.65618E-05

Figure 5-16. Query execution plan shredding with `nodes()`

Tip You can use the `query()`, `value()`, `exist()`, and `nodes()` methods on the rows returned in the virtual table by the `nodes()` method. You can also use the `COUNT(*)` aggregate function and the `IS NULL` predicate. Trying to use other functions or methods on the rows returned by `nodes()` is not allowed.

XML query plans are a wealth of information. You can easily build on the example in Listing 5-20 to retrieve all types of interesting information from XML query plans.

Manipulating XML with `modify()`

The SQL Server `xml` data type supports extensions to XQuery known as XML Data Manipulation Language (XML DML). The `modify()` method updates an XML instance per your specifications. You can use the `modify()` method and XML DML to delete, insert, or replace nodes in your `xml` data type instances. The sample in Listing 5-21 uses the XML DML `replace value of` statement to change the genre of an album listed in an XML document from “Rap” to “Anime & Manga.”

Listing 5-21. XML DML `modify()` Method Sample

```
DECLARE @x xml;

SET @x = N'<?xml version = "1.0"?>
<music>
  <album name = "Fullmetal Alchemist Complete Best">
    <genre>Rap</genre>
    <song artist="L&apos;arc-en-Ciel">Ready Steady Go</song>
    <song artist="Nana Kitade">Indelible Sin</song>
    <song artist="Yellow Generation">To the Other Side of the Door</song>
  </album>
</music>';
```

```

SET @x.modify ('replace value of
  (/music/album[ @name = "Fullmetal Alchemist Complete Best" ]/genre/text())[1]
with
  "Anime & Manga"');

SELECT @x;

```

The example `replace value of` statement takes an XQuery path to identify the node to modify. The `with` clause takes the value to replace the existing node value with. The result of this sample is shown in Figure 5-17.

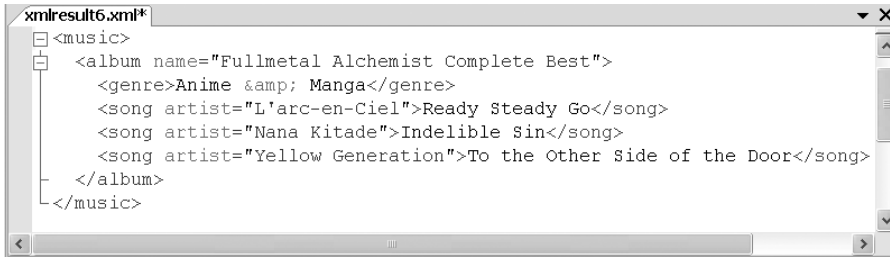


Figure 5-17. XML DML *replace value of* statement sample result

This rounds out the discussion of `xml` data type methods. I will discuss XML DML in greater detail in Chapter 6.

Conditional Evaluation with `if...then...else`

XQuery supports conditional evaluation with an `if...then...else` construct. The `if...then...else` keywords in XQuery operate differently from most procedural languages. In fact, the XQuery `if...then...else` construct is analogous to the SQL `CASE...WHEN...ELSE` expression. Listing 5-22 demonstrates a simple use of `if...then...else`.

Listing 5-22. Sample *if...then* Query

```

DECLARE @x xml;

SET @x = '';

SELECT @x.query ('if (100 lt 200)
  then "100 is less than 200"
  else "100 is not less than 200"');

```

In this example, constants are used for clarity, but they could easily be replaced with path expressions or variables. In this example, the expression in the `if` clause is executed.

```
if (100 lt 200)
```

If the expression returns true, the expression following the `then` clause is returned.

```
then "100 is less than 200"
```


If the expression returns false, the expression following the else clause is returned.

```
else "100 is not less than 200"
```

This operation is exactly like the SQL CASE expression. Unlike the SQL equivalent, however, the else clause in XQuery's if. . .then. . .else is mandatory.

Maximizing XQuery Performance

As I've discussed previously, proper use of XML indexes and XML schema collections will help optimize your XQuery performance on SQL Server. In addition, there are additional steps you can take when writing XQuery queries to help maximize SQL Server XQuery performance. In this section, I'll present some tips to help you get the most out of your SQL Server XQuery queries.

Use the value() Method

The first tip is to use the xml data type value() method to cast singleton atomic values from XML to SQL Server data types and avoid using the SQL Server CAST or CONVERT functions with the query() method. Remember, the query() method returns an untyped xml instance. So the CAST method requires you to first cast the query() result to a varchar or nvarchar string, even if you want to ultimately convert to a numeric or other noncharacter type. You also lose any potential benefits that come with typing your xml instance by associating an XML schema collection to it. Listing 5-23 demonstrates both the efficient and inefficient methods of querying XML for singleton atomic values.

Listing 5-23. *Singleton Atomic Value Query Comparison*

```
/* Inefficient method */
WITH XMLNAMESPACES (DEFAULT
    'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/►
    ProductModelManuInstructions')
SELECT CAST(
    CAST(
        Instructions.query(N'fn:data(/root/Location/@LaborHours)[1]')
        AS varchar(10))
    AS numeric(4, 1))
FROM Production.ProductModel
WHERE Instructions IS NOT NULL;

/* Efficient method */
WITH XMLNAMESPACES (DEFAULT
    'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/►
    ProductModelManuInstructions')
SELECT Instructions.value(N'fn:data(/root/Location/@LaborHours)[1]',
    N'numeric(4, 1)')
FROM Production.ProductModel
WHERE Instructions IS NOT NULL;
```

Avoid Reverse Axis Steps

The double period (..) reverse axis step is a costly axis step that can really hurt your overall XQuery performance. Avoiding .. in your XQuery path expressions has the additional benefit of making your path expressions easier to read, debug, and maintain. It is usually a trivial matter to rewrite path expressions to remove reverse axis steps, but be careful to ensure that the rewritten path expression returns the same results as the original. Consider the example in Listing 5-24, which retrieves all AdventureWorks product assembly steps that require a tool. This is indicated by a presence of a tool node within a step node of the XML instructions.

In the inefficient version, you first search for all tool nodes and use the reverse axis step to return each tool node's parent step node. In the efficient version, you search for all step nodes and then check to see if they contain a tool node. The second version is about three times more efficient than the reverse axis step version.

Listing 5-24. *Reverse Axis Step Comparison*

```
/* Inefficient version */
WITH XMLNAMESPACES (DEFAULT
    'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/►
        ProductModelManuInstructions')
SELECT Instructions.query('(/tool/..') AS ToolStep
FROM Production.ProductModel
WHERE Instructions IS NOT NULL;

/* Efficient version */
WITH XMLNAMESPACES (DEFAULT
    'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/►
        ProductModelManuInstructions')
SELECT Instructions.query('(/step[fn:not(fn:empty(./tool))])') AS ToolStep
FROM Production.ProductModel
WHERE Instructions IS NOT NULL;
```

Avoid // and Wildcards in the Middle

The descendant-or-self axis step, abbreviated with forward slashes (/), is extremely useful when you want a certain node or a bunch of matching nodes but cannot guarantee the exact path to those nodes. I've made use of this axis step in several examples so far. To get the most efficiency from descendant-or-self, use a secondary XML index, and don't use this axis step with wildcards in the middle of a path expression. Listing 5-25 creates a secondary PATH XML index on the Production.ProductModel table and then demonstrates the difference between using // with the * wildcard in the middle of a path expression and using // at the beginning of the path with no wildcards. The second query shown is almost twice as efficient as the first.

Listing 5-25. *descendant-or-self Axis and Wildcard in the Middle Comparison*

```
/* Build the secondary PATH XML Index first */
CREATE XML INDEX SXML_ProductModel_Path
ON Production.ProductModel (Instructions)
```

```

USING XML INDEX PXML_ProductModel_Instructions
FOR PATH;

/* Inefficient version */
WITH XMLNAMESPACES (DEFAULT
    'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➡
        ProductModelManuInstructions')
SELECT Instructions.query('/root/Location/**/material') AS Material
FROM Production.ProductModel
WHERE Instructions IS NOT NULL;

/* Efficient version */
WITH XMLNAMESPACES (DEFAULT
    'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➡
        ProductModelManuInstructions')
SELECT Instructions.query('(/material)') AS Material
FROM Production.ProductModel
WHERE Instructions IS NOT NULL;

```

If you play around with this sample, you might notice that using either the descendant-or-self axis or the wildcard in the middle of a query alone degrades performance to slightly over the efficient version of the query shown, but neither option hurts performance as much as when both are used together in the same path expression.

Use Subqueries

Whenever possible don't force SQL Server to re-evaluate your XQuery queries over and over again. This degrades performance considerably. Instead, execute your XQuery queries in SELECT statement subqueries for maximum efficiency. Listing 5-26 demonstrates both methods.

Listing 5-26. XQuery in Subquery Comparison

```

/* Inefficient method */
WITH XMLNAMESPACES (DEFAULT
    'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➡
        ProductModelManuInstructions')
SELECT CASE
    WHEN Instructions.value('fn:data(/root/Location/@LaborHours)[1]',
        N'numeric(4, 1)') IN (0.5, 1.0) THEN 'Easy'
    WHEN Instructions.value('fn:data(/root/Location/@LaborHours)[1]',
        N'numeric(4, 1)') IN (1.5, 2.0) THEN 'Medium'
    WHEN Instructions.value('fn:data(/root/Location/@LaborHours)[1]',
        N'numeric(4, 1)') IN (2.5, 3.0) THEN 'Hard'
    END AS TaskLevel
FROM Production.ProductModel
WHERE Instructions IS NOT NULL;

```

```

/* Efficient method */
WITH XMLNAMESPACES (DEFAULT
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
ProductModelManuInstructions')
SELECT CASE
    WHEN Labor.LaborHours IN (0.5, 1.0) THEN 'Easy'
    WHEN Labor.LaborHours IN (1.5, 2.0) THEN 'Medium'
    WHEN Labor.LaborHours IN (2.5, 3.0) THEN 'Hard'
END AS TaskLevel
FROM (
    SELECT Instructions.value('fn:data(/root/Location/@LaborHours)[1]',
        N'numeric(4, 1)') AS LaborHours
    FROM Production.ProductModel
    WHERE Instructions IS NOT NULL
) AS Labor;

```

Avoid Predicates in the Middle

Putting comparison predicates in the middle of path expressions, known as *branching*, can cause a significant performance hit. Instead, use the more powerful features of XQuery, like FLWOR expressions, when you need to insert additional comparison logic in the middle of an XQuery expression. The XQuery optimizer can do a better job of optimizing FLWOR expressions than it can for a branched path expression. Listing 5-27 demonstrates the difference between a branched path expression and a FLWOR expression. The FLWOR expression is about twice as efficient as the branched path.

Listing 5-27. Branched Path Expression Comparison

```

/* Inefficient method */
WITH XMLNAMESPACES (DEFAULT
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
ProductModelManuInstructions')
SELECT Instructions.query('/root/Location[@SetupHours eq 0.1]/step')
    AS Step
FROM Production.ProductModel
WHERE Instructions IS NOT NULL;

/* Efficient method */
WITH XMLNAMESPACES (DEFAULT
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
ProductModelManuInstructions')
SELECT Instructions.query('for $i in (/root/Location)
    let $j := $i/@SetupHours
    where $j eq 0.1
    return $i/step') AS Step
FROM Production.ProductModel
WHERE Instructions IS NOT NULL;

```

This also applies to numeric predicates like `[1]`. When possible, apply numeric predicates to an entire query instead of each individual step. So when possible avoid path expressions like the following:

```
/root[1]/Location[1]/step[1]
```

Instead, use path expressions like the following, which can be evaluated more efficiently:

```
(/root/Location/step)[1]
```

As always, when revising path expressions make sure that the revision returns the same results as the original path expression, based on your data.

Summary

I covered a lot of ground in this chapter. The XQuery recommendation describes a powerful and expressive XML query language. The SQL Server implementation of XQuery, though lacking some features, is still very powerful. The `xml` data type is feature-rich, with methods to query XML data, retrieve singleton scalar values, check for the existence of nodes, and shred XML data.

In this chapter, I discussed the XDM, the rich XQuery data model and type system, and the internal logical representation of XML using the XDM. I also explored XQuery queries, which provide a wide array of expressions, including primary expressions, path expressions, quantified expressions, `if. . .then. . .else` expressions, comparison expressions, and the powerful FLWOR expression syntax. Being an expressive, declarative, functional language, the W3C spared no expense in providing several different types of expressions to query XML.

The most important things to take away from this chapter are that XQuery is a powerful query language that supports intuitive basic path expressions, and that while it provides several advanced features like XML construction, XQuery is not at all complicated. It is very easy for SQL developers to grab the concepts in XQuery, but it can be a bit harder for people coming from a strictly procedural programming background (e.g., VB, C#, C++).

Finally, I discussed some of the things that you can do when writing your XQuery queries to ensure that they run as efficiently as possible. Keep in mind that these tips are specific to the SQL Server XQuery implementation, and they should not be applied broadly to other XQuery implementations.

The next chapter will continue the discussion of XQuery with detailed descriptions of XML DML and XQuery functions and operators.



XQuery Functions and Operators and XML DML

In Chapter 5, I discussed the W3C XML Query language (XQuery) standard and the SQL Server XQuery implementation in detail. In addition to its powerful support for expressions, XQuery provides support for several built-in functions and operators. SQL Server XQuery also provides XML Data Manipulation Language (XML DML) extensions for modifying XML data programmatically. In this chapter, I'll discuss support for W3C XQuery standard functions, XQuery operators, SQL Server extension functions, and XML DML in SQL Server.

PREDECLARED NAMESPACES

SQL Server 2008 includes several predeclared XML namespaces. These are useful because you can reference them automatically without the need to declare them yourself. These namespaces are as follows:

- `fn`: the XPath/XQuery functions namespace
- `sql`: the SQL Server extensions namespace
- `sqltypes`: the SQL Server to base type mapping sequence
- `xdm`: the XQuery 1.0/XPath 2.0 data types
- `xml`: the default XML namespace
- `xmlns`: the XML Namespaces declaration namespace
- `xs`: the XML Schema namespace (often declared as `xsd` in the W3C recommendation)
- `xsi`: the XML Schema instance namespace

You've encountered some of these predeclared namespaces in previous chapters and will encounter more of them later, such as the `fn` and `sql` namespaces I will deal with in this chapter. Note that the `local` namespace, specified by the W3C recommendation, is not predeclared in SQL Server 2008.

Using Operators

SQL Server XQuery supports several W3C-specified operators. The operators include mathematical operators, comparison operators, and logical operators. I will discuss the XQuery operators supported by SQL Server in this section.

Calculating with Math Operators

The standard math operators available in SQL Server XQuery are listed in Table 6-1.

Table 6-1. *XQuery Math Operators*

Operator	Description
+	Addition, Unary Plus
-	Subtraction, Unary Minus
*	Multiplication
div	Division
mod	Modulo/Remainder of Division

Math operators can be used in XQuery expressions to perform calculations, as in the following examples:

```
1 + 10
(2.0 * 36.0) div 3.0
(1e+4 * 8) * (2 + 6)
```

Unary plus and minus have the highest precedence, followed by multiplication, division, and the modulo operator. Binary addition and subtraction have the lowest priority of the math operators. Of course, parentheses can be used to group expressions and change the order of evaluation, as in the previous examples.

Cross-Platform Tip The only standard XQuery math operator not implemented by SQL Server is the `idiv` (integer division) operator. This operator is equivalent to casting the result of the standard `div` operator to the `xs:integer` type, like this: `(4.3 div 2.0) cast as xs:integer?`.

Using Comparison Operators

XQuery comparison operators fall into three categories: general comparison, value comparison, and node comparison operators. The XQuery comparison operators are listed in Table 6-2.

Table 6-2. *XQuery Comparison Operators*

Type of Operators	Operators
General Comparison	>, <, =, >=, <=, !=
Value Comparison	gt, lt, eq, ge, le, ne
Node Comparison	>>, <<, is

The logical operators include `and` and `or`. As discussed in Chapter 5, these operators allow you to create compound predicates like the following examples:

```
$x gt 20.0 and $x lt 30.0
$i/@PhysicalOp eq "Nested Loops" or $i/@PhysicalOp eq "Compute Scalar"
```

See Chapter 5 for a detailed discussion of the comparison and logical operators.

Constructing Sequences with the Comma Operator

The comma operator is used in XQuery to construct a sequence, like the following example:

```
(10, ("IX", 8), 7, 6, (), "five", 4, 3, 2.0, 1e+0)
```

The comma operator evaluates each of its operands and results in a sequence like the following:

```
10 IX 8 7 6 five 4 3 2 1
```

As mentioned in Chapter 5, empty sequences contained in a sequence are eliminated, and other sequences contained within a sequence are flattened out to their component parts. Also, while sequences can be composed of nodes or singleton atomic values of different types like the preceding example, SQL Server does not allow *heterogeneous sequences* that mix nodes and singleton atomic values in a single sequence.

Multiple sequences can be combined with the comma operator into a single sequence, as demonstrated in the following example:

```
(1, 2, 3), ("four", "five", "six")
```

The result of combining the two sequences is the single sequence, as the following shows:

```
1 2 3 four five six
```

Tip The comma can also be used to create a cross-product of two sequences when binding variables in a FLWOR (for-let-where-order by-return) expression, also discussed in Chapter 5.

Using XQuery Type Expressions

XQuery includes several *type expressions*, which allow casting of expressions from one type to another, and testing whether a value is an instance of a given type. These expressions are discussed in this section.

Casting XQuery Values

Because XQuery is strongly typed, there may be a need to cast values from one type to another. It is not uncommon to cast `xs:string` values to other types, for instance. XQuery provides the cast expression to convert existing values to a specific data type. The cast expression accepts an input expression and a target type, as shown in Listing 6-1.

Listing 6-1. Simple cast Expression Sample

```
DECLARE @x xml;  
SET @x = '';  
SELECT @x.query(' "123.4567" cast as xs:decimal? ');
```

The *occurrence indicator* (?) following the cast expression in Listing 6-1 indicates that, if the input expression is an empty sequence, the result of the cast is an empty sequence. SQL Server does not support cast expressions without the occurrence indicator. The input expression for a cast expression must be a singleton atomic value, and the target type must be a simple type.

Checking the Instance Type

The `instance of` expression can tell you whether or not a singleton expression is an instance of a given type. The `instance of` expression takes an input expression and a simple or an element type, and returns true if the expression is an instance of the type specified.

Cross-Platform Tip The SQL Server XQuery `instance of` operator does not accept sequences or sequence types, as specified by the XQuery recommendation. According to the recommendation, an `instance of` expression like the following should return true:

```
(1, 2, 3) instance of xs:integer*
```

In the XQuery recommendation, `xs:integer*` represents a sequence of integers. SQL Server XQuery supports only singleton instances, however, and does not support the `*` and `+` occurrence indicators. Keep this in mind if you are porting XQuery code to SQL Server from another platform.

The instance of expression is demonstrated in Listing 6-2.

Listing 6-2. *Using instance of Expressions*

```
DECLARE @x xml;
SET @x = N'<Countries>
  <Country>United States of America</Country>
  <Country>Canada</Country>
  <Country>Mexico</Country>
</Countries>';
SELECT @x.query('123 instance of xs:integer?');
SELECT @x.query('"abc" instance of xs:string?');
SELECT @x.query('(/Countries/Country)[1]
  instance of element(Country, xdt:untyped?)');
```

All these instance of expressions evaluate to true. 123 is an instance of `xs:integer`, abc is an instance of `xs:string`, and the expression `(/Countries/Country)[1]` is an instance of `element(Country, xdt:untyped?)`. The first two examples are self-explanatory, but the third expression deserves a little more discussion. Unlike the `cast as` expression, `instance of` can accept a singleton node instance. In this case, the first instance of `/Countries/Country` is returned. The instance of operator in this example checks if the node returned is an element named `Country` of type `xdt:untyped` (note that the `?` occurrence indicator is required here).

In addition to the element node type, `instance of` can recognize several types of nodes, including those listed in Table 6-3.

Table 6-3. *Node Types*

Node Type	Description
node	Returns true if the expression evaluates to any type of node
item	Returns true if the expression evaluates to an item
element	Returns true if the expression evaluates to an element node
text	Returns true if the expression evaluates to a text node
comment	Returns true if the expression evaluates to a comment node
attribute	Returns true if the expression evaluates to an attribute node
processing-instruction	Returns true if the expression evaluates to a processing instruction node

UNIMPLEMENTED XQUERY EXPRESSIONS

There are other expressions that are defined by the XQuery recommendation that are not implemented in SQL Server 2008 XQuery. These expressions include the following:

- The `castable as` expression, which returns true if an expression is able to be cast to a given type
- The `typeswitch` expression, which chooses one of several expressions based on the dynamic type of an expression
- The `treat as` expression, which provides additional information during the static analysis phase that a given expression should be treated as a particular type
- The `validate` expression, which supports validation of the results of expressions

Perhaps these expressions will be implemented in a future version of SQL Server, but for now any code that requires these expressions must be changed when ported to SQL Server XQuery.

Using XQuery Functions

The W3C defines several built-in functions for XQuery, and SQL Server adds a few of its own to support direct T-SQL-to-XQuery interaction. In this section, I'll discuss the standard XQuery and other functions supported by SQL Server. The built-in XQuery functions are declared in the predefined `fn` namespace, and the SQL Server-specific functions are declared in the `sql` namespace.

Using Data Accessor Functions

XQuery provides accessor functions to access properties of XQuery/XPath Data Model (XDM) instance items. SQL Server XQuery supports two data accessor functions, `fn:string()` and `fn:data()`.

The `fn:string()` function accepts a singleton value expression and returns the value of its argument as an `xs:string`. If the expression passed into `fn:string()` is a node, it returns the `xs:string` value of the node and all its child nodes, concatenated as the result.

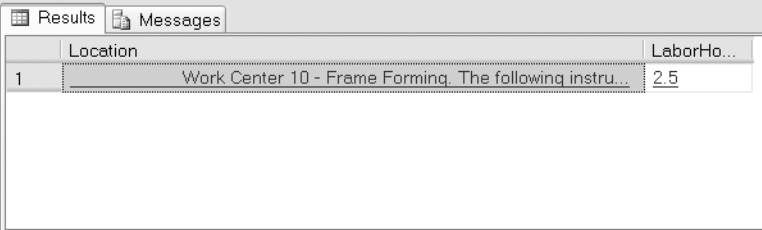
The `fn:data()` function also accepts an expression that returns a singleton value. Unlike `fn:string()`, the `fn:data()` function returns a typed instance of the value of its argument. Listing 6-3 demonstrates the use of `fn:string()` and `fn:data()`.

Listing 6-3. Data Accessor Function Sample

```
SELECT Instructions.query ('declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
    ProductModelManuInstructions";
fn:string((/root/Location)[1])') AS Location,
Instructions.query ('declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
    ProductModelManuInstructions";
```

```
fn:data((/root/Location/@LaborHours)[1])) AS LaborHours
FROM Production.ProductModel
WHERE ProductModelID = 7;
```

The `fn:string()` function in the example returns the `xs:string` value of the first `Location` node, while the `fn:data()` function returns the `xs:decimal` value of the first `@LaborHours` attribute. The results are shown in Figure 6-1.



	Location	LaborHo...
1	Work Center 10 - Frame Forming. The following instru...	2.5

Figure 6-1. Data accessor function query result

Cross-Platform Tip Although the XQuery recommendation provides for several data accessor functions that provide a wealth of information about items in data model instances, the current version of SQL Server supports only two of them—`fn:data()` and `fn:string()`.

Using String Functions

XQuery defines several functions that operate on `xs:string` instances (and types derived from `xs:string`). SQL Server supports a small subset of these functions, which I will describe in this section.

The `fn:concat()` function takes two or more `xs:string` arguments and concatenates them together into one `xs:string` instance. Any empty sequence arguments passed to `fn:concat()` are converted to an empty string before the concatenation.

The `fn:contains()` function takes two `xs:string` arguments and returns `true` if the second argument is contained in the first. The `fn:contains()` function is similar to the T-SQL `CHARINDEX` function, except that it returns an `xs:boolean` value instead of the integer start position of the second string. This function is case sensitive.

The `fn:substring()` function accepts an `xs:string` argument and one or two `xs:decimal` arguments. The function returns a substring of the `xs:string` argument beginning at the specified `xs:decimal` start position and with the specified optional `xs:decimal` length. If the length is not specified, `fn:substring()` returns everything from the specified start position to the end of the `xs:string`.

The `fn:string-length()` function accepts an `xs:string` argument and returns the string's length as an `xs:integer`. The length of the empty sequence is always 0.

Listing 6-4 demonstrates how to use the XQuery string functions.

Listing 6-4. *Sample XQuery String Functions*

```

DECLARE @x xml;
SET @x = N'';
SELECT @x.query('fn:concat("Germ", "any")');
SELECT @x.query('fn:substring("Early to bed and early to rise. . .", 10, 3)');
SELECT @x.query('fn:string-length("A penny saved is a penny earned")');
SELECT @x.query('fn:contains("Teach a man to fish", "fish")');

```

Following are the results of the sample code in Listing 6-4:

```

Germany
bed
31
true

```

Using the Boolean Function

XQuery provides the `fn:not()` function to return the opposite of a given `xs:boolean` value. The `fn:not()` function accepts a single argument and returns an `xs:boolean` value that is the opposite of the effective Boolean value of the argument. If the effective Boolean value of the argument is `true` then `fn:not()` returns `false`, but if the effective Boolean value is `false` then `fn:not()` returns `true`.

EFFECTIVE BOOLEAN VALUE

The effective Boolean value of a given value is the `xs:boolean` equivalent of a given value. The effective Boolean value is calculated by applying the following rules to a value:

1. The empty sequence returns `false`.
2. A sequence whose first item is a node returns `true`.
3. A singleton value of type `xs:boolean` (or derived from `xs:boolean`) returns the value unchanged.
4. A singleton value of type `xs:string`, `xs:anyURI`, or `xs:untypedAtomic` (or derived from any of these), returns `false` if the value has zero length; otherwise, it returns `true`.
5. A singleton value of any numeric type (or derived from a numeric type) returns `false` if the value is NaN or if it is numerically equal to zero; otherwise, it returns `true`.

The SQL Server XQuery implementation limits the types of values that can be passed to `fn:not()`. Only values of `xs:boolean` type or nodes can be passed in.

Listing 6-5 demonstrates how to use `fn:not()` to return the opposite of the effective Boolean value of its argument. Both of the sample XQuery expressions in Listing 6-5 return `true`.

Listing 6-5. *fn:not() Sample*

```

DECLARE @x xml;
SET @x = '';
SELECT @x.query('fn:not( fn:false() )');
SELECT @x.query('fn:not( 10 lt 3 )');

```

Using Numeric Functions

XQuery provides standard functions that operate on numeric values, and SQL Server XQuery supports three of them. The functions supported by SQL Server include `fn:ceiling()`, `fn:floor()`, and `fn:round()`.

The `fn:ceiling()` function accepts a single numeric argument, and it returns the smallest number (closest to negative infinity) with no fractional part that is not smaller than the argument.

The `fn:floor()` function also accepts a single numeric argument, and it returns the largest number (closest to positive infinity) with no fractional part that is not greater than the argument.

The `fn:round()` function accepts a single numeric argument and returns the number with no fractional part that is closest to the argument. The normal mathematical rules for rounding numbers with fractional parts are applied to the argument.

Listing 6-6 demonstrates the use of XQuery numeric functions.

Listing 6-6. *Using XQuery Numeric Functions*

```

DECLARE @x xml;
SET @x = '';
SELECT 'fn:round', @x.query('fn:round(-10.4)'),
    @x.query('fn:round(10.4)');
SELECT 'fn:ceiling', @x.query('fn:ceiling(-10.4)'),
    @x.query('fn:ceiling(10.4)');
SELECT 'fn:floor', @x.query('fn:floor(-10.4)'),
    @x.query('fn:floor(10.4)');

```

Following are the results of the numeric functions used in Listing 6-6:

fn:round	-10 10
fn:ceiling	-10 11
fn:floor	-11 10

Cross-Platform Tip Unfortunately, SQL Server's XQuery implementation does not include the `fn:abs()` absolute value function and the `fn:round-half-to-even()` rounding function. Leaving out the `fn:abs()` function is a particularly odd oversight that will hopefully be corrected in the next version of SQL Server XQuery.

NO USER-DEFINED FUNCTIONS

The SQL Server XML team decided not to implement the W3C-recommended ability to create XQuery user-defined functions in SQL 2008. This is unfortunate, as user-defined functions provide modularity, promote code reuse, and make code more readable. With user-defined functions, you can create modular code libraries to implement business logic or to fill in some of the basic functionality that is currently not implemented in the SQL Server version of XQuery. The mathematical absolute function `fn:abs()` jumps immediately to mind as one example. At any rate, we can only hope the team will implement this functionality in the next version of SQL Server.

Using Aggregate Functions

SQL Server developers will be familiar with the next class of XQuery functions—the aggregate functions. The XQuery aggregate functions are similar in name and functionality to their SQL counterparts. These functions are discussed in this section.

The `fn:count()` function is the equivalent of the SQL `COUNT` aggregate function. The `fn:count()` function returns the number of items contained in a sequence. The `fn:count()` function can accept heterogeneous sequences that mix different atomic types, like `xs:integer` and `xs:string`. Listing 6-7 counts the number of steps required in the instructions for product model 7. The query in Listing 6-7 reports that 28 steps are required to put together product model 7.

Listing 6-7. *Using the fn:count() Aggregate Function*

```
DECLARE @x xml;
SET @x = N'';
SELECT Instructions.query('declare default element namespace
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➡
    ProductModelManuInstructions";
    fn:count(//step)')
FROM Production.ProductModel
WHERE ProductModelID = 7;
```

The `fn:min()` function returns the minimum value from a sequence, and its opposite `fn:max()` returns the maximum value from a sequence. These two functions require that the sequences be homogenous. All items in the sequence must be of the same type, or of the same base type, and must be able to be compared using the `lt` and `gt` operators. You cannot use these functions on a sequence containing both `xs:integer` and `xs:string` values, for instance. But you can use the functions on a sequence containing `xs:integer` and `xs:decimal` values. Listing 6-8 is a simple demonstration of the `fn:min()` and `fn:max()` aggregate functions.

Listing 6-8. *fn:min() and fn:max() Demonstration*

```
DECLARE @x xml;
SET @x = '';
SELECT @x.query('fn:min((1, 2.9, 3.14, 4, 5.0))');
SELECT @x.query('fn:max(("Florida", "California", "Georgia"))');
```


Following are the results of the demonstration in Listing 6-8:

```
1
Georgia
```

The `fn:avg()` and `fn:sum()` functions are the XQuery equivalent of the SQL AVG and SUM aggregate functions. The `fn:avg()` function returns the average of all values in the sequence, while `fn:sum()` returns the sum of all values in the sequence. The `fn:avg()` function is semantically equivalent to the `fn:sum() div fn:count()` for a given sequence. The `fn:sum()` and `fn:avg()` functions can only accept sequences of numeric types that are all of the same base type. The acceptable types are the three numeric base types, or `xdt:untypedAtomic`. Trying to sum or average any other types will raise a static type error. Listing 6-9 demonstrates the `fn:sum()` and `fn:avg()` aggregate functions.

Tip Because of the restriction on types that can be included in an `fn:sum()` or `fn:avg()` sequence, you can't sum or average a sequence like `(1.0, 2.0, 3e+1, 4.0)`. The aggregate functions won't work on a sequence like this that mixes the `xs:decimal` and `xs:double` types.

Listing 6-9. *fn:sum()* and *fn:avg()* Demonstration

```
DECLARE @x xml;
SET @x = N'';
SELECT @x.query('fn:sum( (1, 2.9, 3, 4, 5.0) )');
SELECT @x.query('fn:avg( (5, 6, 7.0, 8, 9) )');
```

Following are the results of the example in Listing 6-9:

```
15.9
7
```

Using `fn:sum()` on the empty sequence returns 0, while using `fn:avg()` on the empty sequence results in an error.

Using Sequence Functions

XQuery defines a wide variety of functions specifically for use on sequences. SQL Server XQuery supports only a small selection of these functions. I'll describe the functions on sequences that SQL Server supports in this section.

The `fn:empty()` function accepts a sequence and returns `true` if the sequence is the empty sequence; otherwise, the function returns `false`.

Cross-Platform Tip The W3C also defines `fn:exists()`, which returns `true` if the sequence is not the empty sequence, the opposite of `fn:empty()`. You can simulate this functionality in SQL Server by using `fn:not(fn:empty())`. This seems like an oversight in the implementation of SQL Server XQuery. It would seem to be a trivial matter to implement `fn:exists()` once `fn:empty()` was implemented. We will see if this and other standard functions are added in future versions of SQL Server XQuery.

The `fn:distinct-values()` function performs a function similar to the T-SQL `DISTINCT` keyword. This function removes all duplicate values from a sequence. The SQL Server implementation of `fn:distinct-values()` requires that all values in the sequence be of comparable types. The function will not work on heterogeneous sequences that mix noncomparable types like `xs:decimal` and `xs:string`. Listing 6-10 shows the `fn:empty()` and `fn:distinct-values()` functions in action.

Listing 6-10. *fn:empty() and fn:distinct-values() Demonstration*

```
DECLARE @x xml;
SET @x = N'';
SELECT @x.query('fn:empty( ) ');
SELECT @x.query('fn:distinct-values( (1.0, 1, 2.0, 4.0, 8, 8.00) )');
```

The results of the sample function calls in Listing 6-10 follow:

```
true
1 2 4 8
```

The `fn:id()` function returns a sequence of element `xs:ID` values that match the values of one or more `xs:IDREF` values supplied as an argument. Listing 6-11 demonstrates the use of the `fn:id()` function.

Listing 6-11. *fn:id() Function Sample*

```
CREATE XML SCHEMA COLLECTION StateSchema
AS N'<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "Country">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name = "Region" minOccurs = "0" maxOccurs = "unbounded">
          <xsd:complexType mixed="true">
            <xsd:attribute name = "id" type = "xsd:ID"/>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name = "State" minOccurs = "0" maxOccurs = "unbounded">
          <xsd:complexType>
            <xsd:attribute name = "area" type = "xsd:IDREF" />
            <xsd:attribute name = "id" type = "xsd:ID" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
```

```

        <xsd:attribute name = "name" type = "xsd:string" />
    </xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:attribute name = "id" type = "xsd:ID" />
<xsd:attribute name = "name" type = "xsd:string" />
</xsd:complexType>
</xsd:element>
</xsd:schema>;
GO

DECLARE @x xml(StateSchema);

SET @x = N'<Country name = "United States" id = "US">
  <Region id = "East-Coast">The east coast of the United States</Region>
  <Region id = "West-Coast">The west coast of the United States</Region>
  <State name = "New Jersey" id = "NJ" area = "East-Coast" />
  <State name = "New York" id = "NY" area = "East-Coast" />
  <State name = "California" id = "CA" area = "West-Coast" />
  <State name = "Washington" id = "WA" area = "West-Coast" />
</Country>';

SELECT @x.query('fn:id((/Country/State[@id="NJ"]/@area))');
GO

```

This example could use a little more explanation. The first step is to create an XML schema collection so you can define a strongly typed `xml` instance. The XML schema defines `Region` elements with an `xs:ID` attribute and `State` elements with `xs:IDREF` attributes that refer back to the corresponding `Region` elements. Then you create a typed `xml` instance and populate it. Now the tricky part: the `fn:id()` query returns the `Region` corresponding to the area defined for the State of New Jersey. To add more detail, the State of New Jersey has an `IDREF` of `East-Coast`. This references the `East-Coast` `Region` `xs:ID`, which the `fn:id()` function returns. Following is the result:

```
<Region id = "East-Coast">The east coast of the United States</Region>
```

The W3C defines several additional functions on sequences that are not implemented in the current version of SQL Server XQuery.

Using Node Functions

The XQuery recommendation defines several functions on nodes, of which SQL Server XQuery supports three. I'll discuss these supported functions in this section.

The `fn:number()` function accepts a node as an argument and returns the node's numeric value. For values that cannot be converted to numbers, an empty sequence is returned. The `fn:local-name()` function returns the local name part of a node name as an `xs:string`. The `fn:namespace-uri()` function returns the namespace URI for a given node.

Listing 6-12 demonstrates the use of these three functions to retrieve node data from an entry in the AdventureWorks Production.ProductModel XML Instructions. The result is shown in Figure 6-2.

Listing 6-12. *Functions on Nodes Demonstration*

```
SELECT Instructions.query('declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
ProductModelManuInstructions";
    fn:number((/root/Location/@LaborHours)[1])) AS Number,
Instructions.query('declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
ProductModelManuInstructions";
    fn:local-name((/root/Location)[1])) AS LocalName,
Instructions.query('declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
ProductModelManuInstructions";
    fn:namespace-uri((/root/Location)[1])) AS URI
FROM Production.ProductModel
WHERE ProductModelID = 7;
```



	Numb...	LocalNa...	URI
1	2.5	Location	http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManuInstructions

Figure 6-2. *Result of functions on nodes*

Using Context Functions

XQuery also defines a handful of context functions, which allow you to retrieve useful information from the dynamic context at runtime. SQL Server XQuery only supports two of these functions: `fn:last()` and `fn:position()`.

The `fn:last()` function returns the number of items in the current XQuery context. Sequence items are indexed beginning with the number 1, and this function returns the `xs:integer` value of the last item in the sequence. When used in a predicate, the last item in the current context sequence is returned. Essentially this function returns the index number of the last item in the current sequence being processed.

The `fn:position()` function returns the index number of the current context item in a sequence being processed. Both of these functions can be used without arguments in the predicate of a path expression.

Listing 6-13 uses the `fn:last()` and `fn:position()` functions in an XQuery predicate to return the third and last tools listed for a product model.

Listing 6-13. *fn:last() and fn:position() Sample*

```

SELECT Instructions.query('declare default element namespace
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
    ProductModelManuInstructions";
    (//tool)[fn:position() = 3 or fn:position() = fn:last()]') AS Tools
FROM Production.ProductModel
WHERE ProductModelID = 7;

```

Following are the results of Listing 6-13:

```

<tool xmlns =
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
    ProductModelManuInstructions">router with a carbide tip 15</tool>
<tool xmlns =
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
    ProductModelManuInstructions">paint harness</tool>

```

DATE AND TIME CONTEXT FUNCTIONS

SQL Server XQuery doesn't support date and time context functions like `current-dateTime`, `current-date`, and `current-time`. These functions can be simulated using the T-SQL `GETDATE` and `CAST` functions, the XQuery constructor functions, and the `sql:variable()` function.

```

DECLARE @now datetime,
        @date date,
        @time time;

SET @now = GETDATE();
SET @date = CAST(@now AS date);
SET @time = CAST(@now AS time);

DECLARE @x xml;
SET @x = N'';
SELECT @x.query('xs:dateTime(sql:variable("@now"))') AS [current-dateTime],
       @x.query('xs:date(sql:variable("@date"))') AS [current-date],
       @x.query('xs:time(sql:variable("@time"))') AS [current-time];

```

The T-SQL `datetime`, `date`, and `time` data types map to the XQuery `xs:dateTime`, `xs:date`, and `xs:time` data types. The T-SQL `date` and `time` data types are new to SQL Server 2008. By assigning the current system date/time to a T-SQL variable using `GETDATE`, you ensure that the values used in your XQuery are stable, as specified in the XQuery recommendation. *Stable* means that the date/time values do not change for the duration of the query.

I will discuss the `sql:variable()` function and constructor functions in the “Using Constructor Functions” and “Using SQL Server XQuery Extension Functions” sections of this chapter, respectively.

Using Constructor Functions

SQL Server supports a large set of XQuery constructor functions that allow you to create instances of most of the built-in XML Schema data types. Creating an instance of a data type with XQuery should be familiar to those with experience in object-oriented programming languages, like the .NET family of languages. The syntax is very similar:

prefix:TYPE (\$arg)

The *prefix:TYPE* specification is the XML Schema data type to create. The *\$arg* argument is the value to assign to the instance. If *\$arg* is the empty sequence, the instance is created and assigned the empty sequence instead of a value. All types with built-in constructors are listed in Table 6-4.

Table 6-4. *Types with Built-In Constructors*

Type		
Base Types		
xs:anyURI	xs:base64Binary	xs:boolean
xs:date	xs:dateTime	xs:decimal
xs:double	xs:duration	xs:float
xs:gDay	xs:gMonth	xs:gMonthDay
xs:gYear	xs:gYearMonth	xs:hexBinary
xs:string	xs:time	
Derived Types		
xs:byte	xs:ENTITY	xs:ID
xs:IDREF	xs:int	xs:integer
xs:language	xs:long	xs:Name
xs:NCName	xs:negativeInteger	xs:nonNegativeInteger
xs:nonPositiveInteger	xs:normalizedString	xs:positiveInteger
xs:short	xs:token	xs:unsignedByte
xs:unsignedInt	xs:unsignedLong	xs:unsignedShort

SQL Server XQuery also supports constructors for user-defined data types that are derived from these types. Additionally, the `xs:boolean` data type has two constructor functions to quickly create instances of the `xs:boolean` data type: `fn:true()` and `fn:false()`. The types that do not support constructors include `xs:duration`, `xs:NMTOKEN`, `xs:NOTATION`, `xs:QName`, `xdt:dayTimeDuration`, and `xdt:yearMonthDuration`. Listing 6-14 demonstrates a simple use of constructor functions.

Listing 6-14. *Constructor Function Sample*

```

DECLARE @x xml;
SET @x = N'';
SELECT @x.query('xs:integer(12345.6) + xs:decimal(1)');
SELECT @x.query('fn:true()');
SELECT @x.query('xs:dateTime("2007-10-31T12:32:46-05:00")');

```

The first query in the example creates an `xs:integer` instance and an `xs:decimal` instance, and then adds the two instances. The second query creates an instance of the `xs:boolean` value `true`. The third query converts a string value to an `xs:dateTime` instance. Following are the results of this example:

```

12346
true
2007-10-31T12:32:46-05:00

```

DATETIME AND TIME ZONES

In SQL Server 2005, the XQuery processor required that a time zone be included in all `xs:dateTime` instances. The XQuery processor in SQL Server 2005 converted the time zone of all `xs:dateTime` instances to UTC time, indicated by Z. In Listing 6-14, I've used the time zone offset "-05:00", which is Eastern Standard Time (US). Notice that SQL Server 2008 does not change the resulting time or time zone. In SQL Server 2005, the resulting time portion of the `xs:dateTime` instance would have been changed to "T17:32:46Z". The `xs:dateTime` instances in SQL Server 2008 also do not require a time zone. If no time zone is supplied, UTC is assumed.

Using QName Functions

In XQuery, QName functions are qualified names that create a mapping between a URI and a namespace prefix. This is really a convenience feature that makes code more manageable because you can use convenient, and often meaningful, abbreviations for namespaces instead of the full URI prefix for every identifier. Without QNames, you would have to manually expand your namespaces, using a complete name like the following to access something as simple as the `fn:not()` function.

```
{http://www.w3.org/2002/11/xquery-functions}:not()
```

The functions related to QNames that are supported by SQL Server include `fn:expanded-QName()`, `fn:local-name-from-QName()`, and `fn:namespace-uri-from-QName()`. These functions allow you to compare and manipulate QNames via XQuery. Listing 6-15 demonstrates their use with a simple example.

Listing 6-15. *Using QName Functions*

```

CREATE XML SCHEMA COLLECTION QNameSchema AS
'<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:element name="root" type="xs:QName"/>
</xs:schema>';
GO

DECLARE @x xml (QNameSchema);
SET @x = N'<root xmlns:ns = "http://TestURI">ns:localName</root>';

SELECT @x.query('fn:namespace-uri-from-QName(/root[1])') AS uri,
       @x.query('fn:local-name-from-QName(/root[1])') AS localname;
GO

```

This example creates a simple XML schema collection to store an `xs:QName` in an XML document. Then it creates a simple XML document with an `xs:QName` value. Finally, the QName functions are used to retrieve the URI and local name from the `xs:QName` in the document.

Using SQL Server XQuery Extension Functions

In addition to the supported XQuery functions, SQL Server provides additional functions to dynamically bind relational data in XML. There are two functions provided to access relational data from within your XQuery queries, `sql:variable()` and `sql:column()`. You've already seen `sql:variable()` in action. This function allows you to access T-SQL variables directly from within your XQuery queries.

The `sql:column()` function binds relational data from a column that you specify in your query. These functions help bridge the gap between relational data and XQuery, which makes SQL Server XQuery even more powerful. Listing 6-16 retrieves a row from the `Person.Contact` table and uses the `sql:column()` function to bind the relational data inside an XQuery XML constructor. The result is shown in Figure 6-3.

Listing 6-16. *Constructing XML with Relational Data*

```

SELECT jc.BusinessEntityID, Resume.query ('declare namespace ns =
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
<contact>
  <name> {
    sql:column("p.LastName"),
    sql:column("p.FirstName"),
    sql:column("p.MiddleName")
  } </name>
</contact>')
FROM HumanResources.JobCandidate jc
INNER JOIN HumanResources.Employee e
  ON jc.BusinessEntityID = e.BusinessEntityID
INNER JOIN Person.Person p
  ON e.BusinessEntityID = p.BusinessEntityID
WHERE jc.BusinessEntityID = 274;

```

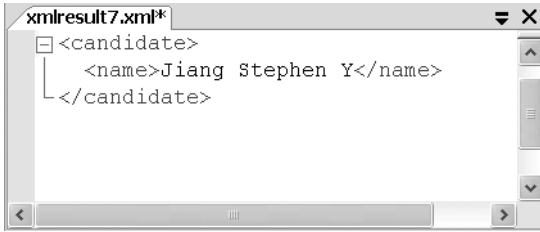



Figure 6-3. Result of XML construction with relational data binding

Modifying XML with XML DML

When XQuery was recommended, one of the concerns brought up was the lack of data manipulation language support. XQuery alone provides no mechanism for performing basic DML operations on XML, like node updates, deletions, and insertions. SQL Server provides extensions to XQuery that give you these capabilities in the form of XML DML.

I introduced XML DML in Chapter 5 with a description of the `xml` data type `modify()` method. XML DML is a set of SQL Server XQuery extensions that provide the ability to manipulate XML data. XML DML consists of three statements that are analogous to their corresponding SQL DML statements. I will discuss each of these XML DML statements and look at usage examples in this section.

Inserting Nodes with insert

The first XML DML statement I will cover is the `insert` statement. This statement inserts one or more nodes into an existing `xml` instance. The XML DML `insert` statement is analogous to the SQL `INSERT` statement. The `insert` statement requires three parts: an expression that evaluates to a node to insert, a location specification, and a second expression to provide a location at which the `insert` operation should take place. The second expression must evaluate to a singleton node. Listing 6-17 uses the XML DML `insert` statement to insert a new `Employment` node into a job applicant's resume.

Note The `xml` data type `modify()` method can only be used in a T-SQL `SET` statement or in the `SET` clause of an `UPDATE` statement.

Listing 6-17. *insert Operation Sample*

```
DECLARE @x xml;

SELECT @x = Resume
FROM HumanResources.JobCandidate
WHERE BusinessEntityID = 274;

SET @x.modify('declare namespace ns =
```

```
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
insert
(
  <ns:Employment>
    <ns:Emp.StartDate>2001-01-01Z</ns:Emp.StartDate>
    <ns:Emp.EndDate>2007-10-31Z</ns:Emp.EndDate>
    <ns:Emp.OrgName>AdventureWorks</ns:Emp.OrgName>
    <ns:Emp.JobTitle>Vice President of Sales</ns:Emp.JobTitle>
  </ns:Employment>
)
before
(/ns:Resume/ns:Employment)[1]');

SELECT @x;
```

In the example, you first retrieve AdventureWorks employee Stephen Jiang's resume into an xml variable. Then you use the XML DML insert statement to insert a new Employment node before the current first Employment node in the resume. The result is a more complete picture of Mr. Jiang's employment history, including his most recent employment with AdventureWorks, as shown in the following snippet of XML data.

```
. . .
<ns:Employment>
  <ns:Emp.StartDate>2001-01-01Z</ns:Emp.StartDate>
  <ns:Emp.EndDate>2007-10-31Z</ns:Emp.EndDate>
  <ns:Emp.OrgName>AdventureWorks</ns:Emp.OrgName>
  <ns:Emp.JobTitle>Vice President of Sales</ns:Emp.JobTitle>
</ns:Employment>

<ns:Employment>
  <ns:Emp.StartDate>1998-03-01Z</ns:Emp.StartDate>
  <ns:Emp.EndDate>2000-12-30Z</ns:Emp.EndDate>
  <ns:Emp.OrgName>Wide World Imports</ns:Emp.OrgName>
  <ns:Emp.JobTitle>Sales Manager</ns:Emp.JobTitle>
. . .
```

Let's take a moment to look at the separate clauses of this insert statement. The insert keyword is immediately followed by the node to be inserted:

```
insert
(
  <ns:Employment>
    <ns:Emp.StartDate>2001-01-01Z</ns:Emp.StartDate>
    <ns:Emp.EndDate>2007-10-31Z</ns:Emp.EndDate>
    <ns:Emp.OrgName>AdventureWorks</ns:Emp.OrgName>
    <ns:Emp.JobTitle>Vice President of Sales</ns:Emp.JobTitle>
  </ns:Employment>
)
```

The `before` clause specifies that the node should be inserted directly before the first `Employment` node:

```
before
  (/ns:Resume/ns:Employment)[1]
```

Alternatively, you could replace the keyword `before` with `after` to insert the new node directly after the specified node as a sibling node. Or you could specify `into`, as `first into`, or as `last into` to indicate that the new node should be inserted as a child of the specified node.

By popular demand, SQL Server 2008 expands on the XML DML functionality in the 2005 release by adding support for parameterized insert statements. You can now use SQL Server `xml` data type variables in the `insert` clause of XML DML. Listing 6-18 modifies Listing 6-17 to demonstrate this new feature.

Listing 6-18. *Using `sql:variable()` with XML DML insert Statement*

```
DECLARE @x xml,
        @y xml;

SET @y = N'<ns:Employment xmlns:ns =
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume">
  <ns:Emp.StartDate>2001-01-01Z</ns:Emp.StartDate>
  <ns:Emp.EndDate>2007-10-31Z</ns:Emp.EndDate>
  <ns:Emp.OrgName>AdventureWorks</ns:Emp.OrgName>
  <ns:Emp.JobTitle>Vice President of Sales</ns:Emp.JobTitle>
</ns:Employment>';

SELECT @x = Resume
FROM HumanResources.JobCandidate
WHERE BusinessEntityID = 274;

SET @x.modify('declare namespace ns =
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
insert
  sql:variable("@y")
before
  (/ns:Resume/ns:Employment)[1]');

SELECT @x;
```

An important thing to note in this example is the addition of the `xmlns:ns` declaration in the `xml` variable content. This namespace declaration is necessary to ensure that the XML content you insert has the proper namespace. The `sql:column()` function is not supported in XML DML statements.

XML DML PERFORMANCE

XML DML, like SQL Server XQuery path expressions, are designed to leverage the power of the SQL Server engine. When you execute an XML DML statement like `insert`, SQL Server performs the function on the relational “shredded” version of the XML. This has performance implications for XML DML. If you are performing XML DML statements against a non-XML indexed `xml` data type instance, SQL Server must first shred the XML data, perform the XML DML statement, and convert it back to XML form. This can incur quite a bit of overhead.

If you are performing the XML DML statement on an XML-indexed instance, the XML indexes must be updated. While this operation is usually less costly than shredding an `xml` data type instance and converting it back to XML on the fly, it is still an overhead consideration that should be considered.

The SQL Server team has also done a good job of optimizing XML DML updates. After an XML DML modification, only the data pages that are changed by the update are actually modified. Despite the SQL Server team’s optimizations, when many XML DML modifications are involved, it makes sense at some point to consider replacing an `xml` instance in its entirety rather than update it in a piecemeal fashion.

Deleting Nodes with delete

The XML DML delete statement is the XML DML equivalent of the SQL `DELETE` statement. The delete statement takes an XQuery expression specifying the node to delete. Listing 6-19 uses the XML DML delete statement to delete a phone number from Mr. Jiang’s resume.

Listing 6-19. *delete Statement Sample*

```
DECLARE @x xml;

SELECT @x = Resume
FROM HumanResources.JobCandidate
WHERE BusinessEntityID = 274;

SET @x.modify('declare namespace ns =
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
delete (/ns:Resume/ns:Address/ns:Addr.Telephone/ns:Telephone
[./ns:Tel.Number = "555-1981"])[1]');

SELECT @x;
```

If you look at Mr. Jiang’s XML resume, you can see that he has two phone numbers on his resume, both “555-1981” and “555-1119”. The delete statement in the listing uses an XQuery path expression to locate the Telephone node containing a Tel.Number node equal to “555-1981”. Once that node is located, it’s deleted.

```
delete (/ns:Resume/ns:Address/ns:Addr.Telephone/ns:Telephone
[./ns:Tel.Number = "555-1981"])[1]
```

The following XML snippet shows the result produced by the example. Notice that Mr. Jiang now has just one Telephone node listed on his resume.

```

<ns:Addr.Telephone>
  <ns:Telephone>
    <ns:Tel.Type>Voice</ns:Tel.Type>
    <ns:Tel.IntlCode>1</ns:Tel.IntlCode>
    <ns:Tel.AreaCode>425</ns:Tel.AreaCode>
    <ns:Tel.Number>555-1119</ns:Tel.Number>
  </ns:Telephone>
</ns:Addr.Telephone>

```

Unlike the insert statement, the path expression for a delete statement is not required to evaluate to a singleton node. You can use the delete statement to delete multiple nodes simultaneously. Use this feature with care, however, so that you don't accidentally delete too many nodes from your xml instances.

Updating Nodes with replace value of

The third and final XML DML statement is the replace value of statement, which is equivalent to the SQL UPDATE statement. The replace value of statement accepts two expressions—one that indicates the node to be replaced and one that specifies the value to replace the node's current value. Like the insert statement, replace value of can only operate on one node at a time, so a singleton node must be specified. Listing 6-20 uses replace value of to update the middle name on Mr. Jiang's resume.

Listing 6-20. replace value of XML DML Sample

```

DECLARE @x xml;

SELECT @x = Resume
FROM HumanResources.JobCandidate
WHERE BusinessEntityID = 274;

SET @x.modify('declare namespace ns =
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
replace value of (/ns:Resume/ns:Name/ns:Name.Middle/text())[1]
with ("Yancy")');

SELECT @x;

```

The replace value of clause locates the text() node of the Name.Middle node in Mr. Jiang's resume:

```
replace value of (/ns:Resume/ns:Name/ns:Name.Middle/text())[1]
```

The with clause indicates that the new value of the node should be set to "Yancy":

```
with ("Yancy")
```

The end result is that Mr. Jiang's middle initial ("Y") is replaced with his full middle name ("Yancy"). This is shown in the following XML snippet.

```
<ns:Name>
  <ns:Name.Prefix>Mr.</ns:Name.Prefix>
  <ns:Name.First>Stephen</ns:Name.First>
  <ns:Name.Middle>Yancy</ns:Name.Middle>
  <ns:Name.Last>Jiang</ns:Name.Last>
  <ns:Name.Suffix />
</ns:Name>
```

Summary

The SQL Server XML team implemented a solid subset of the W3C-recommended XQuery functions and operators. Included are the complete set of logic operators, the vast majority of math operators, a basic subset of standard XQuery functions, and some SQL Server-specific functions. In this chapter, I reviewed the operators introduced in previous chapters and covered all XQuery functions available in SQL Server 2008. I also discussed the W3C-recommended functionality that the SQL Server team decided not to implement for the current release of SQL Server.

I also revisited XML DML, which I introduced in Chapter 5, and discussed all of the available XML DML statements in detail. I discussed examples of the insert, delete, and replace value of statements and explained how they work. An important new addition to XML DML is support for the `sql:variable()` function in insert statements. I also discussed the performance impact of XML DML and considered that in some cases, wholesale replacement of `xml` instances might be more efficient than XML DML.

In the next chapter, I'll introduce SQL Server's XML indexing mechanisms and capabilities that help maximize XML query performance.



Indexing XML

It's not hard to imagine that the SQL Server team discovered early on that the standard SQL Server relational b-tree indexing mechanisms wouldn't work with XML data. The XQuery language, which I discussed in Chapter 5, is designed to query XML nodes while SQL is designed to query relational data stored in tables. XQuery supports a wide variety of functions, operators, and predicates that require access to all types of nodes, especially element and attribute nodes. The standard SQL Server relational indexes do not optimize access to individual XML nodes, which is a requirement to optimize XQuery access.

With that in mind, the SQL Server team developed a special indexing mechanism, applicable only to `xml` data type instances, to optimize XQuery queries. In this chapter, I'll discuss XML indexes and how they can be used to maximize XQuery performance. I'll round out this chapter with a discussion of full-text indexes on the `xml` data type.

Creating a Primary XML Index

In Chapter 5, I discussed how SQL Server XQuery leverages the power of the SQL Server relational query engine to query XML data. When you perform an XQuery query against an `xml` instance, SQL Server shreds your XML data into relational format. Creating a primary index on your `xml` columns causes SQL Server to preshred your XML data, essentially getting the shredding process out of the way up front. Creating a primary XML index eliminates the overhead associated with shredding your XML data during every XQuery query you perform against it. There are three main reasons to create a primary XML index:

- You have several rows of any size XML data in a table, and your usage patterns require you to perform many XQuery queries against them.
- You store large XML documents or content in each row of a table, and your usage patterns indicate you will perform XQuery queries against them regularly.
- You want to create secondary XML indexes on a column to optimize your XQuery queries for a specific type of query.

If you have very little XML data stored in a table or don't plan on performing XQuery queries against the data, the storage requirements for a primary XML index may not be worth the speed increase achieved.

When you create a primary XML index on an `xml` column, SQL Server creates a relational node table representation of your XML data. A simplified representation of a single XML document converted to the node table format is shown in Figure 7-1.

PK	ORDPATH	TAG	NODE	NODETYPE	VALUE	HID
1	1	1 (Resume)	Element	12 (ResumeT)	<i>null</i>	#Resume
1	1.1	2 (Name)	Element	13 (NameT)	<i>null</i>	#Name#Resume
1	1.1.1	3 (Prefix)	Element	2 (xs:string)		#Prefix#Name#Resume
1	1.1.3	4 (First)	Element	2 (xs:string)	Shai	#First#Name#Resume
1	1.1.5	5 (Middle)	Element	2 (xs:string)		#Middle#Name#Resume
1	1.1.7	6 (Last)	Element	2 (xs:string)	Bassli	#Last#Name#Resume
1	1.1.9	7 (Suffix)	Element	2 (xs:string)		#Suffix#Name#Resume

Figure 7-1. Primary XML index node table format (simplified)

You should recognize this node table format from Chapter 5. This is similar to the node table format that's generated when you query non-XML indexed `xml` instances. The main difference between the non-XML indexed node table format and the XML indexed node table format is that the XML indexed version includes a PK column containing the primary key of the table. This PK column in the node format table relates XML data back to a specific row of the source relational table. The contents of the PK column are highlighted in Figure 7-1.

SQL Server provides the `CREATE PRIMARY XML INDEX` statement to create a primary XML index. The `ALTER INDEX` and `DROP INDEX` statements can be used to manage existing primary XML indexes. SQL Server will not allow you to create a primary XML index on a column in a table that does not have a clustered primary key defined on it. This helps ensure that if your table is partitioned, the primary XML index will be properly partitioned as well. There are additional restrictions placed on primary XML index creation, including the following:

- You cannot create a primary XML index on a computed column or `xml` variable, or in a view or table variable.
- You cannot create a primary XML index on a non-`xml` column.
- There can be only one primary XML index on a given `xml` column.
- Although a table can contain multiple primary XML indexes, a primary XML index can contain only a single `xml` column.
- If you change the type of an `xml` column from untyped to typed, or vice versa, you must drop an existing primary XML index first.
- The primary XML index name must be unique; it cannot be the same name as a relational index on the table.
- To create a primary XML index the `ANSI_PADDING`, `ANSI_NULLS`, `QUOTED_IDENTIFIER`, `CONCAT_NULL_YIELDS_NULL`, `ANSI_WARNINGS`, and `ARITHABORT` set options must be set to `ON`. Also the `NUMERIC_ROUNDABORT` set option must be set to `OFF`.

Listing 7-1 creates a primary XML index on the ProductDescription column of the AdventureWorks Production.ProductModel table.

Listing 7-1. *Primary XML Index Creation*

```
SET ARITHABORT ON;
SET CONCAT_NULL_YIELDS_NULL ON;
SET QUOTED_IDENTIFIER ON;
SET ANSI_NULLS ON;
SET ANSI_PADDING ON;
SET ANSI_WARNINGS ON;
SET NUMERIC_ROUNDABORT OFF;
GO

IF EXISTS ( SELECT 1
            FROM sys.indexes i
            WHERE i.name = 'PXML_ProductModel_CatalogDescription'
          )
  DROP INDEX PXML_ProductModel_CatalogDescription
  ON Production.ProductModel;
GO

CREATE PRIMARY XML INDEX PXML_ProductModel_CatalogDescription
ON Production.ProductModel
(
  CatalogDescription
);
GO
```

Notice that the example sets all options properly prior to the primary XML index creation. I also demonstrate the DROP INDEX statement, which requires the newer DROP INDEX *index_name* ON *table_name* syntax. Finally, I create the primary XML index on the xml column.

Caution The old-style T-SQL DROP INDEX syntax, DROP INDEX *table_name.index_name*, has been deprecated and will be removed from a future version of SQL Server. The old-style syntax will not work with XML indexes on SQL Server 2008.

Listing 7-2 includes a simple example XQuery query that demonstrates the efficiency difference between querying XML data with no index and querying XML data with a primary XML index in place.

Listing 7-2. *Simple XQuery Efficiency Example*

```

SELECT Instructions.query ('declare namespace ns =
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➡
    ProductModelManuInstructions";
    /ns:root/ns:Location/ns:step/ns:tool[. = "T-85A framing tool"]')
FROM Production.ProductModel;

```

The sample XQuery query in Listing 7-2 queries the product model XML instructions and returns all tool nodes that specify the T-85A framing tool. The query is deliberately simple, since I just want to look at the efficiency of the generated query plan. Figure 7-2 contains the relevant snippet of the query plan generated by Listing 7-2 when there is no primary XML index on the Instructions column of the Production.ProductModel table.

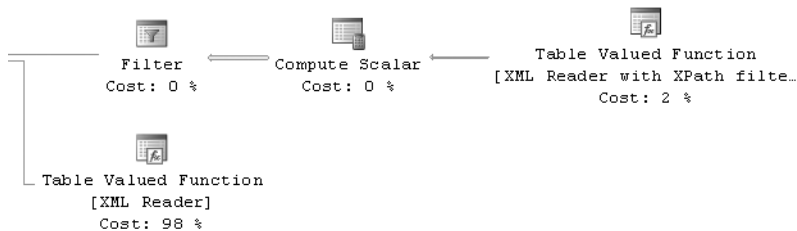


Figure 7-2. *Sample XQuery query plan with no XML index (partial)*

The important thing to notice about this query plan is that it includes a Table Valued Function [XML Reader with XPath filter] step and a Table Valued Function [XML Reader] step that account for almost 100 percent of the query plan cost when taken together. These steps implement the runtime XML shredding mechanism, resulting in a total query plan cost of over 7,000.0! After applying the primary XML index in Listing 7-1, the query plan changes considerably, as shown in Figure 7-3.

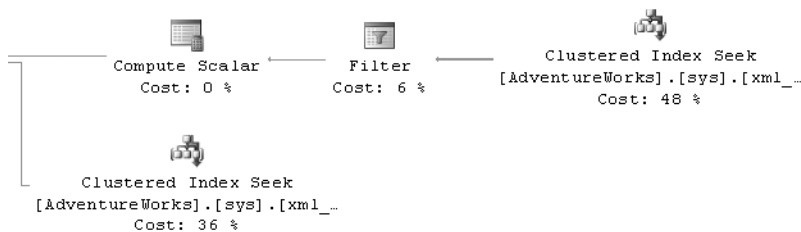


Figure 7-3. *Sample XQuery query plan with primary XML index (partial)*

When the primary XML index is added to the equation, the optimizer is able to replace the various Table Valued Function [XML Reader] operations designed to shred the XML data with far more efficient Clustered Index Seeks on the preshredded data in the primary XML index. The Clustered Index Seeks total up to 84 percent of the total query plan cost, and the total cost of the query plan drops significantly, from over 7,000.0 to a much lower cost of around 0.16.

XML INDEXING CONSIDERATIONS

As you saw in the simple example in Listing 7-2, XML indexing can improve XQuery query performance considerably. With such large gains in performance, XML indexing can be a very attractive solution to improve SQL Server XQuery performance. So the question then becomes, “What is the downside?”

The number-one drawback to adding a primary XML index is that it can consume a large amount of storage space. Remember, when you create a primary XML index, SQL Server is storing all of your XML data in a given column in relational format. This can easily more than double the storage space required to store your XML data. My recommendations here are simple:

1. Decide whether or not you really need to index an `xml` column. If you aren’t planning to query an `xml` column with XQuery, there’s no point in adding a primary XML index to it.
2. Determine your query patterns. If you only perform very occasional queries on an `xml` column, you are querying against a small number of rows at a time, and your XML data is relatively small in size, the added efficiency that a primary XML index brings with it will probably not be worth the storage trade-off.
3. Consider your additional XML indexing needs. If you need to add a secondary XML index to increase the efficiency of a specific class of queries (discussed in the next section), adding a primary XML index is mandatory.

Keep this in mind when planning your XML indexing strategies.

Creating Secondary XML Indexes

While the primary XML index improves XQuery query performance by eliminating the runtime shredding of XML data, performance can still be improved further in many cases by adding another index to the relational representation of your XML data. The secondary XML index provides this functionality.

The secondary XML index can be created with the `CREATE XML INDEX` statement, and existing secondary XML indexes can be managed with the `ALTER INDEX` and `DROP INDEX` statements. You can create three types of secondary XML indexes—`PATH`, `VALUE`, and `PROPERTY`. I will cover all three types in this section.

I already talked about the restrictions on primary XML index creation in the previous section, but there are some additional restrictions that are specific to secondary XML index creation. A secondary XML index can only be created on a column with an existing primary XML index. So the minimum requirements for primary XML index creation also apply to secondary XML index creation (e.g., there must be a primary key on the table).

In addition, you cannot drop a primary XML index if there are any existing secondary XML indexes that rely on it. You must drop the secondary XML indexes before dropping the primary XML index.

Creating PATH Secondary XML Indexes

The PATH secondary XML index is designed to optimize the query access for XQuery path queries of the following types:

- Only the path is specified.
- Only the path is specified and a node value is used in a predicate.

Listing 7-3 builds a PATH secondary XML index on the XML instructions I've been querying in this chapter.

Listing 7-3. Create PATH Secondary XML Index

```
CREATE XML INDEX SXML_ProductModel_Instructions_PATH
ON Production.ProductModel
(
    Instructions
)
USING XML INDEX PXML_ProductModel_Instructions
FOR PATH;
```

In addition to the XML index name, table name, and XML column name, the secondary XML index creation statement requires you to specify the name of the primary XML index you're indexing in the `USING XML INDEX` clause and the type of secondary XML index in the `FOR` clause. In this example, I've chosen to create a PATH secondary XML index. To understand the secondary XML index, it's important to recall that the primary XML index is a relational representation (columns and rows) of your `xml` column. The PATH secondary XML index creates a relational nonclustered index on the primary XML index columns `HID`, `VALUE`, `PK`, and `ORDPATH`, as shown in Figure 7-4.

PK	ORDPATH	TAG	NODE	NODETYPE	VALUE	HID
1	1	1 (Resume)	Element	12 (ResumeT)	<i>null</i>	#Resume
1	1.1	2 (Name)	Element	13 (NameT)	<i>null</i>	#Name#Resume
1	1.1.1	3 (Prefix)	Element	2 (xs:string)		#Prefix#Name#Resume
1	1.1.3	4 (First)	Element	2 (xs:string)	Shai	#First#Name#Resume
1	1.1.5	5 (Middle)	Element	2 (xs:string)		#Middle#Name#Resume
1	1.1.7	6 (Last)	Element	2 (xs:string)	Bassli	#Last#Name#Resume
1	1.1.9	7 (Suffix)	Element	2 (xs:string)		#Suffix#Name#Resume
3			2		1	

Figure 7-4. PATH secondary XML index on node table format data (simplified)

As you can see, this type of secondary XML index is highly optimized for retrieval of data by XQuery path expression and then narrowing of those results by predicate value. The example query used in Listing 7-2 generates a completely different query plan when the PATH secondary XML index is applied, as shown in Figure 7-5.

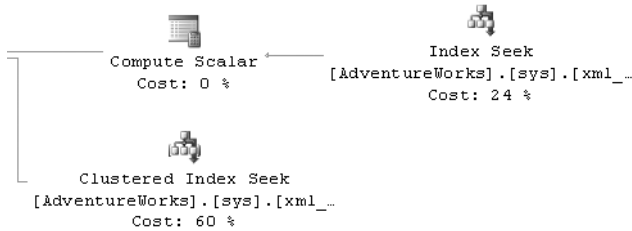


Figure 7-5. Sample XQuery query plan with PATH secondary XML index (partial)

The new query plan shows that the PATH secondary XML index further optimizes the query plan by adding an Index Seek on the secondary XML index. The total plan cost drops from over 7,000.0 with no XML indexes to around 0.16 with only a primary XML index, and then down to around 0.098 with the primary and secondary XML indexes in place.

Tip The PATH secondary XML index is particularly useful in optimizing queries where the `xml` data type `exist()` method is used in the SQL WHERE clause predicate.

Creating VALUE Secondary XML Indexes

The second type of secondary XML index, the VALUE secondary XML index, helps to optimize the following kinds of queries:

- Queries that specify a node value in a predicate and use the descendant-or-self axis (// in abbreviated form) to locate the nodes.
- Queries that use the wildcard character (*) in a path expression or predicate node name.

Basically, the VALUE secondary XML index optimizes for queries where the value is known, but the name and/or exact location of the node containing the value are not known at query time. Listing 7-4 creates a VALUE secondary XML index on the XML instructions I've been querying in this chapter.

Listing 7-4. Create VALUE Secondary XML Index

```

CREATE XML INDEX SXML_ProductModel_Instructions_VALUE
ON Production.ProductModel
(
    Instructions
)
USING XML INDEX PXML_ProductModel_Instructions
FOR VALUE;
  
```

Figure 7-6 shows the nonclustered VALUE secondary XML index created on the primary XML index columns VALUE, HID, PK, and ORDPATH. As you can see from this figure, the VALUE secondary XML index is highly optimized for queries where the node value is known but the full paths to the nodes containing that value are not known.

PK	ORDPATH	TAG	NODE	NODETYPE	VALUE	HID
1	1	1 (Resume)	Element	12 (ResumeT)	null	#Resume
1	1.1	2 (Name)	Element	13 (NameT)	null	#Name#Resume
1	1.1.1	3 (Prefix)	Element	2 (xs:string)		#Prefix#Name#Resume
1	1.1.3	4 (First)	Element	2 (xs:string)	Shai	#First#Name#Resume
1	1.1.5	5 (Middle)	Element	2 (xs:string)		#Middle#Name#Resume
1	1.1.7	6 (Last)	Element	2 (xs:string)	Bassli	#Last#Name#Resume
1	1.1.9	7 (Suffix)	Element	2 (xs:string)		#Suffix#Name#Resume

Diagram illustrating the VALUE secondary XML index structure. The table shows columns PK, ORDPATH, TAG, NODE, NODETYPE, VALUE, and HID. Below the table, three shaded triangular regions are labeled 1, 2, and 3, representing different parts of the index structure.

Figure 7-6. VALUE secondary XML index on node table format data (simplified)

The query in Listing 7-5 demonstrates a simple query that makes use of the VALUE secondary index.

Listing 7-5. Simple VALUE Secondary XML Index Efficiency Sample

```
SELECT Instructions.query ('declare namespace ns =
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/"
ProductModelManuInstructions";
//ns:step[ns:tool = "T-85A framing tool"]')
FROM Production.ProductModel;
```

The partial query plan for the VALUE secondary XML index uses an Index Seek on the VALUE secondary XML index to query the XML data, as shown in Figure 7-7.

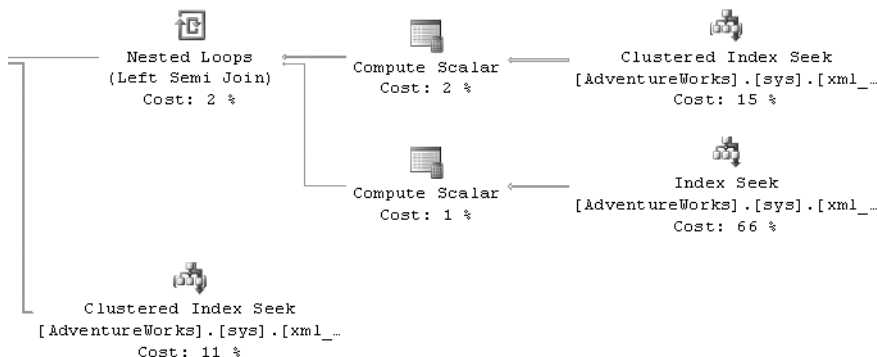


Figure 7-7. Sample XQuery query plan with VALUE secondary XML index (partial)

The total query cost is around 0.51, which is lower than the same query with only a primary XML index (cost around 0.58) or with no XML indexes (cost around 495.0!).

Creating PROPERTY Secondary XML Indexes

The third and final type of secondary XML index is the PROPERTY secondary XML index. This type of XML index is generally used to improve performance of queries that use the `xml` data type `value()` method to retrieve one or more values from your XML data. This type of XML, in which XML data is configured to act as a container for scalar values to be queried on an individual basis, is commonly referred to as a *property bag* configuration. Listing 7-6 creates a PROPERTY secondary XML index on the XML data that has been queried in the examples of this chapter so far.

Listing 7-6. Create PROPERTY Secondary XML Index

```
CREATE XML INDEX SXML_ProductModel_Instructions_PROPERTY
ON Production.ProductModel
(
    Instructions
)
USING XML INDEX PXML_ProductModel_Instructions
FOR PROPERTY;
```

The PROPERTY secondary XML index creates a nonclustered index on the PK, HID, VALUE, and ORDPATH columns of your primary XML index, as shown in Figure 7-8.

PK	ORDPATH	TAG	NODE	NODETYPE	VALUE	HID
1	1	1 (Resume)	Element	12 (ResumeT)	<i>null</i>	#Resume
1	1.1	2 (Name)	Element	13 (NameT)	<i>null</i>	#Name#Resume
1	1.1.1	3 (Prefix)	Element	2 (xs:string)		#Prefix#Name#Resume
1	1.1.3	4 (First)	Element	2 (xs:string)	Shai	#First#Name#Resume
1	1.1.5	5 (Middle)	Element	2 (xs:string)		#Middle#Name#Resume
1	1.1.7	6 (Last)	Element	2 (xs:string)	Bassli	#Last#Name#Resume
1	1.1.9	7 (Suffix)	Element	2 (xs:string)		#Suffix#Name#Resume

Figure 7-8. PROPERTY secondary XML index on node table format data (simplified)

Where the other types of secondary XML indexes are optimized for quickly locating specific nodes, the PROPERTY secondary XML index is optimized for retrieving one or more scalar values from individual xml instances. Listing 7-7 demonstrates a simple query using the xml data type value() method to take advantage of a PROPERTY secondary XML index.

Listing 7-7. *Simple PROPERTY Secondary XML Index Efficiency Sample*

```
SELECT ProductModelID,
       Instructions.value ('declare namespace ns =
       "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
       ProductModelManuInstructions";
       (/ns:root/ns:Location/ns:step/ns:material[. = "aluminum sheet MS-2341"])[1]',
       'varchar(1000)') AS Step,
       Instructions.value ('declare namespace ns =
       "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
       ProductModelManuInstructions";
       (/ns:root/ns:Location/ns:step/ns:tool)[1]', 'varchar(200)') AS Tool
FROM Production.ProductModel
WHERE ProductModelID = 7;
```

The optimizer can take advantage of the PROPERTY secondary XML index to optimize this query because the primary key and path are known ahead of time. The cost for this query is about 0.013, and the query plan is shown in Figure 7-9.

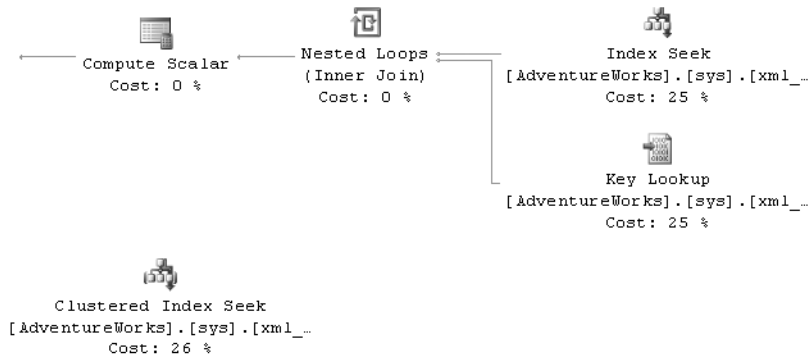


Figure 7-9. *Sample XQuery query plan with PROPERTY secondary XML index (partial)*

The optimization for this query includes a highly efficient Index Seek on the PROPERTY secondary XML index. You may also notice that the query plan includes a Key Lookup operation that adds some overhead, but is still more efficient than no XML index or a primary XML index alone.

USE THEM ALL, AND LET SQL SORT OUT THE MESS?

As you've seen in this section, secondary XML indexes provide additional query efficiency, with each type of secondary XML index providing optimizations for a specific type of query. You can also create multiple secondary XML indexes of different types on a single `xml` column. The question then is why not just create a primary XML index and all three types of secondary XML indexes on your `xml` columns, and give the optimizer as many choices as possible when querying your XML data? Basically this idea is analogous to throwing everything you've got at SQL Server and counting on the query engine to sort out the mess.

As it turns out, the reason you don't want to do this is pretty simple. The secondary XML indexes are simply nonclustered indexes created on your primary XML index. If you have a large number of rows of `xml` data and/or you have very large `xml` data type instances stored in the column, you can end up with very large secondary XML indexes requiring a lot of additional storage space. Remember, each `xml` data type instance can hold over 2 GB of data.

Another important consideration is secondary XML index maintenance. Any time a new row is added to a table containing an `xml` data type column with secondary indexes, or if an existing row is modified, the primary XML index and all secondary XML indexes on the column must be updated as well. This can adversely affect the performance of T-SQL INSERT, UPDATE, MERGE, and DELETE statements as well as XQuery XML Data Manipulation Language (DML) statements.

While giving the optimizer one or more secondary XML indexes to use can increase your XQuery efficiency, that advantage must be weighed against the adverse effects on the performance of SQL DML and XML DML. If, however, your XML data is reasonably static and query efficiency is your only real concern, it might make sense to use multiple secondary XML indexes if your query requirements really demand it. These are some of the concerns that should be addressed as part of an overall XML indexing strategy.

Setting XML Index Options

XML indexes have several options that can be set during XML index creation via CREATE XML INDEX statement options or changed on existing XML indexes via ALTER INDEX statement options. The available options are listed in Table 7-1 with a brief explanation of each.

Table 7-1. XML Index Options

Option	Default	Description
ALLOW_PAGE_LOCKS	ON	This option can be set to ON or OFF. If set to ON, this option allows the database engine to take page locks on the XML index when it is accessed. The database engine controls when page locks are taken on the index.
ALLOW_ROW_LOCKS	ON	This option can be set to ON or OFF. If set to ON, this option allows the database engine to take row locks on the XML index when it is accessed. The database engine controls when row locks are taken on the index.
DROP_EXISTING	OFF	This option can be set to ON or OFF. If set to ON, the existing XML index with the same name is dropped and re-created. You cannot use this option to drop an index and re-create it as another type. For instance, you cannot drop an XML index and re-create it as a relational index or drop a primary XML index and re-create it as a secondary XML index.

Continued

Table 7-1. *(Continued)*

Option	Default	Description
FILLFACTOR	0	This option can be set to an integer value between 0 and 100. The fill factor tells the database engine how full to make leaf levels of the index during index creation and rebuild. The fill factor only applies during index creation and rebuild; the database engine does not attempt to maintain the same levels of empty space at the leaf levels during normal index updates. A fill factor of 0 is the same as a fill factor of 100. Both values indicate that index leaf levels should be filled to capacity.
MAXDOP	0	This option can be set to an integer value between 0 and 64. The MAXDOP option specifies the maximum degree of parallelism used during the XML index CREATE or ALTER statement. Setting this option to 0 allows the database engine to determine how many processors are used up to the number of processors available. Setting MAXDOP to 1 restricts the operation to use one, and only one, processor (no parallelism). Setting the option to a value between 2 and 64 allows the database engine to use up to the number of processors specified in a parallel plan.
PAD_INDEX	OFF	This option can be set to ON or OFF. The PAD_INDEX option works in conjunction with the FILLFACTOR option to specify free space in the intermediate leaf levels of the XML index. If PAD_INDEX is set to ON, the database engine uses the FILLFACTOR value to set intermediate leaf level padding. If PAD_INDEX is set to OFF, or FILLFACTOR is set to 0 or 100, the intermediate leaf levels are filled to near capacity.
SORT_IN_TEMPDB	OFF	This option can be set to ON or OFF. If set to ON, intermediate sort results that are generated during XML index creation are stored in the tempdb database. This can decrease the time required to build an XML index if tempdb is on a different physical device, but it can increase the amount of disk space needed during the index build. If this option is set to OFF, intermediate sort results are stored in the current database.
STATISTICS_NORECOMPUTE	OFF	This option can be set to ON or OFF. If set to ON, out-of-date statistics are not automatically recomputed. If set to OFF, statistics are automatically updated when they go out of date. If you set this option to ON, you will have to manually update statistics on your indexes to ensure optimal query plan generation.

Listing 7-8 shows a selection of these options used in an ALTER INDEX statement that rebuilds the SXML_ProductModel_Instructions_PATH secondary XML index created previously in this section.

Listing 7-8. *Altering an XML Index*

```
ALTER INDEX SXML_ProductModel_Instructions_PATH
ON Production.ProductModel
REBUILD WITH
(
```

```
    SORT_IN_TEMPDB = ON,  
    MAXDOP = 1,  
    FILLFACTOR = 80,  
    PAD_INDEX = ON  
);
```

Full-Text Indexing XML

SQL Server has included the capability to generate and query full-text indexes since version 7.0. In SQL Server 2005, full-text indexing was expanded to include XML content. SQL Server 2008 continues to allow you to create full-text indexes on `xml` type columns using the `CREATE FULLTEXT INDEX` statement.

Full-text indexing allows you flexibility in searching your XML data, since it can apply standard word-breakers, stemmers, and thesaurus functionality to your XML content in a wide variety of languages. Word-breakers apply language-specific rules to divide up your full-text indexed content into word-sized tokens. In the English language, for instance, a space character can be used as a divider between two words. Stemmers are a full-text indexing mechanism that allows you to search for variations of words, “run” for “running” and “ran,” for instance. Full-text indexing also supports thesaurus functionality that allows you to create a language-specific thesaurus to support custom word substitutions and word expansions. As an example, you might create a thesaurus that substitutes the word “red” when the words “crimson,” “scarlet,” or “vermillion” are encountered. While a full treatment of full-text indexing is beyond the scope of this book, I will discuss how to create a full-text index on `xml` data type columns in this section.

SQL Server 2008 supports full-text indexing of XML content with an XML word-breaker. The XML word-breaker allows SQL Server to index individual tokens in your XML data. This word-breaker implements some behaviors specific to XML data, which include the following:

- The XML word-breaker uses XML tags to break apart content in addition to the standard word-breaker characters defined for specific languages (such as punctuation marks, spaces, and so on).
- The XML word-breaker does not index markup information, such as XML tags and attributes, which are considered part of the markup and not the actual content of the XML data.

The first step to full-text indexing an `xml` column is to create a full-text catalog. Listing 7-9 shows the `CREATE FULLTEXT CATALOG` statement used to create a full-text catalog in the AdventureWorks database.

Listing 7-9. *Creating a Full-Text Catalog*

```
CREATE FULLTEXT CATALOG FTC_Instructions_XML;
```

The second step is to create a full-text index on the `xml` data type column. This is demonstrated in Listing 7-10.

Listing 7-10. *Creating a Full-Text Index on an xml Column*

```

CREATE FULLTEXT INDEX
ON Production.ProductModel
(
    Instructions
)
KEY INDEX PK_ProductModel_ProductModelID
ON FTC_Instructions_XML;

```

Tip When you create the full-text index, you must specify a key index column to be used by the full-text index. This must be a single-column, non-nullable, unique index on the table. An integer primary key on the table is used most often, and it is an excellent choice for this function.

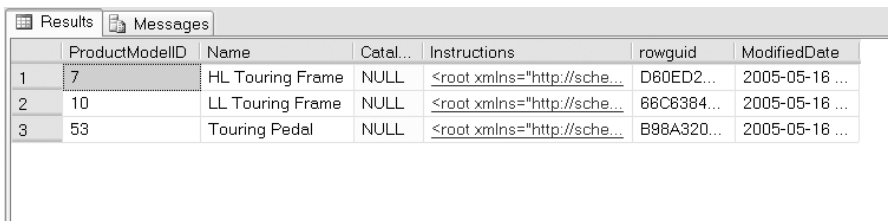
Once you have successfully created a full-text index on your xml column, you can use the T-SQL CONTAINS and FREETEXT predicates (and the CONTAINSTABLE and FREETEXTTABLE functions) to perform full-text searches on your XML data. Listing 7-11 demonstrates a simple FREETEXT full-text search on the Instructions column on which you just created a full-text index. The FREETEXT predicate automatically performs word-stemming and thesaurus lookups so that word variants are located as well. This query returns four rows, as shown in Figure 7-10.

Listing 7-11. *Simple Full-Text Search with FREETEXT Predicate*

```

SELECT *
FROM Production.ProductModel
WHERE FREETEXT (Instructions, 'applies');

```



The screenshot shows a SQL Server query results window with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with 7 columns: ProductModelID, Name, Catal..., Instructions, rowguid, and ModifiedDate. The table contains 3 rows of data. The first row (rowid 1) has ProductModelID 7, Name 'HL Touring Frame', Catal... NULL, and Instructions '<root xmlns="http://sche...'. The second row (rowid 2) has ProductModelID 10, Name 'LL Touring Frame', Catal... NULL, and Instructions '<root xmlns="http://sche...'. The third row (rowid 3) has ProductModelID 53, Name 'Touring Pedal', Catal... NULL, and Instructions '<root xmlns="http://sche...'. The 'rowguid' column contains values like 'D60ED2...', '86C8384...', and 'B98A320...'. The 'ModifiedDate' column shows '2005-05-18 ...' for all rows.

	ProductModelID	Name	Catal...	Instructions	rowguid	ModifiedDate
1	7	HL Touring Frame	NULL	<root xmlns="http://sche...	D60ED2...	2005-05-18 ...
2	10	LL Touring Frame	NULL	<root xmlns="http://sche...	86C8384...	2005-05-18 ...
3	53	Touring Pedal	NULL	<root xmlns="http://sche...	B98A320...	2005-05-18 ...

Figure 7-10. *Results of full-text search on Instructions column*

If you were to look at the raw XML data of the four rows returned by the full-text FREETEXT search performed in Listing 7-11, you would notice that the word “applies” does not appear anywhere in the XML content. The word “Apply” does, however, appear within the XML content. Since the FREETEXT predicate automatically stems the search word to match word variants, the end result is the four matches shown in Figure 7-10.

Full-text search predicates and functions can be used in conjunction with the xml data type methods like query() and value(). When used together, the full-text search predicates are used to narrow down the result set before the xml data type methods are applied. Listing 7-12

modifies the example in Listing 7-11 to eliminate those that don't contain the word “aluminum” in the material node.

Listing 7-12. *Full-Text Search Combined with xml Data Type Predicate*

```
SELECT *
FROM Production.ProductModel
WHERE FREETEXT(Instructions, 'applies')
AND Instructions.exist('declare namespace ns =
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
ProductModelManuInstructions";
(//ns:step/ns:material[contains(., "aluminum")])') = 1;
```

This example returns three rows that match the following criteria:

- The XML data contains a variant of the word “applies” anywhere within its content.
- The XML data has a material node containing the word “aluminum.”

The results are shown in Figure 7-11.

	ProductModelID	Name	Cat...	Instructions	rowguid	ModifiedDate
1	7	HL Touring Frame	NULL	<root xmlns="..."	D60ED...	2005-05-18...
2	10	LL Touring Frame	NULL	<root xmlns="..."	68C63...	2005-05-18...

Figure 7-11. *Results of full-text search combined with xml method*

FULL-TEXT CONTAINS VS. XQUERY CONTAINS

The full-text search CONTAINS predicate is not the same as the XQuery contains predicate. The XQuery contains predicate performs a substring match similar to the T-SQL CHARINDEX function. The matches performed by the XQuery contains predicate are case sensitive.

The T-SQL CONTAINS predicate, on the other hand, includes all of the flexibility of the SQL Server full-text search functionality. This includes the ability to perform thesaurus lookups, word stemming, and proximity searches.

The downside to the T-SQL CONTAINS predicate is that you cannot specify node paths to narrow your search using them. The T-SQL full-text search is an all-or-nothing proposition—if you want to search for a word in your XML data using a full-text search, the word can appear anywhere in the XML content. This is why it makes sense to use the T-SQL CONTAINS predicate in conjunction with the XQuery contains predicate for maximum flexibility and performance.

Summary

SQL Server provides a good deal of XML and XQuery functionality. As discussed in Chapter 5, the SQL Server XQuery implementation leverages the power of the SQL Server relational query engine to a great extent. Optimizing performance of XQuery queries on your `xml` columns can rely to a large degree on your XML indexing strategy.

In this chapter I discussed primary XML indexes and the XQuery performance gains you can achieve by using them. I also discussed the three specific types of secondary XML indexes—PATH, VALUE, and PROPERTY secondary XML indexes—and the query types they are designed to optimize.

Of course, greater efficiency is not free, and there is always a trade-off. In the case of XML indexes, I discussed the SQL DML and XML DML performance penalties as well as the greater additional storage requirements.

Finally, I discussed the powerful `xml` full-text indexing capabilities in SQL Server with examples of full-text index creation, full-text searches, and combinations of full-text search and `xml` data type methods.

In the next chapter, I will discuss using SQLCLR (Common Language Runtime) to invoke Extensible Stylesheet Language Transformations (XSLT) directly from your T-SQL code.



XSLT and the SQLCLR

So far I've covered quite a lot of ground discussing native SQL Server 2008 XML functionality. However, there are some things that aren't covered by the native Transact SQL (T-SQL) and `xml` data type enhancements in SQL Server. Using Extensible Stylesheet Language Transformations (XSLT) functionality is one area where you have to go outside of the built-in T-SQL functionality and access the .NET functionality available through the SQLCLR (SQL Common Language Runtime).

XSLT is a W3C recommendation for a language for transforming XML documents into other XML documents. In this chapter, I will talk about the W3C XSLT recommendation and the SQLCLR functionality that will enable you to access XSLT functionality directly from within SQL Server.

XSLT gives you the ability to take an XML document as input, add or remove attributes, rearrange and sort elements, manipulate content, and make processing decisions based on the results of node tests and other logical constructs. One of the most common uses for XSLT is to transform XML documents into XHTML for presentation on the client tier. Another common use in the back end is to take XML data from different sources and convert it to a common format. I'll discuss both of these uses for XSLT and demonstrate performing XSL transformations directly from T-SQL in this chapter.

Transforming XML

During the XSL transformation process, an input XML document and an XSLT stylesheet are both fed into an XSLT processor. The XSLT processor uses the XSLT stylesheet to manipulate the contents of the XML document fed into it to produce the XML result document. The original source XML document is left unaltered by the process, with all transformations being output to a new XML document. This process is shown in Figure 8-1.

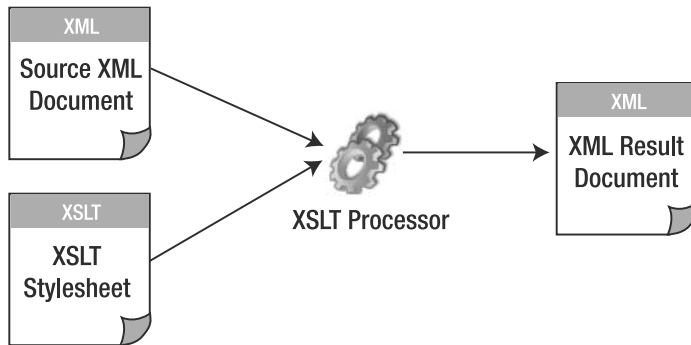


Figure 8-1. *XSLT processing overview*

XSLT uses XPath to locate, navigate, and perform node tests in the source document. XPath is also the foundation for XQuery path expressions, which I talked about in Chapter 5. XSLT is a functional language, and XSLT stylesheets are XML documents that are composed of templates. XSLT stylesheet templates, in turn, are built from predefined XSLT elements.

I will talk about how to create your own XSLT stylesheets in this chapter, but first I will address a more immediate issue—accessing XSLT from T-SQL.

Accessing XSLT Through .NET

As I mentioned previously, T-SQL does not provide built-in support for XSLT. The .NET Framework, however, offers an excellent XSLT transformation facility via its `System.Xml.Xsl` namespace. Fortunately, you can easily access this .NET Framework functionality through SQL Server's SQLCLR integration. This requires you to create and register a .NET assembly and create T-SQL functions and procedures to access the functionality exposed by the assembly. Before I get into the details of SQLCLR assembly creation and deployment, I need to do a little administrative work to ensure SQLCLR is enabled. Listing 8-1 enables SQLCLR functionality in the AdventureWorks database.

Listing 8-1. *Enable SQLCLR in the Database*

```
USE AdventureWorks;
GO
EXEC sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
EXEC sp_configure 'clr enabled', 1;
GO
RECONFIGURE;
GO
```

I will also be implementing some functionality that creates XML files in the file system. SQLCLR assemblies need the external access level of security in order to access the file system. To allow this functionality, you need to make sure the database is set to TRUSTWORTHY mode, as shown in Listing 8-2.

Listing 8-2. *Turn on Database Trustworthy Mode*

```
ALTER DATABASE AdventureWorks  
SET TRUSTWORTHY ON;
```

The user installing the assembly must also have `EXTERNAL ACCESS ASSEMBLY` permissions. The statement to grant these permissions is shown in Listing 8-3.

Listing 8-3. *Granting External Access Assembly*

```
USE master;  
GRANT EXTERNAL ACCESS ASSEMBLY  
TO principal; -- specify the name of a principal to grant access to here  
GO
```

DEPLOYING SQLCLR ASSEMBLIES

After you've enabled the SQLCLR option in the database, set the database to TRUSTWORTHY mode, and granted external access assembly permissions to the principal, as described in this section, it's time to compile and deploy the assembly. The source code for the assemblies used in the examples in this book is included in the download samples for this book, available from the Apress web site at www.apress.com/book/sourcecode. To compile and deploy a SQLCLR sample, open the appropriate Visual Studio solution file in Visual Studio. Once you have opened the solution, you will need to change the database connection string in the database properties page of the solution. Note that if you are using Visual Studio 2005, deploying to SQL Server 2008 requires you to download and install a patch from Microsoft's web site at <http://download.microsoft.com>.

Once you've changed the database connection string for the assembly, choose the Deploy option on the Build menu. This option compiles the code, installs the assembly on the server, and creates the T-SQL user-defined functions and stored procedures necessary to access the SQLCLR functionality. Once you've deployed the code to the server, you can access the procedures and functions through SQL Server Management Studio (SSMS).

Alternatively, the T-SQL scripts to install the precompiled assemblies, functions, and procedures are included with the sample downloads. This is useful if you want to install the sample SQLCLR code without compiling and deploying it yourself through Visual Studio.

In the example for this section, deploying the sample code through Visual Studio or running the T-SQL installation scripts in SSMS will install the SQLCLR assembly, procedures, and functions, including `fn_XsltTransform` and `p_XsltTransformToFile`.

The assembly itself is written in C# and compiled with Visual Studio 2005. I'll start by creating some private functions that support .NET XSL transformation, as shown in Listing 8-4. The SQLCLR user-defined functions and stored procedures I'll create in this section will rely heavily on these support functions.

Listing 8-4. *SQLCLR Transform Function*

```

using System.Data.SqlTypes;
using System.Text;
using System.Xml;
using System.Xml.Xsl;
using System.IO;

namespace Apress.Samples
{
    public partial class XsltAssembly
    {
        private static XmlReader CreateXmlReader(SqlXml source_xml)
        {
            byte[] xmldata = Encoding.UTF8.GetBytes(source_xml.Value);
            MemoryStream stream = new MemoryStream(xmldata);
            return XmlReader.Create(stream);
        }

        private static XslCompiledTransform CreateXsltStylesheet (SqlXml source_xslt)
        {
            XslCompiledTransform xslt_stylesheet = new XslCompiledTransform();
            xslt_stylesheet.Load(CreateXmlReader(source_xslt));
            return xslt_stylesheet;
        }

        private static MemoryStream ApplyStylesheet(XmlReader source_xml_reader,
            XslCompiledTransform xslt_stylesheet)
        {
            MemoryStream buffer = new MemoryStream();
            StreamWriter stream = new StreamWriter(buffer);
            xslt_stylesheet.Transform(source_xml_reader, XmlWriter.Create(stream));
            return buffer;
        }

        private static SqlXml Transform(SqlXml source_xml,
            SqlXml source_xslt)
        {
            XmlReader source_xml_reader = CreateXmlReader(source_xml);
            XslCompiledTransform xslt_stylesheet = CreateXsltStylesheet(source_xslt);
            return new SqlXml(ApplyStylesheet(source_xml_reader, xslt_stylesheet));
        }
    }
}

```

Each of the private support functions in Listing 8-4 provides a small piece of the total functionality required to perform XSL transformations in .NET:

- The `CreateXmlReader` function accepts a SQL Server `xml` parameter (.NET `SqlXml` data type) and returns a .NET `XmlReader`.
- The `CreateXsltStyleSheet` function takes a SQL Server `xml` parameter and creates and loads a .NET `XslCompiledTransform` with it.
- The `ApplyStylesheet` function accepts an `XmlReader` and an `XslCompiledTransform` (created by the first two functions), applies the XSL transformation, and returns a .NET `MemoryStream` containing the result.
- The `Transform` function takes two `SqlXml` parameters, the source XML document, and the source XSLT stylesheet. This function calls the other functions to create the required `XmlReader`, `XslCompiledTransform`, and perform the actual transformation.

Note In this chapter, I'm covering some new ground by taking advantage of SQLCLR integration. I'm covering a very specific use case here, but a full discussion of the intricacies of SQLCLR programming is beyond the scope of this book. The book *Pro SQL Server 2005 Assemblies* by Robin Dewson and Julian Skinner (Apress, 2005) is an excellent resource for learning the details of SQLCLR programming.

There are a couple items of note in Listing 8-4. First, it's important to notice that I use the `SqlXml` data type, which is the .NET equivalent of the SQL Server `xml` data type. Second, I'm using the `XslCompiledTransform.NET 2.0` class, which provides an efficient implementation of the XSLT 1.0 standard. For those who have implemented XSLT solutions in .NET 1.1, keep in mind that `XslCompiledTransform` completely replaces the .NET 1.1 `XsltTransform` class.

The assembly I create will provide two public methods that can be accessed directly from T-SQL. The first is a function called `fn_XsltTransform`, which accepts an `xml` source document and an `xml` XSLT stylesheet as parameters. The function returns the `xml` result of applying the XSLT stylesheet to the source XML. A common use for this function might be applying a stylesheet to XML data stored in, or generated by, SQL Server and transferring the resultant XML to the application/user interface layer. The C# source for this function is shown in Listing 8-5.

Listing 8-5. *fn_XsltTransform Source Listing*

```
using System.Data.SqlTypes;

namespace Apress.Samples
{
    public partial class XsltAssembly
    {
        [Microsoft.SqlServer.Server.SqlFunction]
        public static SqlXml fn_XsltTransform(SqlXml source_xml,
            SqlXml source_xslt)
        {
```

```

        SqlXml result = new SqlXml();
        if (source_xml.IsNull ||
            source_xslt.IsNull)
            result = SqlXml.Null;
        else
            result = Transform(source_xml, source_xslt);
        return result;
    }
}
}

```

The second public method exposes a stored procedure called `p_XsltTransformToFile`. This procedure accepts an xml source document, an xml XSLT stylesheet, and an output file name. Like the `fn_XsltTransform` function, this procedure applies the XSLT stylesheet to the source XML you provide. The difference is that the `p_XsltTransformToFile` procedure saves the result to an output file you specify. A common use for this function might be a generation of XHTML or other XML files from data stored in, or generated by, SQL Server. Listing 8-6 is the C# source for the `p_XsltTransformToFile` procedure.

Listing 8-6. *p_XsltTransformToFile Source Listing*

```

using System.Data.SqlTypes;
using System.Xml;
using System.IO;

namespace Apress.Samples
{
    public partial class XsltAssembly
    {
        [Microsoft.SqlServer.Server.SqlProcedure]
        public static void p_XsltTransformToFile(SqlXml source_xml,
            SqlXml source_xslt,
            SqlString output_file)
        {
            SqlXml result = new SqlXml();
            if (!source_xml.IsNull &&
                !source_xslt.IsNull &&
                !output_file.IsNull)
            {
                result = Transform(source_xml, source_xslt);
                StreamWriter sw = new StreamWriter(output_file.Value);
                sw.Write(result.Value);
                sw.Dispose();
            }
        }
    };
}

```

The `fn_XsltTransform` function and `p_XsltTransformToFile` procedure give you the basic tools you need to explore XSL transformations in SQL Server. Next I'll discuss the W3C XSLT recommendation and some SQL Server–based XSLT examples.

Performing a Simple Transformation

For the first XSL transformation, I will generate a simple HTML file with a summary report of AdventureWorks customers by state. I'll start by showing the demonstration code in Listing 8-7 and then look at the results and discuss the fine points of the transformation.

Listing 8-7. *AdventureWorks XSLT Customer Summary Report*

```
DECLARE @source_xslt xml,
        @source_xml xml;

SELECT @source_xml =
(
    SELECT ic1.CountryRegionName AS "Country",
        (
            SELECT COUNT(*) AS "Count",
                ic2.StateProvinceName AS "State-Name"
            FROM Sales.vIndividualCustomer ic2
            WHERE ic2.CountryRegionName = ic1.CountryRegionName
            GROUP BY ic2.CountryRegionName, ic2.StateProvinceName
            FOR XML PATH ('State'), TYPE
        ) AS "Customers"
    FROM Sales.vIndividualCustomer ic1
    WHERE ic1.CountryRegionName = 'United States'
    GROUP BY CountryRegionName
    ORDER BY CountryRegionName
    FOR XML PATH ('Individual-Customer-Summary'), TYPE
);

SET @source_xslt = N'<xsl:stylesheet version = "1.0"
    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
    xmlns = "http://www.w3.org/1999/xhtml">

    <xsl:template match = "/Individual-Customer-Summary">
        <html>

            <head>
                <style type = "text/css">

                    table {
                        border-width: 1px;
                        border-style: solid;
                        border-color: black;
                        border-collapse: collapse;
```

```

        background-color: white;
        width: 50%;
    }

    table th {
        border-width: 1px;
        padding: 3px;
        border-style: dotted;
        border-color: gray;
        background-color: #6666dd;
        font-family: arial;
        font-size: 12px;
        color: white;
    }

    table td.light {
        border-width: 1px;
        padding: 3px;
        border-style: dotted;
        border-color: gray;
        background-color: white;
        font-family: arial;
        font-size: 12px;
    }

    table td.dark {
        border-width: 1px;
        padding: 3px;
        border-style: dotted;
        border-color: gray;
        background-color: #66ffff;
        font-family: arial;
        font-size: 12px;
    }
}

</style>
</head>

<body>
    <h2>
        AdventureWorks Customer Breakdown: <xsl:value-of select = "Country"/>
    </h2>
    <table>
        <tr>
            <th>
                State
            </th>

```

```

        <th>
            Customer Count
        </th>
    </tr>
    <xsl:for-each select="Customers/State">
        <xsl:sort select="Count" data-type="number" order="descending"/>
        <xsl:sort select="State-Name" order="ascending"/>
        <tr>
            <xsl:choose>
                <xsl:when test = "position() mod 2 = 0">
                    <td class = "dark">
                        <xsl:value-of select = "State-Name" />
                    </td>
                    <td class="dark">
                        <xsl:value-of select = "Count"/>
                    </td>
                </xsl:when>
                <xsl:otherwise>
                    <td class="light">
                        <xsl:value-of select = "State-Name" />
                    </td>
                    <td class="light">
                        <xsl:value-of select = "Count"/>
                    </td>
                </xsl:otherwise>
            </xsl:choose>
        </tr>
    </xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>';

```

```

EXEC p_XsltTransformToFile @source_xml,
    @source_xslt,
    N'C:\customer_report.html';

```

The code begins by declaring two variables to hold the source XSLT stylesheet and the source XML document. For the next step, I use nested FOR XML PATH queries to convert the source relational data to XML format and assign the result to the @source_xml variable. A snippet of the source XML document generated by this query is shown in Figure 8-2.



Figure 8-2. Source XML generated by nested FOR XML PATH

Then I assign the XSLT stylesheet to the @source_xslt variable and call the p_XsltTransformToFile procedure. The result of the XSL transformation is a file called customer_report.html, shown in Figure 8-3.



State	Customer Count
California	4445
Washington	2285
Oregon	1073
Illinois	6
Ohio	4
Texas	4
Florida	3
Georgia	3
New York	3
Arizona	2
Utah	2
Wyoming	2
Alabama	1
Kentucky	1
Maryland	1
Massachusetts	1
Minnesota	1
Mississippi	1
Missouri	1
Montana	1
North Carolina	1
South Carolina	1
Virginia	1

Figure 8-3. *customer_report.html* sample XSL transformation result

Elements of XSLT Stylesheets

So how did I go from the source XML document to the final HTML file result in this example? For this answer, I'll discuss the sample XSLT stylesheet and the transformations it specifies in detail. The stylesheet begins with an `xsl:stylesheet` element, which is the root element for every XSLT stylesheet. This example stylesheet contains a couple of namespace declarations in the root element. The first namespace declaration defines the `xsl` namespace, and the second declares the default namespace for XHTML.

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
  xmlns = "http://www.w3.org/1999/xhtml">
  ...
</xsl:stylesheet>
```

Tip There is an element called `xsl:transform`, which is an alias for the `xsl:stylesheet` element. You can use either element as the root element for your XSLT stylesheets. For simplicity, I'll use `xsl:stylesheet` exclusively in my code samples.

Immediately after the root `xsl:stylesheet` element is an `xsl:template` element. Templates are the building blocks of XSLT stylesheets. Every stylesheet must have at least one template. The `match` attribute used with the `xsl:template` element specifies an XPath expression to associate with the template.

```
<xsl:template match = "/Individual-Customer-Summary">
    ...
</xsl:template>
```

This stylesheet is simple, with just one template. I will be outputting HTML, so I need to wrap the output with proper HTML tags, like `html`, `style`, `body`, and so on. I'm also going to be a good Web citizen and use the modern W3C Cascading Stylesheets (CSS) recommendation for the HTML formatting instead of the old-style attribute styles. The `style` section in the HTML head section contains all the various formatting styles I'll use, like the table heading styles, table borders, table cell colors, and so on. The styles include two different cell styles, one with a white background and one with a slightly darker background, which I'll use to alternate row colors in the final result. This entire section of HTML is copied directly into the result file with no manipulation by the XSLT processor.

```
<html>

<head>
  <style type = "text/css">

    table {
      border-width: 1px;
      border-style: solid;
      border-color: black;
      border-collapse: collapse;
      background-color: white;
      width: 50%;
    }

    table th {
      border-width: 1px;
      padding: 3px;
      border-style: dotted;
      border-color: gray;
      background-color: #6666dd;
      font-family: arial;
      font-size: 12px;
      color: white;
    }
  </style>
</head>

<body>
```

```
table td.light {
    border-width: 1px;
    padding: 3px;
    border-style: dotted;
    border-color: gray;
    background-color: white;
    font-family: arial;
    font-size: 12px;
}

table td.dark {
    border-width: 1px;
    padding: 3px;
    border-style: dotted;
    border-color: gray;
    background-color: #66ffff;
    font-family: arial;
    font-size: 12px;
}

</style>
</head>
...
</html>
```

HTML, CSS, AND XHTML

In 1999, the W3C approved the HTML 4.01 specification for web-based publishing (this recommendation is commonly known as HTML4). In 2000, the W3C quickly followed up with the XHTML (Extensible HTML) standard, which redefines HTML as an XML application. During this same time, vendors were just starting to get serious about implementing the 1996 W3C recommendation for CSS functionality in their browser products.

Historically speaking, attributes played a key role in HTML formatting. All the way up to HTML4, there is a heavy reliance on attributes to specify colors, borders, spacing, position, and just about every other formatting option supported by HTML. With the adoption of XHTML, most of these attributes were deprecated in favor of the more powerful and flexible CSS model. In an attempt to follow modern user interface coding standards, I've used CSS and generated properly formed HTML in the examples of this chapter. All HTML results have been tested for standards conformance in both Internet Explorer 6 and Firefox 2.0.

The HTML body contains the first XSL-applied transformation, which is an `xsl:value-of` element in the header. The `select` attribute can take a literal value or an XPath expression, which provides the value to insert into the result document. In this instance, I specified the `Country` element from the source XML document. Recall that I specified the `/Individual-Customer-Summary` XPath expression for this template's `match` attribute expression, so the path that I specify in this `xsl:value-of` element is relative to `/Individual-Customer-Summary`. This

means the full path to this specific element is actually `/Individual-Customer-Summary/Country`. I also define the two-column HTML table that will display the formatted results.

```
<body>
  <h2>
    AdventureWorks Customer Breakdown: <xsl:value-of select = "Country"/>
  </h2>
  <table>
    <tr>
      <th>
        State
      </th>
      <th>
        Customer Count
      </th>
    </tr>
    ...
  </table>
  ...
</body>
```

Note If you want to use string literals in the `select` attribute of the `xsl:value-of` element, enclose it in apostrophes within the double quotes. In this example, you could hard-code a country name in there like this:

```
<xsl:value-of select = "&apos;United States&apos;" />
```

You can use the `'` entity like I did in this example, or you can use regular single quotes (don't forget to double them up to escape them in SQL code). Don't worry about the extra quotes within quotes if you're using numeric literals, since XSLT assumes a number in the `select` attribute automatically means you want to assign a numeric literal.

The engine that drives the XSLT results is the `xsl:for-each` element. The `xsl:for-each` element loops over all source XML elements that match the `select` attribute XPath expression. In this case, you want all `Customers/State` elements (remember, I start with the `/Individual-Customer-Summary` element, which is the root I defined for this template).

```
<xsl:for-each select="Customers/State">
  ...
</xsl:for-each>
```

Nested immediately inside the `xsl:for-each` element I've included two `xsl:sort` elements. The `xsl:sort` element defines the sort order for the elements I'm looping over. The effects of the `xsl:sort` elements are cumulative. In this case, I sort the results by `Count` in descending order, and

then by State-Name in ascending order. This means the results are sorted in reverse order by customer counts (largest numbers at the top), and then in alphabetical order by state name. Notice that I use the data-type attribute and specify number for the Count sort. If I don't do this, the Count sort order will be alphabetical instead of numeric, and I'll end up with strange results like the number 2 following the number 1073, which could really confuse users.

```
<xsl:sort select="Count" data-type="number" order="descending"/>
<xsl:sort select="State-Name" order="ascending"/>
```

Note Valid values for the `xsl:sort data-type` attribute are text, number, and QName. The default is text, which gives very strange results if you're trying to sort numbers. The `order` attribute can be either ascending or descending.

The next portion of the XSLT template formats and outputs rows, demonstrating the `xsl:choose` element along the way. In this case, I'm using `xsl:choose` with the `position` function to alternate row styles (background color specifically) in the output. You might recall the `position` function from the discussion of XQuery in Chapter 6. This function returns an integer value indicating the current context position in the source document. It returns 1 for the first matching node, 2 for the second match, and so on.

```
<tr>
  <xsl:choose>
    <xsl:when test = "position() mod 2 = 0">
      <td class = "dark">
        <xsl:value-of select = "State-Name" />
      </td>
      <td class = "dark">
        <xsl:value-of select = "Count"/>
      </td>
    </xsl:when>
    <xsl:otherwise>
      <td class = "light">
        <xsl:value-of select = "State-Name" />
      </td>
      <td class = "light">
        <xsl:value-of select = "Count"/>
      </td>
    </xsl:otherwise>
  </xsl:choose>
</tr>
```

XSL:CHOOSE

The `xsl:choose` element is analogous to the SQL searched CASE expression. This element contains one or more `xsl:when` elements that are analogous to WHEN clauses in SQL searched CASE expressions. A SQL searched CASE expression is one where each WHEN clause is a predicate. Like the SQL searched CASE expression, each `xsl:when` element takes a test attribute that contains an expression. If the expression for an `xsl:when` element evaluates to true, the content of that element is used. The `xsl:choose` element can also contain an `xsl:otherwise` element which is equivalent to the CASE expression's ELSE clause. If all the `xsl:when` elements evaluate to false, the content of the `xsl:otherwise` element is used. XSLT does not require that your expression be a true Boolean expression. Any expression in XSLT can evaluate to an effective Boolean value. Chapter 6 has a discussion of effective Boolean value in XQuery.

This first sample was fairly simple, but it covered a good deal of the main functionality of XSLT stylesheets. In the next example, I'll cover more details of XSLT syntax and demonstrate a back-end transformation.

Performing a Back-End Transformation

This next example will demonstrate performing what I call a “back-end” XSL transformation. While technically all XSL transformations performed on SQL Server occur in the back end, in this instance, I'm referring to the target of the transformation. The most common type of XSL transformation is probably transforming XML documents to a presentation-layer format like HTML, but XSLT is also useful for taking XML data in varying formats and transforming it to a common format for back-end use.

Consider a situation where you receive customer orders in XML format from different sources. These sources might be different web sites that sell your products, brick-and-mortar retailers, government agencies, or combinations of these. Trying to convince all of your customers to change their ordering systems to send you orders in a common format might be an uphill battle, but that's OK because you can transform the data you receive into your own standard format using XSLT. For this example, I'll assume that AdventureWorks accepts XML orders from customers. I'll also assume that those XML orders can arrive in different XML formats, but the back-end processing requires a single common format. Figure 8-4 illustrates this process.

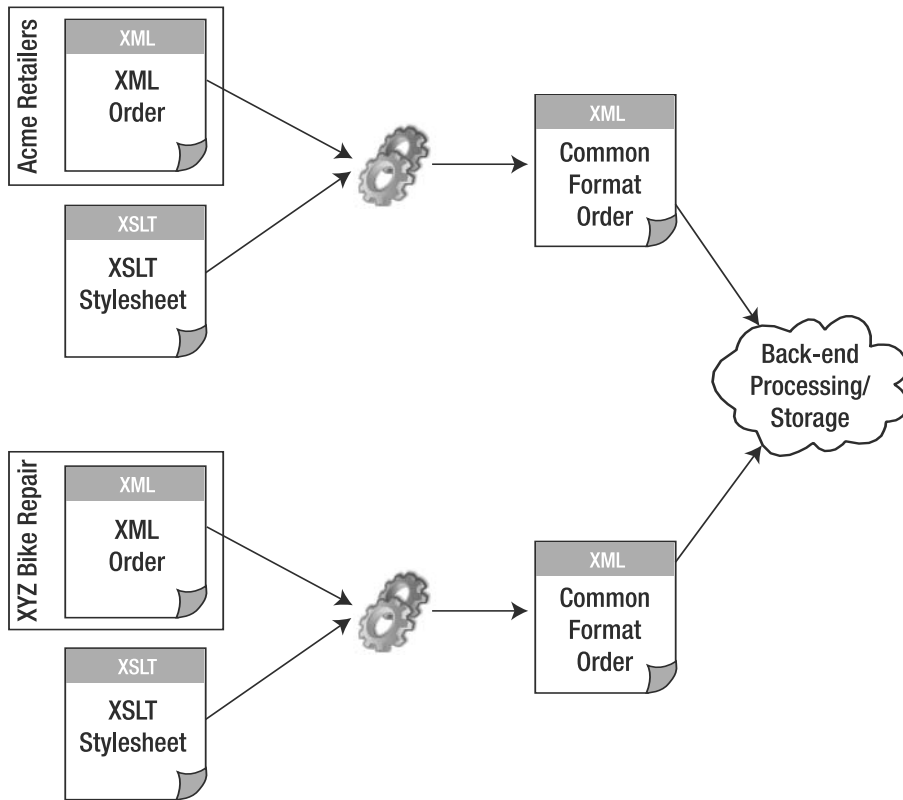


Figure 8-4. *Transforming different orders to a common XML format*

In the old days when ANSI X12 Electronic Data Interchange (EDI) was the only way to transfer electronic orders, buyers and sellers essentially created their own custom data structures by picking and choosing a seemingly random set of records from a very large all-encompassing standard. To deal with all the different source data structures, you either had to create multiple complex parsers by hand or buy expensive software that mapped EDI transaction records to output files and database fields.

XSLT, on the other hand, allows you to quickly and easily create stylesheets that convert XML source data from different sources into a common format. In the example for this section, I'm going to convert orders from two different customers into a common structure using XSLT. I'll begin by creating two XML orders in Listing 8-8, one from a customer called Acme Retailers and a second from XYZ Bike Repair. These orders have different structures that I'll eventually convert to the common format.

Listing 8-8. *XML Orders from Different Sources*

```

DECLARE @acme_retailers_order xml,
        @acme_xslt xml,
        @xyz_bike_repair_order xml,
        @xyz_xslt xml,
        @common_format_order xml;

SET @xyz_bike_repair_order = N'<retail-order>
  <dates order = "2003-08-17T00:00:00" due = "2003-08-29T00:00:00" />
  <account>10-4030-024952</account>
  <purchase-order>P0938345</purchase-order>
  <subtotal>71.5800</subtotal>
  <taxamount>5.7264</taxamount>
  <shipping>1.7895</shipping>
  <total>79.0959</total>
  <contact firstname = "Amanda" lastname = "Long" middlename = "A"
    emailaddress = "amanda31@adventure-works.com" emailopt = "2"
    phone = "328-555-0124" />
  <bill-to-address line1 = "10 Napa Ct."
    city = "Lebanon"
    postal = "97355"
    state = "OR" />
  <ship-to-address same-as-billing = "YES" />
  <order-items>
    <line id = "1">
      <item>TI-R982</item>
      <quantity>1</quantity>
      <price-per-unit>32.6000</price-per-unit>
    </line>
    <line id = "2">
      <item>TT-R982</item>
      <quantity>1</quantity>
      <price-per-unit>3.9900</price-per-unit>
    </line>
    <line id = "3">
      <item>HL-U509-B</item>
      <quantity>1</quantity>
      <price-per-unit>34.9900</price-per-unit>
    </line>
  </order-items>
</retail-order>';

SET @acme_retailers_order = N'<Order>
  <Order-Date>2002-03-01T00:00:00</Order-Date>
  <Due-Date>2002-03-13T00:00:00</Due-Date>
  <PO-Num>P09802194761</PO-Num>
  <Account-Number>10-4020-000026</Account-Number>

```



```

<Subtotal>4659.3105</Subtotal>
<Tax>372.7448</Tax>
<Freight>116.4828</Freight>
<TotalDue>5148.5381</TotalDue>
<Customer>
  <First-Name>James</First-Name>
  <Last-Name>Hamilton</Last-Name>
  <Middle-Name>R.</Middle-Name>
  <Email-Address>james7@adventure-works.com</Email-Address>
  <Email-Promotion>0</Email-Promotion>
  <Phone>418-555-0115</Phone>
  <Billing-Address>
    <Street-Address>1 Corporate Center Drive</Street-Address>
    <City>Miami</City>
    <ZIP>33127</ZIP>
    <State>FL </State>
  </Billing-Address>
  <Shipping-Address>
    <Street-Address>937 West Palm Court</Street-Address>
    <City>Miami</City>
    <ZIP>33127</ZIP>
    <State>FL </State>
  </Shipping-Address>
</Customer>
<Order-Details>
  <Detail>
    <Qty>1</Qty>
    <Catalog-Num>BK-R50R-44</Catalog-Num>
    <Price>419.4589</Price>
  </Detail>
  <Detail>
    <Qty>2</Qty>
    <Catalog-Num>BK-R50B-52</Catalog-Num>
    <Price>419.4589</Price>
  </Detail>
  <Detail>
    <Qty>3</Qty>
    <Catalog-Num>BK-R68R-52</Catalog-Num>
    <Price>874.7940</Price>
  </Detail>
</Order-Details>
</Order>';

```

The XML orders received from customers contain similar information, including customer billing and shipping addresses, order line items, and other relevant information. The end result will be that all Acme Retailers orders and all XYZ Bike Repair orders will be transformed to a common XML format, as shown in Figure 8-5.



Figure 8-5. Customer orders in common XML format

As shown in Listing 8-8, orders from different customers are sent to AdventureWorks in completely different formats. XYZ Bike Repair took advantage of XML attributes in their XML order structure. Acme Retailers, on the other hand, took a different approach and avoided XML attributes, choosing an element-centric format instead. Listing 8-9 shows the XSLT stylesheet used for the Acme Retailers order format.

Listing 8-9. Acme Retailers XSLT Stylesheet

```

SET @acme_xslt = N'<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

  <xsl:strip-space elements = "*" />
  <xsl:output method = "xml" encoding = "utf-16" />

  <xsl:template match = "/Order">
    <common-format-order>
      <date-info>
        <ordered>

```

```
<xsl:value-of select = "Order-Date" />
</ordered>
<due>
  <xsl:value-of select = "Due-Date" />
</due>
</date-info>
<account-info>
  <account-number>
    <xsl:value-of select = "Account-Number" />
  </account-number>
  <purchase-order-number>
    <xsl:value-of select = "PO-Num" />
  </purchase-order-number>
  <billing-info>
    <name>
      <first>
        <xsl:value-of select = "Customer/First-Name" />
      </first>
      <middle>
        <xsl:value-of select = "Customer/Middle-Name" />
      </middle>
      <last>
        <xsl:value-of select = "Customer/Last-Name" />
      </last>
    </name>
    <email>
      <xsl:attribute name = "opt-in">
        <xsl:choose>
          <xsl:when test = "Customer/Email-Promotion = 0">No</xsl:when>
          <xsl:otherwise>Yes</xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>
      <xsl:value-of select = "Customer/Email-Address" />
    </email>
    <phone>
      <xsl:value-of select = "Customer/Phone" />
    </phone>
    <address>
      <street-address>
        <xsl:value-of select = "Customer/Billing-Address/Street-Address" />
      </street-address>
      <city>
        <xsl:value-of select = "Customer/Billing-Address/City" />
      </city>
      <state>
        <xsl:value-of select = "Customer/Billing-Address/State" />
      </state>
    </address>
  </billing-info>
</account-info>
</date-info>
</ordered>
```

```

        <zip>
            <xsl:value-of select = "Customer/Billing-Address/ZIP" />
        </zip>
    </address>
</billing-info>
<shipping-info>
    <address>
        <street-address>
            <xsl:value-of select = "Customer/Shipping-Address/Street-Address" />
        </street-address>
        <city>
            <xsl:value-of select = "Customer/Shipping-Address/City" />
        </city>
        <state>
            <xsl:value-of select = "Customer/Shipping-Address/State" />
        </state>
        <zip>
            <xsl:value-of select = "Customer/Shipping-Address/ZIP" />
        </zip>
    </address>
</shipping-info>
</account-info>
<order-info>
    <xsl:for-each select = "Order-Details/Detail">
        <detail-line>
            <item>
                <xsl:value-of select = "Catalog-Num" />
            </item>
            <quantity>
                <xsl:value-of select = "Qty" />
            </quantity>
            <price>
                <xsl:value-of select = "Price" />
            </price>
        </detail-line>
    </xsl:for-each>
    <summary>
        <subtotal>
            <xsl:value-of select = "Subtotal" />
        </subtotal>
        <tax>
            <xsl:value-of select = "Tax" />
        </tax>
        <shipping>
            <xsl:value-of select = "Freight" />
        </shipping>
        <total>

```

```

        <xsl:value-of select = "TotalDue" />
    </total>
</summary>
    <xsl:comment>TERMS: 30 NET 10, 60 NET 5</xsl:comment>
</order-info>
</common-format-order>
</xsl:template>
</xsl:stylesheet>';

```

In the preceding example, I've added a couple of elements to the start of the XSLT stylesheet outside of the `xsl:template` element. The `xsl:strip-space` element takes an `elements` attribute that contains a space-separated list of elements to strip extra white space from. In this case, I've used the wildcard (*) indicator to tell XSLT to strip extra white space from all elements in the output. I'm also using the `xsl:output` element to specify the output method and encoding, as shown here:

```

<xsl:strip-space elements = "*" />
<xsl:output method = "xml" encoding = "utf-16" />

```

Tip XSLT also supports the `xsl:preserve-space` element, which preserves white space for a specified list of elements. The `xsl:preserve-space` element is the opposite of the `xsl:strip-space` element.

The rest of the XSLT template uses much of the same type of functionality used in the previous example, with a few noteworthy differences. The first difference is the `xsl:attribute` element in the email output element. The `xsl:attribute` element creates an attribute with the specified content. In this case, I am looking at the `Email-Promotion` source element and using an `xsl:choose` element to set the output `opt-in` attribute to `Yes` or `No` based on its value:

```

<email>
    <xsl:attribute name = "opt-in">
        <xsl:choose>
            <xsl:when test = "Customer/Email-Promotion = 0">No</xsl:when>
            <xsl:otherwise>Yes</xsl:otherwise>
        </xsl:choose>
    </xsl:attribute>
    <xsl:value-of select = "Customer/Email-Address" />
</email>

```

Another interesting addition is the `xsl:comment` element at the bottom of the stylesheet. This element adds an XML comment with the specified content to the output:

```

<xsl:comment>TERMS: 30 NET 10, 60 NET 5</xsl:comment>

```

Apart from the introduction of these additional features, the Acme Retailers XSLT stylesheet is very similar in form and functionality to the example created in the previous section. The XYZ Bike Repair XSLT stylesheet contains a few more differences. Recall that XYZ Bike Repair chose to send their orders in an attribute-centric XML format—one in which XML attributes are

used extensively in the structure. I have to account for this in their XSLT stylesheet, which is shown in Listing 8-10.

Listing 8-10. *XYZ Bike Repair XSLT Stylesheet*

```
SET @xyz_xslt = N'<xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

  <xsl:strip-space elements = "*" />
  <xsl:output method = "xml" encoding = "utf-16" />

  <xsl:template match = "/retail-order">
    <common-format-order>
      <date-info>
        <ordered>
          <xsl:value-of select = "dates/@order" />
        </ordered>
        <due>
          <xsl:value-of select = "dates/@due" />
        </due>
      </date-info>
      <account-info>
        <account-number>
          <xsl:value-of select = "account" />
        </account-number>
        <purchase-order-number>
          <xsl:value-of select = "purchase-order" />
        </purchase-order-number>
        <billing-info>
          <name>
            <first>
              <xsl:value-of select = "contact/@firstname" />
            </first>
            <middle>
              <xsl:value-of select = "contact/@middlename" />
            </middle>
            <last>
              <xsl:value-of select = "contact/@lastname" />
            </last>
          </name>
          <email>
            <xsl:attribute name = "opt-in">
              <xsl:choose>
                <xsl:when test = "contact/@emailopt = 0">No</xsl:when>
                <xsl:otherwise>Yes</xsl:otherwise>
              </xsl:choose>
            </xsl:attribute>
            <xsl:value-of select = "contact/@emailaddress" />
          </email>
        </billing-info>
      </account-info>
    </common-format-order>
  </xsl:template>
</xsl:stylesheet>
```

```

</email>
<phone>
  <xsl:value-of select = "contact/@phone" />
</phone>
<address>
  <street-address>
    <xsl:value-of select = "bill-to-address/@line1" />
  </street-address>
  <city>
    <xsl:value-of select = "bill-to-address/@city" />
  </city>
  <state>
    <xsl:value-of select = "bill-to-address/@state" />
  </state>
  <zip>
    <xsl:value-of select = "bill-to-address/@postal" />
  </zip>
</address>
</billing-info>
<shipping-info>
  <xsl:choose>
    <xsl:when test = "ship-to-address/@same-as-billing = &apos;YES&apos;">
      <address>
        <street-address>
          <xsl:value-of select = "bill-to-address/@line1" />
        </street-address>
        <city>
          <xsl:value-of select = "bill-to-address/@city" />
        </city>
        <state>
          <xsl:value-of select = "bill-to-address/@state" />
        </state>
        <zip>
          <xsl:value-of select = "bill-to-address/@postal" />
        </zip>
      </address>
    </xsl:when>
    <xsl:otherwise>
      <address>
        <street-address>
          <xsl:value-of select = "ship-to-address/@line1" />
        </street-address>
        <city>
          <xsl:value-of select = "ship-to-address/@city" />
        </city>
        <state>
          <xsl:value-of select = "ship-to-address/@state" />
        </state>
      </address>
    </xsl:otherwise>
  </xsl:choose>
</shipping-info>

```

```

        </state>
        <zip>
            <xsl:value-of select = "ship-to-address/@postal" />
        </zip>
    </address>
</xsl:otherwise>
</xsl:choose>
</shipping-info>
</account-info>
<order-info>
    <xsl:for-each select = "order-items/line">
        <detail-line>
            <item>
                <xsl:value-of select = "item" />
            </item>
            <quantity>
                <xsl:value-of select = "quantity" />
            </quantity>
            <price>
                <xsl:value-of select = "price-per-unit" />
            </price>
        </detail-line>
    </xsl:for-each>
    <summary>
        <subtotal>
            <xsl:value-of select = "subtotal" />
        </subtotal>
        <tax>
            <xsl:value-of select = "taxamount" />
        </tax>
        <shipping>
            <xsl:value-of select = "shipping" />
        </shipping>
        <total>
            <xsl:value-of select = "total" />
        </total>
    </summary>
    <xsl:comment>TERMS: 30 NET 5</xsl:comment>
</order-info>
</common-format-order>
</xsl:template>
</xsl:stylesheet>';

```

Like the Acme Retailers stylesheet, I've used the `xsl:strip-space` and `xsl:output` elements in the XYZ Bike Repair stylesheet. Apart from the structural differences, the main difference in functionality is in the `xsl:value-of` elements in this stylesheet. When addressing attributes in XPath, you prefix them with the at sign (@), as shown here:


```

<ordered>
  <xsl:value-of select = "dates/@order" />
</ordered>

```

The XYZ Bike Repair stylesheet makes extensive use of these attribute-centric XPath expressions to achieve the result. In the end, you simply pass the XML order from the customer and the matching stylesheet to the `fn_XsltTransform` function to get the common format XML orders. Listing 8-11 shows how to use `fn_XsltTransform` to convert the orders to a common format.

Listing 8-11. *fn_XsltTransform Transformations*

```

SELECT @common_format_order = dbo.fn_XsltTransform (@acme_retailers_order,
    @acme_xslt);

SELECT @common_format_order;

SELECT @common_format_order = dbo.fn_XsltTransform (@xyz_bike_repair_order,
    @xyz_xslt);

SELECT @common_format_order;

```

Once you have converted your customer orders to a standard common format, you can more easily manipulate them, process them, generate reports and query them, and shred them into relational format for storage in a SQL database. In the next example, I will return to the front-end/application layer to demonstrate advanced XSLT techniques.

Advanced XSL Transformations

So far the transformations have involved a single template. This is fine for simple transformations, but sometimes you'll have more complex requirements, like recursion. In this section, I will look at a multitemplate XSLT stylesheet, which supports recursion to produce HTML output from hierarchical XML. For this example, I'm going to reach back to Chapter 2 for the hierarchical Bill of Materials (BOM). I'll modify the query slightly to return only finished goods, as shown in Listing 8-12.

Listing 8-12. *Finished Goods Hierarchical BOM*

```

DECLARE @bom xml,
    @xslt xml;

SET @bom = (SELECT a.ComponentID AS "@id",
    p.ProductNumber AS "@number",
    p.Name AS "name",
    p.Color AS "color",
    p.ListPrice AS "list-price",
    a.PerAssemblyQty AS "quantity",
    p.Size AS "size",
    p.SizeUnitMeasureCode AS "unit-of-measure",

```

```

(
    SELECT b.ComponentID AS "@id",
           p.ProductNumber AS "@number",
           p.Name AS "name",
           p.Color AS "color",
           p.ListPrice AS "list-price",
           b.PerAssemblyQty AS "quantity",
           p.Size AS "size",
           p.SizeUnitMeasureCode AS "unit-of-measure",
    (
        SELECT c.ComponentID AS "@id",
               p.ProductNumber AS "@number",
               p.Name AS "name",
               p.Color AS "color",
               p.ListPrice AS "list-price",
               c.PerAssemblyQty AS "quantity",
               p.Size AS "size",
               p.SizeUnitMeasureCode AS "unit-of-measure",
        (
            SELECT d.ComponentID AS "@id",
                   p.ProductNumber AS "@number",
                   p.Name AS "name",
                   p.Color AS "color",
                   p.ListPrice AS "list-price",
                   d.PerAssemblyQty AS "quantity",
                   p.Size AS "size",
                   p.SizeUnitMeasureCode AS "unit-of-measure",
            (
                SELECT e.ComponentID AS "@id",
                       p.ProductNumber AS "@number",
                       p.Name AS "name",
                       p.Color AS "color",
                       p.ListPrice AS "list-price",
                       e.PerAssemblyQty AS "quantity",
                       p.Size AS "size",
                       p.SizeUnitMeasureCode AS "unit-of-measure"
            FROM Production.BillofMaterials e
            INNER JOIN Production.Product p
                ON e.ComponentID = p.ProductID
            WHERE e.ProductAssemblyID = d.ComponentID
                  AND e.EndDate IS NULL
            FOR XML PATH (N'item'), TYPE
        )
    )
    FROM Production.BillofMaterials d
    INNER JOIN Production.Product p
        ON d.ComponentID = p.ProductID
    WHERE d.ProductAssemblyID = c.ComponentID

```

```

        AND d.EndDate IS NULL
        FOR XML PATH (N'item'), TYPE
    )
    FROM Production.BillofMaterials c
    INNER JOIN Production.Product p
        ON c.ComponentID = p.ProductID
    WHERE c.ProductAssemblyID = b.ComponentID
        AND c.EndDate IS NULL
    FOR XML PATH (N'item'), TYPE
)
FROM Production.BillofMaterials b
INNER JOIN Production.Product p
    ON b.ComponentID = p.ProductID
WHERE b.ProductAssemblyID = a.ComponentID
    AND b.EndDate IS NULL
FOR XML PATH(N'item'), TYPE
)
FROM Production.BillofMaterials a
INNER JOIN Production.Product p
    ON a.ComponentID = p.ProductID
WHERE a.EndDate IS NULL
    AND p.FinishedGoodsFlag = 1
FOR XML PATH(N'item'), ROOT(N'items'), TYPE);

```

The Multitemplate Stylesheet

The stylesheet I will create contains two templates. The stylesheet has a main template for all first-level assemblies and a second named template that calls itself recursively for second-, third-, and fourth-level assemblies. Listing 8-13 defines the recursive multitemplate stylesheet.

Listing 8-13. Recursive Multitemplate XSLT Stylesheet

```

SET @xslt = N'<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml">

  <xsl:template match="/items">
    <html>

      <head>
        <style type="text/css">
          table {
            border-width: 1px;
            border-spacing: ;
            border-style: solid;
            border-color: black;
            border-collapse: collapse;
            background-color: white;

```

```

        width: 100%;
    }

    table th {
        border-width: 1px;
        padding: 3px;
        border-style: dotted;
        border-color: gray;
        background-color: #00ffff;
        font-family: arial;
        font-size: 12px;
    }

    table td {
        border-width: 1px;
        padding: 3px;
        border-style: dotted;
        border-color: gray;
        background-color: white;
        font-family: arial;
        font-size: 12px;
    }
</style>
</head>

<body>
    <h2>AdventureWorks Finished Goods List</h2>
    <table>
        <tr>
            <th>Level</th>
            <th>ID</th>
            <th>Number</th>
            <th>Name</th>
            <th>Color</th>
            <th>List Price</th>
            <th>Quantity</th>
            <th>Size</th>
            <th>UOM</th>
        </tr>
        <xsl:for-each select="item">
            <tr bgcolor="#00ffff">
                <td>1</td>
                <td>
                    <xsl:value-of select="@id"/>
                </td>
                <td>
                    <xsl:value-of select="@number"/>
                </td>
            </tr>
        </xsl:for-each>
    </table>

```

```

        <td>
            <xsl:value-of select="name"/>
        </td>
        <td>
            <xsl:value-of select="color"/>
        </td>
        <td>
            <xsl:value-of select="list-price"/>
        </td>
        <td>
            <xsl:value-of select="quantity"/>
        </td>
        <td>
            <xsl:value-of select="size"/>
        </td>
        <td>
            <xsl:value-of select="unit-of-measure"/>
        </td>
    </tr>
    <xsl:call-template name = "details">
        <xsl:with-param name="level">1</xsl:with-param>
    </xsl:call-template>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>

<xsl:template name = "details">
    <xsl:param name="level" />
    <xsl:if test="item">
        <tr>
            <td colspan="9">
                <table>
                    <tr>
                        <th>Level</th>
                        <th>ID</th>
                        <th>Number</th>
                        <th>Name</th>
                        <th>Color</th>
                        <th>List Price</th>
                        <th>Quantity</th>
                        <th>Size</th>
                        <th>UOM</th>
                    </tr>
                    <xsl:for-each select="item">
                        <tr>

```

```

        <td>
            <xsl:value-of select="$level + 1"/>
        </td>
        <td>
            <xsl:value-of select="@id"/>
        </td>
        <td>
            <xsl:value-of select="@number"/>
        </td>
        <td>
            <xsl:value-of select="name"/>
        </td>
        <td>
            <xsl:value-of select="color"/>
        </td>
        <td>
            <xsl:value-of select="list-price"/>
        </td>
        <td>
            <xsl:value-of select="quantity"/>
        </td>
        <td>
            <xsl:value-of select="size"/>
        </td>
        <td>
            <xsl:value-of select="unit-of-measure"/>
        </td>
    </tr>
    <xsl:if test="item">
        <xsl:call-template name="details">
            <xsl:with-param name="level" select="$level + 1"/>
        </xsl:call-template>
    </xsl:if>
</xsl:for-each>
</table>
</td>
</tr>
</xsl:if>
</xsl:template>
</xsl:stylesheet>';

```

As with the previous XML-to-HTML transformation example in this chapter, I start with the CSS definition. I then move on to define the HTML body and the table column headers:

```

<body>
    <h2>AdventureWorks Finished Goods List</h2>
    <table>
        <tr>

```

```

        <th>Level</th>
        <th>ID</th>
        <th>Number</th>
        <th>Name</th>
        <th>Color</th>
        <th>List Price</th>
        <th>Quantity</th>
        <th>Size</th>
        <th>UOM</th>
    </tr>
    ...
</table>
...
</body>

```

The `xsl:for-each` element loops through all the item elements in the hierarchical BOM listing and outputs the items to the HTML table:

```

<xsl:for-each select="item">
    <tr bgcolor="#00ffff">
        <td>1</td>
        <td>
            <xsl:value-of select="@id"/>
        </td>
        <td>
            <xsl:value-of select="@number"/>
        </td>
        <td>
            <xsl:value-of select="name"/>
        </td>
        <td>
            <xsl:value-of select="color"/>
        </td>
        <td>
            <xsl:value-of select="list-price"/>
        </td>
        <td>
            <xsl:value-of select="quantity"/>
        </td>
        <td>
            <xsl:value-of select="size"/>
        </td>
        <td>
            <xsl:value-of select="unit-of-measure"/>
        </td>
    </tr>
    ...
</xsl:for-each>

```

After each first-level item row is output, the main template calls the recursive named template to output the lower-level items. The XSLT `xsl:call-template` element calls the template named `details`. When the template is called, a parameter named `level` (set to an initial value of 1) is passed via the `xsl:with-param` element:

```
<xsl:call-template name = "details">
  <xsl:with-param name="level">1</xsl:with-param>
</xsl:call-template>
```

The `details` template starts by declaring a parameter with the `xsl:param` element. When you pass a parameter to a template, the `xsl:with-param` elements in the calling template require matching `xsl:param` elements inside the called template:

```
<xsl:template name = "details">
  <xsl:param name = "level" />
  ...
</xsl:template>
```

Recursion in the Stylesheet

The recursive `details` template uses the `xsl:if` element with an XPath expression of `item`. This expression is a simple XPath expression that evaluates to true (effective Boolean value) if there are any `item` elements under the current context node. In other words, the `xsl:if` expression evaluates to true if you are not currently at the bottom level of the source XML hierarchy. If the `xsl:if` test expression evaluates to true, the contents of the `xsl:if` element are evaluated and returned:

```
<xsl:if test="item">
  ...
</xsl:if>
```

If the `xsl:if` test expression evaluates to true, a nested HTML table with the same format as the outer table is created in the output. The level of the item in the hierarchy is also identified by the value of the `level` parameter, which is output after adding 1. Notice that the value of the `level` parameter is retrieved using the format `$level` in the `xsl:value-of` element:

```
<tr>
  <td colspan="9">
    <table>
      <tr>
        <th>Level</th>
        <th>ID</th>
        <th>Number</th>
        <th>Name</th>
        <th>Color</th>
        <th>List Price</th>
        <th>Quantity</th>
        <th>Size</th>
        <th>UOM</th>
      </tr>
```



```

<xsl:for-each select="item">
  <tr>
    <td>
      <xsl:value-of select="$level + 1"/>
    </td>
    <td>
      <xsl:value-of select="@id"/>
    </td>
    <td>
      <xsl:value-of select="@number"/>
    </td>
    <td>
      <xsl:value-of select="name"/>
    </td>
    <td>
      <xsl:value-of select="color"/>
    </td>
    <td>
      <xsl:value-of select="list-price"/>
    </td>
    <td>
      <xsl:value-of select="quantity"/>
    </td>
    <td>
      <xsl:value-of select="size"/>
    </td>
    <td>
      <xsl:value-of select="unit-of-measure"/>
    </td>
  </tr>
  ...
</table>
</td>
</tr>

```

After each row of output in the lower level is generated, the template calls itself recursively, adding 1 to the level parameter it calls itself with. When you reach the bottom level of the hierarchy, the details template will stop calling itself recursively:

```

<xsl:if test="item">
  <xsl:call-template name="details">
    <xsl:with-param name="level" select="$level + 1"/>
  </xsl:call-template>
</xsl:if>

```

To perform the transformation, you once again use the `p_XsltTransformToFile` procedure, as shown in Listing 8-14.

Listing 8-14. *Creating the HTML Finished Goods List*

```
EXEC dbo.p_XsltTransformToFile @bom, @xslt, 'C:\finished_goods_list.html';
```

The end result of the transformation is the properly nested finished goods BOM in HTML format, as shown in Figure 8-6.

AdventureWorks Finished Goods List								
Level	ID	Number	Name	Color	List Price	Quantity	Size	UOM
1	749	BK-R93R-62	Road-150 Red, 62	Red	3578.2700	1.00	62	CM
2	519	SA-R522	HL Road Seat Assembly		196.9200	1.00		
3	497	PB-6109	Pinch Bolt		0.0000	4.00		
3	528	SL-0931	Seat Lug		0.0000	1.00		
3	530	SP-2981	Seat Post		0.0000	1.00		
3	913	SE-R995	HL Road Seat/Saddle		52.6400	1.00		
2	717	FR-R92R-62	HL Road Frame - Red, 62	Red	1431.5000	1.00	62	CM
3	324	CS-2812	Chain Stays		0.0000	2.00		
4	486	MS-2341	Metal Sheet 5		0.0000	1.00		
3	325	DC-8732	Decal 1		0.0000	2.00		
3	326	DC-9824	Decal 2		0.0000	1.00		
3	327	DT-2377	Down Tube		0.0000	1.00		
4	483	MS-1256	Metal Sheet 3		0.0000	1.00		

Figure 8-6. *Result of nested BOM XML-to-HTML transformation*

XSLT's ability to transform XML to HTML is extremely useful, and support for named templates and hierarchical and recursive transformations provides a very powerful tool to generate an intricate and flexible user interface experience from raw XML data in the back end.

Summary

XSLT is a powerful and flexible language designed to make XML document manipulation and transformation very simple. While SQL Server does not provide out-of-the-box support to access XSLT directly from T-SQL, .NET does provide exactly that kind of support. Since SQL Server 2008 allows you to access .NET functionality from T-SQL, it is not a big stretch to access the .NET built-in XSLT support through SQLCLR functions and procedures.

In this chapter, I talked about using SQLCLR integration to make XSLT available to your T-SQL server-side code. I spent a good deal of the chapter discussing basic XSLT functionality and how to use it for very common user interface (UI) tasks, like generating HTML reports. I also talked about how XSLT can be used to standardize input from various sources, such as XML orders from different customers. Finally, I looked at some of the more advanced XSLT

functionality, like named templates and recursion to navigate hierarchical XML and generate intricate output documents.

In the next chapter, I will discuss SQL Server's built-in support for XML-based HTTP Simple Object Access Protocol (SOAP) endpoints and how this functionality can be used to further extend the power of your client and middle-tier applications.



HTTP SOAP Endpoints

I've covered a wide variety of SQL Server–based XML tools so far, including the `xml` data type, XML Schema, XQuery, and Extensible Stylesheet Language Transformations (XSLT). In addition to all this XML functionality, SQL Server also provides support for XML-based Simple Object Access Protocol (SOAP) endpoints over Hypertext Transfer Protocol (HTTP). SOAP provides the ability to serialize objects using XML, and HTTP provides the communication protocol to send and receive SOAP-based objects. In a nutshell, this means that you can use a SQL Server instance directly as a web service server. SQL Server provides built-in support for exposing stored procedures and user-defined functions as web service methods using SOAP. Using HTTP SOAP endpoints, you can access these exposed stored procedures and functions remotely with minimal effort.

SQL Server 2005 introduced HTTP SOAP endpoints, which proved to be a significant advance over the SQL Server 2000 model. SQL Server 2000 relied heavily on legacy Internet Information Services (IIS) and Component Object Model (COM)-based objects for providing web service support to SQL Server. SQL Server 2008 continues support for the simpler and more secure SQL Server 2005 SOAP endpoint model. In this chapter, I will discuss how to set up and configure HTTP SOAP endpoints and how to access them from your client applications.

IT SEEMED LIKE A GOOD IDEA AT THE TIME. . .

Back in SQL Server 2005, Microsoft introduced HTTP SOAP endpoints to expose server-side stored procedures and user-defined functions as web methods. With SQL Server 2008, HTTP SOAP endpoints have been deprecated. There's no official word yet on why, but it's a good guess that database administrator feedback concerning potential security issues played a big part in the decision. As of the time of this writing, there's no official word on the deprecation other than the following message you'll receive when you create an HTTP SOAP endpoint:

Creating and altering SOAP endpoints will be removed in a future version of SQL Server. Avoid using this feature in new development work, and plan to modify applications that currently use it.

I decided to retain this chapter after finding out about the deprecation of this feature because some will need to provide ongoing support and migration for SQL Server 2005 applications that currently use HTTP SOAP endpoints. As the error message suggests, however, you should plan to use alternative methods of exposing your data and SQL Server functionality and plan to move away from HTTP SOAP endpoints as soon as possible.

Creating Endpoints

According to the W3C, an *endpoint* is an “association between a binding and a network address . . . that may be used to communicate with an instance of a service.” Basically, an endpoint is a network location designated for web service communications with a specific communication protocol and data format. SQL Server allows you to define SQL Server–based endpoints that use HTTP as the communication protocol and SOAP as the data format.

Note SQL Server also supports Transfer Control Protocol (TCP) endpoints, but they are only available for configuring administrative tools like Service Broker and database mirroring. Web service applications must use the HTTP protocol.

SQL Server makes endpoint creation and management very easy with the Transact-SQL (T-SQL) `CREATE ENDPOINT`, `ALTER ENDPOINT`, and `DROP ENDPOINT` statements. To create an HTTP SOAP endpoint on SQL Server, you first need to create a stored procedure or user-defined function on the server. Later you will build a client-side web service consumer application in C#, which I call the AdventureWorks Product Browser. This simple application will allow you to easily browse the AdventureWorks catalog by product category, subcategory, and model. I'll take advantage of HTTP SOAP endpoints to retrieve the necessary information for the products from the .NET application.

SECURE YOUR SQL SERVER ENDPOINTS

There is some concern from database administrators about exposing SQL Server directly to web traffic via the HTTP SOAP endpoints. This concern is not unreasonable, since you could potentially open up your server to unwanted traffic and attacks from the Internet. However, HTTP SOAP endpoints provide security that is very tightly integrated with SQL Server's built-in security model. If your SQL Server is secure, your exposure to attacks via endpoints is minimized. If your server is not properly secured, endpoints only exacerbate the problem. Of course, an insecure SQL Server is probably more likely to be attacked using tried-and-true attacks, like SQL injection or password brute-force/dictionary attacks over the standard SQL Server TCP and user datagram protocol ports than via SOAP endpoints.

On the other hand, the vast majority of SQL Servers that are set up to expose web service methods via HTTP SOAP endpoints tend to be set up for internal use only, behind firewalls and with the additional network security provided by network administrators on a local area network or wide area network. If you set up any SQL Server for exposure over the Internet (endpoints enabled or not), make sure you do a thorough security analysis of your server and network to minimize the potential that your server, network, and data can become compromised. Though they are outside the scope of this book, the same type of precautions should be taken when exposing any computer to the Internet, including web servers and other networked computers.

The first step is to create the procedures and functions that you want to expose as web service methods directly from SQL Server. In this case, I'll create three web service methods: one to retrieve a list of products to populate the user interface, another to get product photographs, and a third to retrieve the HTML transformed version of the catalog description for a product.

Listing 9-1 shows the `p_GetProductHierarchy` stored procedure that I will use to retrieve product listings with appropriate product category, subcategory, and model information.

Listing 9-1. *p_GetProductHierarchy Procedure to Retrieve Product Listings*

```
CREATE PROCEDURE dbo.p_GetProductHierarchy
AS
BEGIN
    SELECT p.ProductID,
           p.Name AS ProductName,
           pc.ProductCategoryID,
           pc.Name AS ProductCategoryName,
           ps.ProductSubcategoryID,
           ps.Name AS ProductSubcategoryName,
           pm.ProductModelID,
           pm.Name AS ProductModelName
    FROM Production.Product p
    INNER JOIN Production.ProductModel pm
        ON pm.ProductModelID = p.ProductModelID
    INNER JOIN Production.ProductSubcategory ps
        ON p.ProductSubcategoryID = ps.ProductSubcategoryID
    INNER JOIN Production.ProductCategory pc
        ON pc.ProductCategoryID = ps.ProductCategoryID
    ORDER BY pc.Name, ps.Name, pm.Name, p.Name;
END
GO
```

Listing 9-2 is the user-defined function `fn_GetProductPhoto`, which retrieves a product photograph image for a given product. It can return either the full-sized image or the thumbnail photo, depending on value of the `@Thumbnail` bit flag.

Listing 9-2. *fn_GetProductPhoto Function to Retrieve a Product Photograph*

```
CREATE FUNCTION dbo.fn_GetProductPhoto (@ProductID INT, @Thumbnail BIT)
RETURNS VARBINARY(MAX)
AS
BEGIN
    RETURN (
        SELECT CASE @Thumbnail
            WHEN 1 THEN pp.ThumbNailPhoto
            ELSE pp.LargePhoto
        END
        FROM Production.Product p
        INNER JOIN Production.ProductProductPhoto ppp
            ON p.ProductID = ppp.ProductID
        INNER JOIN Production.ProductPhoto pp
            ON ppp.ProductPhotoID = pp.ProductPhotoID
        WHERE p.ProductID = @ProductID
    )
END
```

```
);
END
GO
```

The final function I will expose is called `fn_GetHtmlCatalogDescription`. This function transforms the XML catalog description for a product via XSLT and the `fn_XsltTransform` I previously created in Chapter 8. The XSLT stylesheet used is based on the Microsoft-supplied sample stylesheet `ProductDescription.xsl`. Listing 9-3 is the source listing for the `fn_GetHtmlCatalogDescription` function.

Listing 9-3. *fn_GetHtmlCatalogDescription Function to Retrieve HTML Description*

```
CREATE FUNCTION dbo.fn_GetHtmlCatalogDescription (@ProductID INT)
RETURNS xml
AS
BEGIN
    DECLARE @xslt xml;
    SET @xslt = N'<?xml version=''1.0''?>
<xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
    version = "1.0"
    xmlns:html="http://www.w3.org/1999/xhtml"
    xmlns:x = "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
        ProductModelDescription"
    xmlns:wm = "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
        ProductModelWarrAndMain"
    xmlns = "http://www.w3.org/1999/xhtml"
    xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➤
        ProductModelDescription"
    xmlns:wf="http://www.adventure-works.com/schemas/OtherFeatures"
    exclude-result-prefixes="html wm x p1 wf">

<xsl:preserve-space elements = "*" />
<xsl:output method = "html" />

<xsl:template match = "/">
    <html>
        <head>
            <style>
                body { font-family: Arial, Helvetica, sans-serif; }
                h1 { font-size: bigger; color: #336699; text-align: center; }
                table.specs { width: 85%; border-width: 1px;
                    cell-spacing: 0px; cell-padding: 0px;
                    border-color: black; border-style: solid; }
                tr.specs { background-color: #336699; }
            </style>
        </head>
```



```

    <body>
      <xsl:apply-templates select="//x:ProductDescription"/>
    </body>
  </html>
</xsl:template>

```

```

<xsl:template match="x:ProductDescription">
  <h1>
    <xsl:value-of select="@ProductModelName"/> -
    <xsl:value-of select="@ProductModelID"/>
  </h1>
  <xsl:if test="x:Features/*">
    <table>
      <tr>
        <td>
          <ul>
            <xsl:for-each select="x:Features/*">
              <li>
                <b>
                  <xsl:value-of select="local-name()"/>
                </b>:
                <xsl:for-each select="*|text()">
                  <xsl:choose>
                    <xsl:when test = "text()">
                      <xsl:copy-of select="text()"/>
                    </xsl:when>
                    <xsl:otherwise>
                      <xsl:copy-of select="."/>
                    </xsl:otherwise>
                  </xsl:choose>
                  <xsl:text>&#32;</xsl:text>
                </xsl:for-each>
              </li>
            </xsl:for-each>
          </ul>
        </td>
      </tr>
    </table>
  </xsl:if>
  <xsl:apply-templates select="x:Summary"/>
  <xsl:if test="x:Specifications/*">
    <table class="specs">
      <tr class="specs">
        <th colspan="2" style="color:white;">
          Specifications
        </th>
      </tr>
    </table>
  </xsl:if>

```

```

        <xsl:for-each select="x:Specifications/*">
        <tr valign="top">
            <th align="left">
                <xsl:value-of select="local-name()"/>
            </th>
            <td align="left">
                <xsl:value-of select="."/>
            </td>
        </tr>
        </xsl:for-each>
    </table>
</xsl:if>
</xsl:template>

<xsl:template match="x:Summary">
    <table>
        <tr>
            <td>
                <xsl:copy-of select="*" />
            </td>
        </tr>
    </table>
</xsl:template>

</xsl:stylesheet>';

DECLARE @xml_catalog_description xml;

SELECT @xml_catalog_description = pm.CatalogDescription
FROM Production.Product p
INNER JOIN Production.ProductModel pm
    ON p.ProductModelID = pm.ProductModelID
WHERE p.ProductID = @ProductID;

RETURN dbo.fn_XsltTransform(@xml_catalog_description, @xslt);
END
GO

```

Once the functions and procedures are created, it's time to create an HTTP SOAP endpoint to expose them. Listing 9-4 shows the CREATE ENDPOINT statement I will use to create the HTTP SOAP endpoint and expose the web service methods I've defined.

Listing 9-4. *Create an HTTP SOAP Endpoint*

```

CREATE ENDPOINT AdvWorksProductBrowser
STATE = STARTED
AS HTTP
(
    PATH = '/Browser',
    AUTHENTICATION = ( INTEGRATED ),
    PORTS = ( CLEAR ),
    SITE = '*',
    CLEAR_PORT = 888
)
FOR SOAP
(
    WEBMETHOD 'GetProductHierarchy'
    (
        NAME = 'AdventureWorks.dbo.p_GetProductHierarchy',
        SCHEMA = STANDARD,
        FORMAT = ROWSETS_ONLY
    ),
    WEBMETHOD 'GetProductPhoto'
    (
        NAME = 'AdventureWorks.dbo.fn_GetProductPhoto',
        SCHEMA = STANDARD,
        FORMAT = ALL_RESULTS
    ),
    WEBMETHOD 'GetHtmlCatalogDescription'
    (
        NAME = 'AdventureWorks.dbo.fn_GetHtmlCatalogDescription',
        SCHEMA = STANDARD,
        FORMAT = ALL_RESULTS
    ),
    BATCHES = DISABLED,
    WSDL = DEFAULT,
    LOGIN_TYPE = WINDOWS,
    DATABASE = 'AdventureWorks'
);

```

The `CREATE ENDPOINT` statement is the key to this process, so I'll break down the statement used. The first part creates the endpoint, gives it the name `AdvWorksProductBrowser`, and sets its initial state to `STARTED`.

```

CREATE ENDPOINT AdvWorksProductBrowser
STATE = STARTED

```

The `AS HTTP` clause defines the parameters for the HTTP protocol. The `PATH` option in the example assigns a relative path of `/Browser` for the service. The `AUTHENTICATION` mode I've chosen for this example is `INTEGRATED`, which uses Windows Integrated Authentication. Alternatively, you can specify basic, digest, NTLM, or Kerberos as the authentication method. The `PORTS`

option lets you specify whether clients can connect using clear HTTP or secure HTTP ports. In the example, I chose clear ports. The `SITE` option specifies the name of the host computer. You can force clients to use a specific server name or IP address when connecting. The asterisk (*) , which I used in the example, allows client access using any name or address that is not explicitly reserved. The `CLEAR_PORT` setting sets the port number for non-SSL client connections. This setting can be important if you're running IIS on the same server, since IIS has a habit of intercepting HTTP requests on the default port 80.

```
AS HTTP
(
    PATH = '/Browser',
    AUTHENTICATION = ( INTEGRATED ),
    PORTS = ( CLEAR ),
    SITE = '*',
    CLEAR_PORT = 888
)
```

The `FOR SOAP` clause specifies the payload type, or the data format used by the endpoint. The `FOR SOAP` clause contains `WEBSERVICE` subclauses which define the web service methods we're exposing. Each `WEBSERVICE` subclause specifies a name by which the method is known to outside applications and the name of the SQL Server function or procedure that fulfills the web service method request. In the example, I assign a web service name of `GetProductHierarchy` to the `dbo.p_GetProductHierarchy` procedure, for instance. You can also use these clauses to specify whether an inline XML schema is returned with the results, as well as the format in which results are returned.

Tip For stored procedures, set the `FORMAT` to `ROWSETS_ONLY`, and for user-defined functions, set the `FORMAT` to `ALL_RESULTS`. Note that I've also specified that an inline XML schema should be returned with the results. These settings are required if you want to return results as .NET `DataSets`.

The `FOR SOAP` clause also sets some additional endpoint-wide settings. In the example, I disabled batches, which allow ad hoc querying of the database over the HTTP endpoints. I've also set the `WSDL` settings to the default, set the default login type to `WINDOWS`, and set the default database to `AdventureWorks`.

HTTP ENDPOINTS AND AD HOC QUERYING

The `BATCHES` option is a powerful option, but it should be used with care. Any time you allow ad hoc querying of your database, you should carefully consider the security ramifications. I would advise against allowing ad hoc querying over HTTP SOAP endpoints unless you have a compelling reason. And even then you should perform a very thorough security review to ensure that no unauthorized access is allowed to your database and to make sure that users cannot execute destructive ad hoc T-SQL code on your server.

```
FOR SOAP
(
    WEBMETHOD 'GetProductHierarchy'
    (
        NAME = 'AdventureWorks.dbo.p_GetProductHierarchy',
        SCHEMA = STANDARD,
        FORMAT = ROWSETS_ONLY
    ),
    WEBMETHOD 'GetProductPhoto'
    (
        NAME = 'AdventureWorks.dbo.fn_GetProductPhoto',
        SCHEMA = STANDARD,
        FORMAT = ALL_RESULTS
    ),
    WEBMETHOD 'GetHtmlCatalogDescription'
    (
        NAME = 'AdventureWorks.dbo.fn_GetHtmlCatalogDescription',
        SCHEMA = STANDARD,
        FORMAT = ALL_RESULTS
    ),
    BATCHES = DISABLED,
    WSDL = DEFAULT,
    LOGIN_TYPE = WINDOWS,
    DATABASE = 'AdventureWorks'
);
```

SQL Server has the ability to generate Web Services Description Language (WSDL) documents, which describe the web service methods it can offer to clients. You can retrieve the standard WSDL document by pointing your browser at your SQL Server HTTP SOAP endpoint with a `?wsdl` parameter. In the example, my server is named `Sql2008`, the path is `/Browser`, and the port is 888. Entering the full path to my WSDL document in Internet Explorer, `http://Sql2008:888/Browser?wsdl`, returns the WSDL document shown in Figure 9-1. This is also a very simple check to ensure that your HTTP endpoint is properly configured and communicating correctly.



Figure 9-1. WSDL document for HTTP SOAP endpoint example

Tip Some older applications and platforms require a simpler WSDL format. For those, you can pull up a backward-compatible simple WSDL document using the `?wsdlsimple` parameter in place of `?wsdl`. For most applications, however, the full standard WSDL document is the correct choice.

Consuming Endpoints

HTTP SOAP endpoints, like any web services, are relatively easy to consume from .NET. You simply add a web reference to our client application in Visual Studio, write a bit of code to access the methods, and you're well on your way. After creating a new C# project, simply add a web reference by right-clicking References in the Solution Explorer and selecting Add Web Reference, as shown in Figure 9-2.

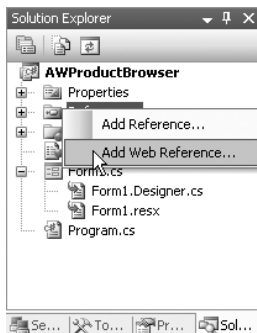


Figure 9-2. Adding a web reference in Visual Studio

The Add Web Reference window pops up and requires you to enter the URL for the endpoint WSDL document. As I mentioned previously, this example is located at <http://Sql2008:888/Browser?wsdl>. Figure 9-3 shows the Add Web Reference window.

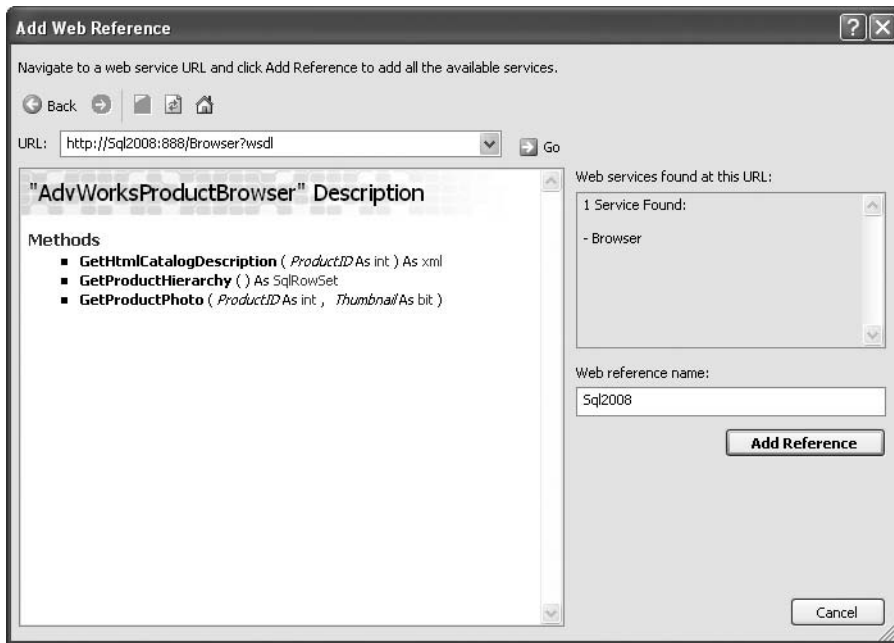


Figure 9-3. *Add Web Reference window*

Once the web reference has been added, Visual Studio automatically generates a proxy class wrapper of the same name for the web service. Once done, I'll create a Windows Form and drag over a tree view control, a picture box control, and a web browser control onto it. Next I add some private functions to perform the actual web service method calls. These functions are shown in Listing 9-5.

Listing 9-5. *Web Service Method Call Private Functions*

```
private void PopulateSearchTree()
{
    Sql2008.AdvWorksProductBrowser ws = new Sql2008.AdvWorksProductBrowser();
    ws.Credentials = CredentialCache.DefaultCredentials;
    DataSet ds = ws.GetProductHierarchy();

    string CategoryName = "";
    string SubcategoryName = "";
    string ProductName = "";
    TreeNode CategoryNode = new TreeNode();
    TreeNode SubcategoryNode = new TreeNode();
    foreach (DataRow dr in ds.Tables[0].Rows)
    {
```

```

        CategoryName = dr["ProductCategoryName"].ToString();
        SubcategoryName = dr["ProductSubcategoryName"].ToString();
        ProductName = dr["ProductName"].ToString();
        if (CategoryName != CategoryNode.Text)
        {
            CategoryNode = new TreeNode();
            CategoryNode.Text = CategoryName;
            treeView1.Nodes.Add(CategoryNode);
        }
        if (SubcategoryName != SubcategoryNode.Text)
        {
            SubcategoryNode = new TreeNode();
            SubcategoryNode.Text = SubcategoryName;
            CategoryNode.Nodes.Add(SubcategoryNode);
        }
        TreeNode ProductNode = new TreeNode();
        ProductNode.Text = ProductName;
        ProductNode.Tag = (int)dr["ProductID"];
        SubcategoryNode.Nodes.Add(ProductNode);
    }
}

private Image GetProductImage(int ProductID)
{
    Sql2008.AdvWorksProductBrowser ws = new Sql2008.AdvWorksProductBrowser();
    ws.Credentials = CredentialCache.DefaultCredentials;
    SqlBinary photo = ws.GetProductPhoto(new SqlInt32(ProductID), SqlBoolean.False);
    MemoryStream ms = new MemoryStream(photo.Value);
    return Image.FromStream(ms);
}

private string GetHtmlProductDescription(int ProductID)
{
    Sql2008.AdvWorksProductBrowser ws = new Sql2008.AdvWorksProductBrowser();
    ws.Credentials = CredentialCache.DefaultCredentials;
    AWProductBrowser.Sql2008.xml html =
        ws.GetHtmlCatalogDescription(new SqlInt32(ProductID));
    string html_string = "<html><body><b>No Description</b></body></html>";
    if (html != null)
        html_string = html.Any[0].OuterXml;
    return html_string;
}

```

As you can see, the `PopulateSearchTree` function is the most elaborate of the three. The web service method call itself is very simple, though. As shown in the following code snippet, you simply create an instance of the proxy class in .NET, set the security `Credentials` to the current user's credentials (since I'm using integrated security), and assign the results of the web service method call to a `DataSet`.


```
Sql2008.AdvWorksProductBrowser ws = new Sql2008.AdvWorksProductBrowser();
ws.Credentials = CredentialCache.DefaultCredentials;
DataSet ds = ws.GetProductHierarchy();
```

The remainder of the code in the `PopulateSearchTree` function simply loops through the `DataSet` result rows after the call to `p_GetProductHierarchy` and fills the tree view control with `TreeNode`s. The `GetProductImage` function calls its user-defined function in a similar fashion and assigns the results to a `SqlBinary` variable. A little manipulation is required to convert the `SqlBinary` result to a `MemoryStream`, which is then converted to an image for display in the Windows Form.

```
Sql2008.AdvWorksProductBrowser ws = new Sql2008.AdvWorksProductBrowser();
ws.Credentials = CredentialCache.DefaultCredentials;
SqlBinary photo = ws.GetProductPhoto(new SqlInt32(ProductID), SqlBoolean.False);
MemoryStream ms = new MemoryStream(photo.Value);
return Image.FromStream(ms);
```

The `GetHtmlProductDescription` performs similarly, except that the resultant XHTML is converted to a string that is returned to the calling routine.

```
Sql2008.AdvWorksProductBrowser ws = new Sql2008.AdvWorksProductBrowser();
ws.Credentials = CredentialCache.DefaultCredentials;
AWProductBrowser.Sql2008.xml html =
    ws.GetHtmlCatalogDescription(new SqlInt32(ProductID));
string html_string = "<html><body><b>No Description</b></body></html>";
if (html != null)
    html_string = html.Any[0].OuterXml;
return html_string;
```

Finally, I add a little functionality to the tree view control's `AfterSelect` event to update the picture box image and web browser HTML. I'll also add a call to the `PopulateSearchTree` function in the form's `Load` event.

```
private void treeView1_AfterSelect(object sender, TreeViewEventArgs e)
{
    if (e.Node != null && e.Node.Tag != null)
    {
        pictureBox1.Image = GetProductImage((int)e.Node.Tag);
        webBrowser1.DocumentText = GetHtmlProductDescription((int)e.Node.Tag);
    }
    else
    {
        pictureBox1.Image = null;
        webBrowser1.DocumentText = null;
    }
}

private void Form1_Load(object sender, EventArgs e)
{
    PopulateSearchTree();
}
```

The end result is a simple Windows client application that allows you to browse the category/subcategory/product hierarchy and view catalog information for any given product in that hierarchy. Figure 9-4 shows the AdventureWorks Product Browser client application in action.

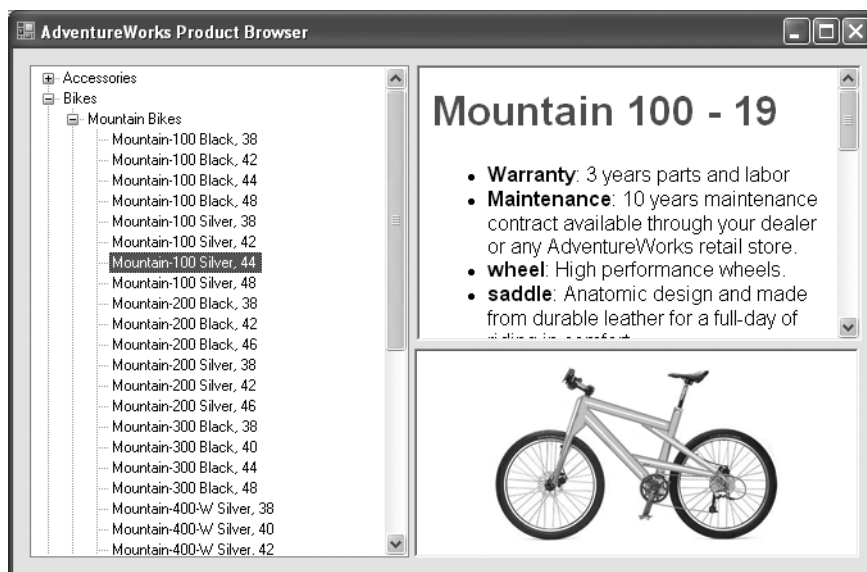


Figure 9-4. AdventureWorks Product Browser client application

Summary

As you saw in this chapter, HTTP SOAP endpoints provide a simple way to expose stored procedures and user-defined functions for remote access. Endpoints provide advantages over the normal methods of exposing web service functionality. These advantages include a security model that is tightly integrated with SQL Server security, easy setup and administration, and efficiency advantages over many web server-based options. Although this feature has been deprecated in SQL Server 2008 and will be removed from a future version of SQL Server, developers and administrators may be faced with the task of migrating existing code from SQL Server 2005 that uses HTTP SOAP endpoints. Because of this fact, it's important to understand their purpose and available features.

I also demonstrated how easy it is to access web service methods from a .NET client application. Although Visual Studio provides a simple interface to access web service methods, it's not difficult to access a web service from other platforms and programming languages. That's the attractiveness and real power behind web services and the Service-Oriented Architecture (SOA)—the promise that you can access functionality remotely from any platform, anywhere. Although examples are beyond the scope of this book, it's not at all difficult to access web services from ASP.NET on Windows, Perl on Linux, or even Java on UNIX.

In this chapter, I began looking at SQL Server XML from the client's perspective, and in the next chapter, I will continue that discussion with a more detailed look at XML from the .NET client side.



.NET XML Support

The Microsoft .NET Framework offers a wide variety of XML functionality through the `System.Xml` namespace and other related classes. Using the .NET Framework, you can do just about anything you could possibly want with XML, including many XML-specific tasks that are not available directly from Transact-SQL (T-SQL). The .NET Framework's XML functionality is available on the client side and, as you saw in Chapter 8, it is also readily accessible to SQL Server via SQL Common Language Runtime (SQLCLR) integration. In this chapter, I'll look at the `System.Xml` namespace and other XML-related tasks you can perform using the Framework's functionality.

XML Validation

The SQL Server `xml` data type provides XML Schema validation, but it provides only minimal Document Type Definition (DTD) support and no support for XML Data-Reduced (XDR) schema validation. You may have a need to validate your XML data against complex DTDs and legacy XDR schemas, in which case you can use the .NET `System.Xml.XmlReader` class.

DTD, XDR, AND XSD

DTDs provide the most basic standard level of validation available to XML. DTDs provide simple structure and string-based content validation; however, they do not provide data typing functionality. DTDs also use a legacy non-XML format inherited from Standard Generalized Markup Language (SGML). DTDs have the advantage of having been part of the XML recommendation since the beginning, so nearly all XML parsers support them to some degree.

XDR schemas represent Microsoft's first attempt at implementing an early working draft of the W3C XML Schema recommendation. XDR functionality is Microsoft-specific, and you won't find it implemented widely on other platforms. XDR functionality for constraining XML structure and typing content has been superseded by the official W3C XML Schema recommendation, and support for XDR is provided for backward-compatibility reasons. You normally won't use XDR schemas for new functionality, though you might run into them while supporting legacy applications.

XML Schema Definition (XSD) documents represent the physical implementation of schemas based on the XML Schema recommendation. XML Schema provides a thorough data typing system and the ability to define both simple and complex content. I discussed the SQL Server implementation of XSD in detail in Chapter 4.

The main reason for continued support of both DTDs and XDR schemas even after the introduction of the powerful and flexible W3C XML Schema recommendation is backward-compatibility. DTDs are nearly universal, regardless of platform, because they are part of the original W3C XML recommendation and have been around so long. XDR is still used in some internal .NET Framework functionality, like serializing `System.Data.DataSet` instances, so XDR support is still required internally in some instances.

The example for this section will focus on using .NET to validate an XML document against an external DTD, functionality that is not available directly through the T-SQL `xml` data type. This is probably one of the more common scenarios you will face in the real world, as many businesses have invested a good deal of resources into the creation of standardized external DTDs over the years.

I'll begin by introducing the Organization for the Advancement of Structured Information Standards (OASIS). This not-for-profit consortium drives development of open standards for the public, government, and business sectors. One of the OASIS member sections, LegalXML, is tasked with creating standards for the electronic exchange of legal data. Among the standards produced by this technical committee is the LegalXML eContracts version 1.0 specification. The eContracts specification is designed to promote the efficient creation, exchange, and publication of legal contracts via XML. Listing 10-1 is a simple LegalXML eContract for AdventureWorks to sell.

Listing 10-1. *LegalXML eContract Example*

```
<?xml version="1.0" encoding="UTF-16"?>

<!DOCTYPE contract
SYSTEM "c:\eContracts-Reference.dtd">

<contract xmlns="urn:oasis:names:tc:eContracts:1.0"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <metadata>
    <dc:creator>The Law Firm of Dewey, Cheatham, and Howe</dc:creator>
    <dc:date>2007-12-01</dc:date>
  </metadata>
  <title><text>Sample AdventureWorks Bicycle Sales Contract</text></title>
  <contract-front>
    <date-block>
      <date>Agreement dated: <em>2007-12-01</em></date>
    </date-block>
    <parties>
      <party id="Seller">
        <person-record><name>AdventureWorks Bicycles, Inc.</name></person-record>
      </party>
      <party id="Buyer">
```

```

    <person-record><name>XYZ Bike Sales and Repair</name></person-record>
  </party>
</parties>
</contract-front>
<body>
  <block number-type="upperalpha">
    <item number="A">
      <title><text>Identities of the Parties</text></title>
      <block>
        <text>
          Seller, <em>AdventureWorks Bicycles, Inc.,</em> whose primary
          business address is 15600 Redmond Way, in the city of Redmond, state
          of Washington, is in the business of manufacturing, selling, and
          servicing bicycles and bicycle-related goods and service. Buyer,
          <em>XYZ Bike Sales and Repair,</em> whose business address
          is 42 Crescent Avenue, in the city of Newark, state of New Jersey,
          is in the business of retail sales of bicycles and bicycle services.
        </text>
      </block>
    </item>
    <item number="B">
      <block>
        <text>
          Seller agrees to transfer and deliver to Buyer, on or before the
          3rd day of March, 2008, the below-described goods in the quantities
          specified:

          Thirty (30) Red Road-150, 44 inch bicycles, catalog item number
            BK-R93R-44
          Twenty-three (23) Red Road-450, 52 inch bicycles, catalog item number
            BK-R68R-52
          Nineteen (19) Black Road-650, 60 inch bicycles, catalog item number
            BK-R50B-60
          Thirty-six (36) Silver Mountain-200, 46 inch bicycles, catalog item
            number BK-M68S-46
        </text>
      </block>
    </item>
    <item number="C">
      <block>
        <text>
          Buyer agrees to accept the goods and pay for them according to the
          terms further set out in the addendum to this contract.

          Upon delivery of goods, Buyer agrees to pay to Seller the sum of two
          hundred and twenty thousand U.S. dollars (US$220,000) in full.
        </text>
      </block>
    </item>
  </block>
</body>

```

```

    </block>
  </item>
  <item number="D">
    <block>
      <text>
        Until received by Buyer, all risk of loss to the above-described
        goods is borne by Seller.

        Seller warrants that the goods are free from any and all security
        interests, liens, and encumbrances.
      </text>
    </block>
  </item>
</block>
</body>
</contract>

```

This XML example represents a very simple LegalXML eContract, with some portions omitted for brevity. A full description of the LegalXML eContracts standard is beyond the scope of this book, but the important thing is that LegalXML makes their eContracts DTD freely available to everyone. A portion of the eContracts DTD is shown in Listing 10-2.

Listing 10-2. *Partial LegalXML eContracts DTD*

```

<!--
#####
# CONTRACT
#####
-->

<!ELEMENT contract (metadata?, title, subtitle*, contract-front?,
  body, back?, attachments*)>

<!ATTLIST contract
  id          ID          #IMPLIED
  class       CDATA       #IMPLIED
  orient      (portrait | landscape) #IMPLIED
  xml:lang    CDATA       #IMPLIED

  xmlns       CDATA  #FIXED  "urn:oasis:names:tc:eContracts:1:0"
  xmlns:dc    CDATA  #FIXED  "http://purl.org/dc/elements/1.1/"
  xmlns:xi    CDATA  #FIXED  "http://www.w3.org/2001/XInclude"
>

<!--

```

```
#####
# CONTRACT-FRONT
#####
-->

<!ELEMENT contract-front (((date-block?, parties) | block+),
    background?)>

<!ATTLIST contract-front
    id          ID          #IMPLIED
    class       CDATA       #IMPLIED
    xml:lang    CDATA       #IMPLIED
>

<!--
#####
# DATE-BLOCK
#####
-->

<!ELEMENT date-block (#PCDATA | reference | em | statutory-em | strike | sub
    | sup | conditional | object | term | phrase | field
    | note | note-in-line | name | address | date | party
    | person-record)*>

<!ATTLIST date-block
    id          ID          #IMPLIED
    xml:lang    CDATA       #IMPLIED
>
. . .
```

The full DTD for LegalXML eContracts, and the LegalXML eContracts standard is available at www.legalxml.org. The DTD is included in the sample download code, courtesy of the copyright holders, OASIS LegalXML and Elkera Pty Limited.

When you include a DTD DOCTYPE declaration in an `xml` data type instance, SQL Server performs very limited checks on it. SQL Server validates only inline DTDs, and it only checks the syntax of the DTD for correctness and expands entity references. As a final step, SQL Server strips the inline DTD from the XML data. In the case of the LegalXML eContract example, I want to validate the entire document against the DTD, including entities, structures, and content constraints. I also want to load the DTD from the file system, functionality that is performed by XML with the `SYSTEM` keyword, which references the location of an external DTD.

For this example, I will create and deploy a SQLCLR user-defined function (UDF), which will validate an XML document with a DTD, declared to be either external or internal. The core validation functionality is provided by the `System.Xml.XmlReader` and `System.Xml.XmlReaderSettings` classes. The `XmlReader` will validate an XML document or fragment using the settings provided by the `XmlReaderSettings` instance. This user-defined function implements two methods, the private `MyHandler` method to handle `XmlReader` exceptions and the public `fn_ValidateDTD` method, which accepts an XML document as a `SqlString`. This is important—you cannot pass an `xml` data type instance to the user-defined function in this case because SQL Server will strip off the DTD DOCTYPE

declaration before passing it to the function. The full listing for this assembly is shown in Listing 10-3.

Listing 10-3. *fn_ValidateDTD SQLCLR UDF Listing*

```
using System;
using System.Data.SqlTypes;
using System.Xml;
using System.IO;
using System.Text;
using System.Xml.Schema;

namespace Apress.Samples
{
    public partial class ValidationDTD
    {
        [Microsoft.SqlServer.Server.SqlFunction]
        public static SqlBoolean fn_ValidateDTD(SqlString xml)
        {
            SqlBoolean result = new SqlBoolean(true);
            try
            {
                UnicodeEncoding encoder = new UnicodeEncoding();
                byte[] buffer = encoder.GetBytes(xml.Value);
                MemoryStream stream = new MemoryStream(buffer);

                XmlReaderSettings settings = new XmlReaderSettings();
                settings.CloseInput = true;
                settings.ValidationFlags = settings.ValidationFlags |
                    XmlSchemaValidationFlags.ReportValidationWarnings;
                settings.ConformanceLevel = ConformanceLevel.Document;
                settings.ValidationType = ValidationType.DTD;
                settings.ProhibitDtd = false;

                XmlReader reader = XmlReader.Create(stream, settings);

                settings.ValidationEventHandler +=
                    new ValidationEventHandler(MyHandler);

                while (reader.Read()) { ; }
            }
            catch (Exception ex)
            {
                result = SqlBoolean.False;
                throw (new Exception(ex.Message));
            }
            return result;
        }
    }
}
```



```

private static void MyHandler(object sender, ValidationEventArgs e)
{
    throw (e.Exception);
}
}
};

```

The `fn_ValidateDTD` function starts by converting the XML document passed into it to a `System.IO.MemoryStream`, as shown by the following:

```

UnicodeEncoding encoder = new UnicodeEncoding();
byte[] buffer = encoder.GetBytes(xml.Value);
MemoryStream stream = new MemoryStream(buffer);

```

Then the function sets `XmlReaderSettings` options for validation. I set the `CloseInput` setting to `true`, which closes the `XmlReader` automatically upon completion. In addition, the `ValidationFlags` option is set to report validation warnings as well as errors, and the `ConformanceLevel` setting is set to document-level conformance. I also set the `ValidationType` setting to `DTD` to prevent XML schema or XDR validations, and I set `ProhibitDtd` to `false` to allow DTDs to be loaded from external sources. For security reasons, loading of external DTDs is disabled by default.

```

XmlReaderSettings settings = new XmlReaderSettings();
settings.CloseInput = true;
settings.ValidationFlags = settings.ValidationFlags |
    XmlSchemaValidationFlags.ReportValidationWarnings;
settings.ConformanceLevel = ConformanceLevel.Document;
settings.ValidationType = ValidationType.DTD;
settings.ProhibitDtd = false;

```

Tip The functionality of the `XmlValidatingReader` from .NET 1.1 has been replaced in .NET 2.0 by the `XmlReader` and the `XmlReaderSettings` classes. If you see references to the `XmlValidatingReader` in the literature, keep in mind that it is obsolete.

Next I create the `XmlReader` using the `XmlReaderSettings` I previously set and add a reference to the event handler, which will handle validation events like warnings and validation errors.

```

XmlReader reader = XmlReader.Create(stream, settings);

settings.ValidationEventHandler +=
    new ValidationEventHandler(MyHandler);

```

The final step is to execute the `XmlReader`'s `Read()` method to validate the document. The entire function is enclosed in a `try . . . catch` block to catch any possible exceptions that occur during the validation. These exceptions are passed back up to SQL Server.

```
while (reader.Read()) { ; }
```

The source code for this example is included in the sample code downloads for this chapter, so you can compile, deploy, and execute the code yourself; or you can use the `CREATE ASSEMBLY` statement provided in the sample scripts to deploy the assembly and avoid compiling it yourself. Because the code can access external DTDs, the database you deploy to must be marked `TRUSTWORTHY` and the assembly must be marked with `EXTERNAL_ACCESS` permissions. The instructions for compilation and deployment are similar to those provided in Chapter 8. This example also requires that you copy the DTD file `eContracts-Reference.dtd` to the root of the local C: drive. Listing 10-4 is a partial listing demonstrating validation of a LegalXML eContract against the DTD.

Tip Make sure that your SQL Server service account has access to the root directory of the C: drive to run this sample. If your SQL Server service account doesn't have access, you can copy the eContracts DTD to a subdirectory and modify the code in Listing 10-4 accordingly.

Listing 10-4. *Validating a LegalXML eContract Against the DTD*

```
DECLARE @eContract nvarchar(max);

SET @eContract = N'<?xml version="1.0" encoding="UTF-16"?>

<!DOCTYPE contract
  SYSTEM "c:\ eContracts-Reference.dtd">

<contract xmlns="urn:oasis:names:tc:eContracts:1:0"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <metadata>
    <dc:creator>The Law Firm of Dewey, Cheatham, and Howe</dc:creator>
    <dc:date>2007-12-01</dc:date>
  </metadata>
  <title>
    <text>Sample AdventureWorks Bicycle Sales Contract</text>
  </title>
  . . .
</contract>';

SELECT dbo.fn_ValidateDTD(@eContract);
```

The `<!DOCTYPE. . .SYSTEM. . .>` declaration in the XML document indicates a validation against an external DTD. The `fn_ValidateDTD` function call invokes the SQLCLR function I created and returns 1 if the XML document passed in is valid, as in the example. If the XML document passed in is invalid according to the DTD, as in Listing 10-5, an exception is thrown by the SQLCLR UDF.

Listing 10-5. *Invalid XML Document Validation Against a DTD*

```
DECLARE @eContract nvarchar(max);

SET @eContract = N'<?xml version="1.0" encoding="UTF-16"?>

<!DOCTYPE contract
  SYSTEM "c:\eContracts-Reference.dtd">

<sales-invoice>
</sales-invoice>';

SELECT dbo.fn_ValidateDTD(@eContract);
```

The attempt to validate an invalid XML document against the LegalXML eContracts DTD in Listing 10-5 generates an error message like the following:

```
Msg 6522, Level 16, State 2, Line 11
A .NET Framework error occurred during execution of user-defined routine or
aggregate "fn_ValidateDTD":
System.Exception: Root element name must match the DocType name.
System.Exception:
    at Apress.Samples.ValidationDTD.fn_ValidateDTD(SqlString xml)
```

Accessing XML on the Web

Back in Chapter 2, I discussed using the OPENROWSET function to load XML documents from the file system into xml data type instances. But what if you want to access XML documents from an external source that is not the local file system? SQLCLR integration can help with this also. A very common requirement is to access XML documents and DTDs from remote and local web servers. A very common XML application is Resource Description Framework (RDF) Site Summary (also known as “Really Simple Syndication,” or RSS), which is used to publish blogs and other content to content aggregators and reader software. An in-depth description of RSS is beyond the scope of this discussion, but I will consider a short sample of an RSS feed from the SQL Server Central web site. The XML snippet in Figure 10-1 shows a subset of a SQL Server Central RSS feed.



Figure 10-1. SQL Server Central RSS feed (partial)

Defining the details of the RSS feed format is beyond the scope of this book, but for the most part, the format is self-describing. Information about the RSS feed itself and the publisher is available as elements, like the required title, link, and description directly below the channel element. Each published article is listed as an item element with descriptive subelements, like title, description, pubDate, and link.

In this example, I will use a SQLCLR user-defined function to retrieve the RSS feed from the SQL Server Central web site and shred it to relational format. This type of utility could be used to store and aggregate content directly in SQL Server, where the server's advanced querying and reporting capabilities could be used to retrieve links to relevant content quickly and easily.

The SQLCLR `fn_GetRemoteXML` user-defined function accepts a Uniform Resource Identifier (URI) as a parameter and retrieves an XML file from the specified location. In this case, I am retrieving the RSS feed previously shown from the SQL Server Central web site at the URI `http://www.sqlservercentral.com/Xml/Rss/Articles/SQL+Server+2008`. This RSS feed contains a list of SQL Server Central articles tagged for SQL Server 2008. The `fn_GetRemoteXML` function is a relatively simple function written in C#, as shown in Listing 10-6.

Listing 10-6. C# Source for `fn_GetRemoteXML` SQLCLR Function

```
using System.Data.SqlTypes;
using System.Net;
using System.IO;
using System.Text;

namespace Apress.Sample
{
    public partial class UserDefinedFunctions
    {
```

```

[Microsoft.SqlServer.Server.SqlFunction]
public static SqlXml fn_GetRemoteXML(SqlString uri)
{
    WebClient wc = new WebClient();
    byte[] reqXML = wc.DownloadData(uri.Value);
    MemoryStream ms = new MemoryStream(reqXML);
    return new SqlXml(ms);
}
};
}

```

The function creates a `System.Net.WebClient` object and downloads the file specified by the URI passed in to a byte array.

```

WebClient wc = new WebClient();
byte[] reqXML = wc.DownloadData(uri.Value);

```

Then it creates a `System.IO.MemoryStream` from the document downloaded into the byte array.

```

MemoryStream ms = new MemoryStream(reqXML);

```

Finally, the function returns a `SqlXml` instance created from the `MemoryStream`.

```

return new SqlXml(ms);

```

The `fn_GetRemoteXML` user-defined function end result is that a well-formed XML file can be retrieved from a web server and returned as an `xml` data type instance. Listing 10-7 retrieves an RSS feed from the SQL Server Central web site, and then uses the `xml` data type `nodes()` method to shred the result into relational format. The resulting RSS feed in relational format is shown in Figure 10-2.

Listing 10-7. *Retrieving and Shredding an RSS Feed*

```

DECLARE @rss xml;

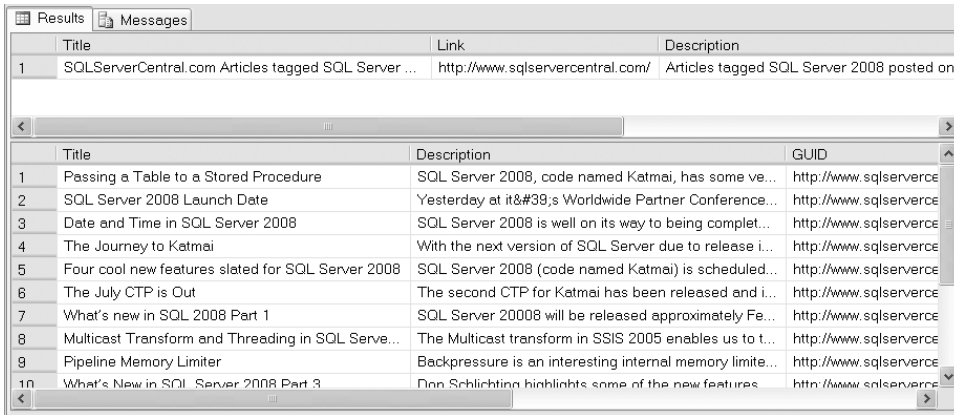
SET @rss = dbo.fn_GetRemoteXML('http://www.sqlservercentral.com/Xml/Rss/➤
Articles/SQL+Server+2008');

SELECT Col.value('title[1]', 'varchar(200)') AS Title,
       Col.value('link[1]', 'varchar(200)') AS Link,
       Col.value('description[1]', 'varchar(800)') AS [Description],
       Col.value('language[1]', 'varchar(20)') AS Lang,
       Col.value('ttl[1]', 'int') AS TTL,
       Col.value('managingEditor[1]', 'varchar(100)') AS ManagingEditor
FROM @rss.nodes('/rss/channel') AS Feed(Col)

SELECT Col.value('title[1]', 'varchar(200)') AS Title,
       Col.value('description[1]', 'varchar(800)') AS [Description],
       Col.value('guid[1]', 'varchar(200)') AS [GUID],

```

```
Col.value('pubDate[1]', 'datetime') AS [PubDate],
Col.value('link[1]', 'varchar(200)') AS [Link]
FROM @rss.nodes('/rss/channel/item') FeedItems (Col);
```



	Title	Link	Description
1	SQLServerCentral.com Articles tagged SQL Server ...	http://www.sqlservercentral.com/	Articles tagged SQL Server 2008 posted on

	Title	Description	GUID
1	Passing a Table to a Stored Procedure	SQL Server 2008, code named Katmai, has some ve...	http://www.sqlservice
2	SQL Server 2008 Launch Date	Yesterday at it's Worldwide Partner Conference...	http://www.sqlservice
3	Date and Time in SQL Server 2008	SQL Server 2008 is well on its way to being complet...	http://www.sqlservice
4	The Journey to Katmai	With the next version of SQL Server due to release i...	http://www.sqlservice
5	Four cool new features slated for SQL Server 2008	SQL Server 2008 (code named Katmai) is scheduled...	http://www.sqlservice
6	The July CTP is Out	The second CTP for Katmai has been released and i...	http://www.sqlservice
7	What's new in SQL 2008 Part 1	SQL Server 2008 will be released approximately Fe...	http://www.sqlservice
8	Multicast Transform and Threading in SQL Serve...	The Multicast transform in SSIS 2005 enables us to t...	http://www.sqlservice
9	Pipeline Memory Limiter	Backpressure is an interesting internal memory limite...	http://www.sqlservice
10	What's New in SQL Server 2008 Part 3	Dnn Schlichting highlights some of the new features...	http://www.sqlservice

Figure 10-2. Result of shredding an RSS feed

You can use the same type of SQLCLR UDF to retrieve text and HTML files as well, although with the current state of the Web, you can't rely on most web pages being well-formed XHTML. In fact, it's safe to say that the vast majority of web pages out there are not in compliance with the XHTML standard, so be cautious if you are attempting to download and parse web pages.

REST Services

In 2000 Roy Fielding, one of the authors of the HTTP specification, introduced the concept of Representational State Transfer (REST) services in his doctoral dissertation. Fielding describes REST in terms of a network architectural principle, and it was one of the founding principles of the World Wide Web. The concept of REST has been applied in the context of web services, the result of which is a simplification of data access and manipulation over the standard HTTP Simple Object Access Protocol (SOAP) web service model. Using SQLCLR, SQL Server can take advantage of web-based REST services. I will talk about how to access REST services in this section.

Note Though REST is well-defined, there are some web-based services that claim REST conformance but, in reality, are simply XML over HTTP transfers that are not designed on REST principles. For our purposes the difference between true REST services and plain old XML over HTTP is unimportant.

A good example of a web-based REST service is the Yahoo! Geocoding Application Programming Interface (API). This API accepts an address and returns the associated latitude and longitude information for mapping applications. The service can be accessed in its simplest form via the URI `http://local.yahooapis.com/MapsService/V1/geocode` with an appid

and a location parameter. The `appid` parameter is a unique application identifier assigned to your application when you register with Yahoo! Developer Network. The location parameter is a comma-separated physical location identifier containing, at minimum, a combination of street address, city, state, and/or ZIP code. I will use the default `appid` of `YahooDemo` in the demonstration code in this section. You can register for a unique `appid` at <http://developer.yahoo.com/maps/rest/V1/geocode.html>.

Tip At the time of writing, Yahoo! limits the number of requests by IP address to 5,000 per day. This rate is subject to change, and you can get the latest request limit information at the web address given previously.

To access the Yahoo! Geocoding API REST service, I'll first create a SQLCLR assembly that performs a Yahoo! Geocoding API REST service request and returns the `xml` type result. Listing 10-8 is the C# source code for this SQLCLR function.

Listing 10-8. *fn_YahooGeocodeREST Geocoding Function*

```
using System.Data.SqlTypes;
using System.Net;
using System.IO;
using System.Text;

namespace Apress.Sample
{
    public partial class UserDefinedFunctions
    {
        [Microsoft.SqlServer.Server.SqlFunction]
        public static SqlXml fn_YahooGeocodeREST(SqlString location)
        {
            WebClient wc = new WebClient();
            string uri = string.Format("http://local.yahooapis.com/" +
                "MapsService/V1/geocode?appid=YahooDemo&location={0}",
                location.Value);
            byte[] reqXML = wc.DownloadData(uri);
            MemoryStream ms = new MemoryStream(reqXML);
            return new SqlXml(ms);
        }
    };
}
```

Note Because we are accessing the Web with this function, the assembly requires external access permissions. The compilation and installation for this SQLCLR function is similar to those given in Chapter 8.

The function is similar in form and function to the example in the previous section, with the exception that the URI address is hard coded to a specific service location with the location parameter added at the last possible moment.

```
string uri = string.Format(http://local.yahooapis.com/" +
    "MapsService/V1/geocode?appid=YahooDemo&location={0}",
    location.Value);
```

To use the function, simply pass in a comma-separated physical location. Listing 10-9 demonstrates by geocoding the address of Madison Square Garden in New York City. The result of the sample call to this function is the XML document shown in Figure 10-3.

Listing 10-9. Geocoding Madison Square Garden

```
SELECT dbo.fn_YahooGeocodeREST('3 Penn Plaza, New York, NY');
```

The full URI generated by the function, with parameters, is shown by the following:

```
http://local.yahooapis.com/MapsService/V1/geocode?appid=YahooDemo&
location=3 Penn Plz, New York, NY
```

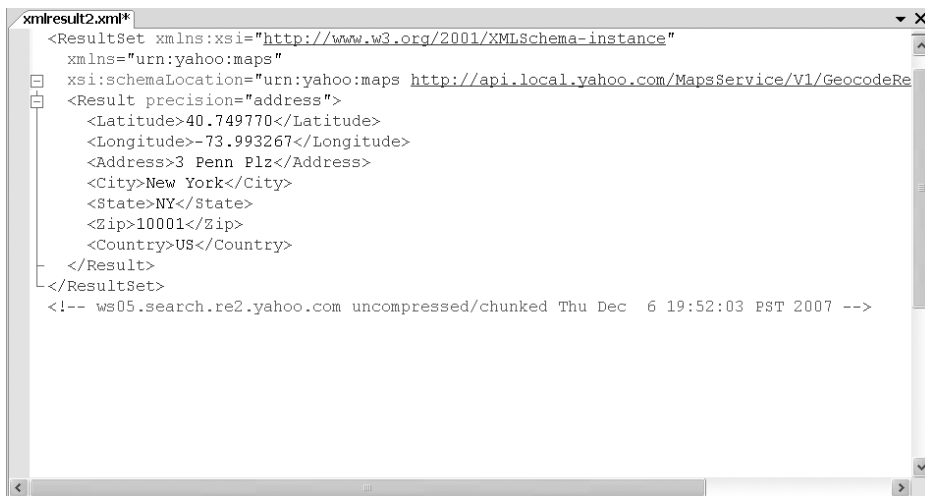


Figure 10-3. Result of Yahoo! Geocoding API REST service

The simplicity of REST-style (or “RESTful”) services makes them enticing to developers, and there are many available for a wide variety of applications. Another service I will consider in the next and final example is the eBay Shopping Services API. This API offers REST services that allow you to perform many of the same tasks that you can perform directly from eBay’s web site. By modifying the previous code sample, you can access eBay’s REST services to perform item searches.

Note In order to use the eBay REST services you have to register with the eBay Developers Program at <https://developer.ebay.com/join/> to get a unique appid. Your appid should replace the sample appid in the indicated place in the source code. eBay provides a video introduction to its development program at http://ebay.custhelp.com/cgi-bin/ebay.cfg/php/enduser/std_adp.php?p_faqid=1193.

The code in Listing 10-10 shows a simple eBay REST service call to perform an item search.

Listing 10-10. *eBay REST Service Item Search Function*

```
using System.Data.SqlTypes;
using System.Net;
using System.IO;
using System.Text;

namespace Apress.Sample
{
    public partial class UserDefinedFunctions
    {
        [Microsoft.SqlServer.Server.SqlFunction]
        public static SqlXml fn_eBaySearchREST(SqlString keywords)
        {
            WebClient wc = new WebClient();

            // Replace the following with your own appid
            string appid = "eBayAPID-73f4-45f2-b9a3-c8f6388b38d8";

            string uri = string.Format("http://open.api.ebay.com/shopping?" +
                "callname=FindItems&QueryKeywords={0}&appid={1}&version=517&" +
                "MaxEntries=50", keywords.Value, appid);

            byte[] reqXML = wc.DownloadData(uri);
            MemoryStream ms = new MemoryStream(reqXML);
            return new SqlXml(ms);
        }
    };
}
```

Don't forget to replace the appid in the following line with your own appid:

```
// Replace the following with your own appid
string appid = "eBayAPID-73f4-45f2-b9a3-c8f6388b38d8";
```

The URI to the eBay REST item search service is generated right before the call is made.

```
string uri = string.Format("http://open.api.ebay.com/shopping?" +
    "callname=FindItems&QueryKeywords={0}&appid={1}&version=517&" +
    "MaxEntries=50", keywords.Value, appid);
```

To search for *Star Wars*-related items on eBay, call the SQLCLR function with the appropriate keywords. The search function accepts a space-delimited list of keywords to search for, as shown in Listing 10-11.

Listing 10-11. *Searching eBay for “Star Wars”*

```
SELECT dbo.fn_eBaySearchREST('Star Wars');
```

The complete URI generated by this query is shown in the following (the appid will be different once you replace it with your own):

```
http://open.api.ebay.com/shopping?callname=FindItems&
QueryKeywords=Star Wars&appid=eBayAPID-73f4-45f2-b9a3-c8f6388b38d8&
&version=517&MaxEntries=50
```

The result of the search is shown in Figure 10-4.

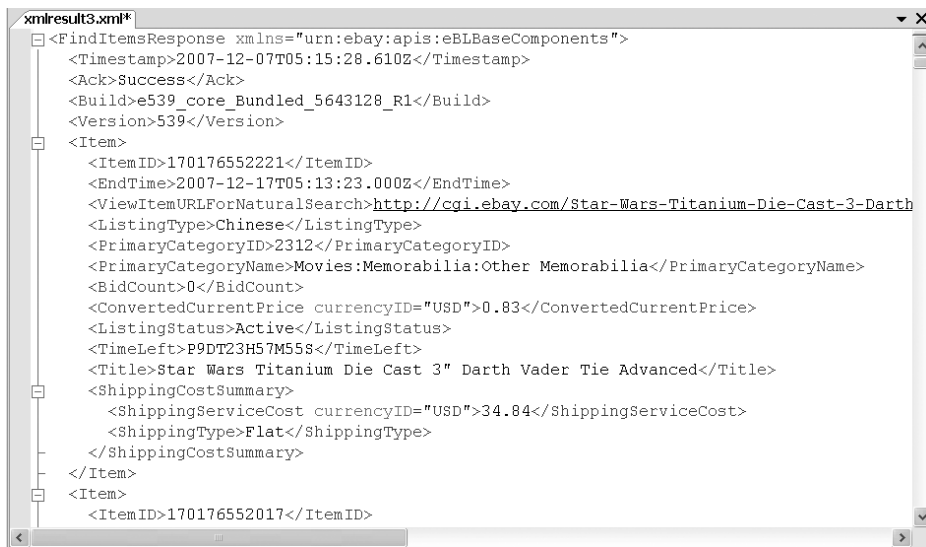


Figure 10-4. *Result of eBay item search REST service*

The eBay item search REST service retrieves complete information on currently available items on eBay that match your search criteria. Once you have retrieved this information, you can easily shred it, query it, and otherwise manipulate it using XQuery and the xml data type methods.

Note A full description of eBay's REST services is beyond the scope of this book, but detailed information is available at the eBay Developers Program REST Developer Center located at <http://developer.ebay.com/developercenter/rest/Default.aspx>.

The true power of REST services is simplicity, but that simplicity is also REST's shortcoming. For instance, REST services do not have any built-in security mechanisms defined, and they rely totally on the security provided by the network and other hardware and software solutions. REST services also do not have any standard mechanism defined for serializing and deserializing complex objects. Because of their simplicity, REST-style services are extremely useful for generating highly portable, relatively low overhead data access and manipulation services quickly and easily. This style of service is especially useful for exposing legacy data to other systems.

.NET XML Classes

The .NET Framework offers quite a bit of XML functionality via its XML classes. Many of these classes are available through the `System.Xml` namespace, although additional XML functionality is available through other namespaces. In this section, I will look at some of the XML functionality that can be particularly useful from a SQLCLR development perspective.

System.Xml Namespace

The `XmlNode` class represents a single self-contained XML node instance, with all the functionality for navigating Document Object Model (DOM) trees, performing XPath node selection, and editing DOM instances. As the base class for most .NET `System.Xml` classes, `XmlNode` is by far the most important. `XmlNode` is used to iterate through the results of XML DOM node selection methods.

I previously used the `XmlDocument` class in Chapter 8 to perform XML Stylesheet Language (XSL) transformations. `XmlDocument` is a very important class for manipulating and querying XML data in the .NET Framework. `XmlDocument` is derived from the `XmlNode` class, representing XML documents. `XmlDocument` exposes DOM instance attributes and elements via the XML DOM. A DOM instance is a treelike representation of an XML document that exposes its contents via collections. Consider the XML document in Listing 10-12, which is a simplified version of an actual AdventureWorks sales order header.

Listing 10-12. *Simplified Sales Order Header*

```
<sales-order-header>
  <row id = "71805">
    <order-date>2004-06-01T00:00:00</order-date>
    <account-number>10-4020-000573</account-number>
    <total-due amount = "76535.5524" />
    <contact>
      <first-name>Pilar</first-name>
      <last-name>Ackerman</last-name>
    </contact>
  </row>
</sales-order-header>
```

Loading this sample XML document into a .NET `XmlDocument` object generates an internal XML DOM instance that resembles the tree structure in Figure 10-5.

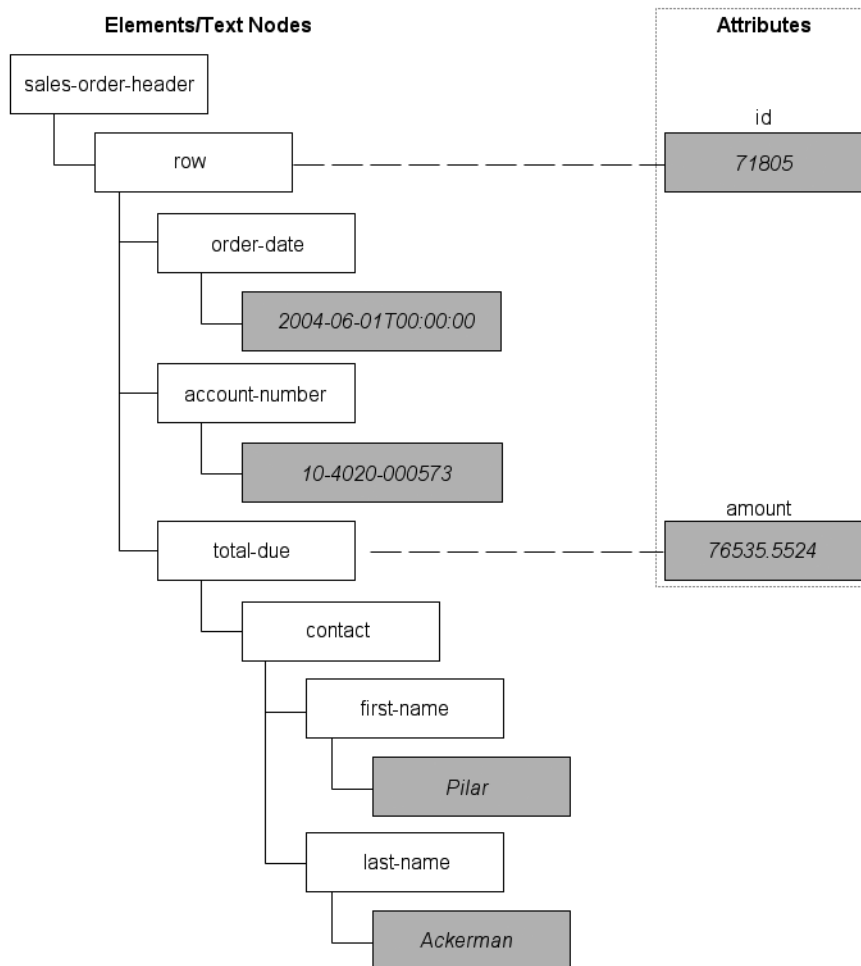


Figure 10-5. Internal representation of an XML DOM instance

The `XmlDocument` class exposes methods that allow you to navigate, manipulate, query, and serialize XML DOM instances. The `XmlDocument` class is the main class you will use when accessing XML functionality through the SQLCLR.

`XmlNodeList` is another important `System.Xml` class that represents an ordered collection of `XmlNode` instances. The simple .NET example in Listing 10-13 demonstrates using the `XmlDocument`, `XmlNodeList`, and `XmlNode` classes to perform a simple XPath expression against an XML document and print the results on the console.

Listing 10-13. Simple .NET XPath Example

```
using System;
using System.Xml;
```

```

namespace Apress.Samples
{
    class XmlDocumentExample
    {
        static void Main(string[] args)
        {
            XmlDocument xd = new XmlDocument();
            xd.LoadXml("<?xml version = \"1.0\"?>" +
                "<sales-order>" +
                "  <row id=\"71805\">" +
                "    <order-date>2004-06-01T00:00:00</order-date>" +
                "    <account-number>10-4020-000573</account-number>" +
                "    <order-amounts total-due=\"76535.5524\">" +
                "      <sub-total>69262.9433</sub-total>" +
                "      <freight>1731.5736</freight>" +
                "      <tax-amount>5541.0355</tax-amount>" +
                "    </order-amounts>" +
                "    <contact>" +
                "      <name>" +
                "        <first>Pilar</first>" +
                "        <last>Ackerman</last>" +
                "      </name>" +
                "      <email>pilar1@adventure-works.com</email>" +
                "      <phone>1 (11) 500 555-0132</phone>" +
                "    </contact>" +
                "  </row>" +
                "</sales-order>");
            XmlNodeList x1 = xd.SelectNodes("//order-amounts/*");
            foreach (XmlNode xn in x1)
            {
                Console.WriteLine(String.Format("{0} = {1}", xn.Name,
                    xn.InnerText));
            }
        }
    }
}

```

In the example, the `LoadXml` method of the `XmlDocument` is used to populate an `XmlDocument` instance.

```

XmlDocument xd = new XmlDocument();
xd.LoadXml("<?xml version = \"1.0\"?>" +
    "<sales-order>" +
    "  <row id=\"71805\">" +
    "    <order-date>2004-06-01T00:00:00</order-date>" +
    "    <account-number>10-4020-000573</account-number>" +
    "    <order-amounts total-due=\"76535.5524\">" +
    "      <sub-total>69262.9433</sub-total>" +

```

```

"    <freight>1731.5736</freight>" +
"    <tax-amount>5541.0355</tax-amount>" +
"  </order-amounts>" +
"  <contact>" +
"    <name>" +
"      <first>Pilar</first>" +
"      <last>Ackerman</last>" +
"    </name>" +
"    <email>pilar1@adventure-works.com</email>" +
"    <phone>1 (11) 500 555-0132</phone>" +
"  </contact>" +
" </row>" +
"</sales-order>");

```

Next the `SelectNodes` method of the `XmlDocument` applies an XPath expression to retrieve all child nodes of the `order-amounts` element. The results of `SelectNodes` are returned as an `XmlNodeList`.

```
XmlNodeList x1 = xd.SelectNodes("//order-amounts/*");
```

Finally the `XmlNode` class is used to iterate the `XmlNodeList` and print the `Name` and `InnerText` of each node returned by the XPath expression. The results are shown in Figure 10-6.

```

foreach (XmlNode xn in x1)
{
    Console.WriteLine(String.Format("{0} = {1}", xn.Name,
        xn.InnerText));
}

```

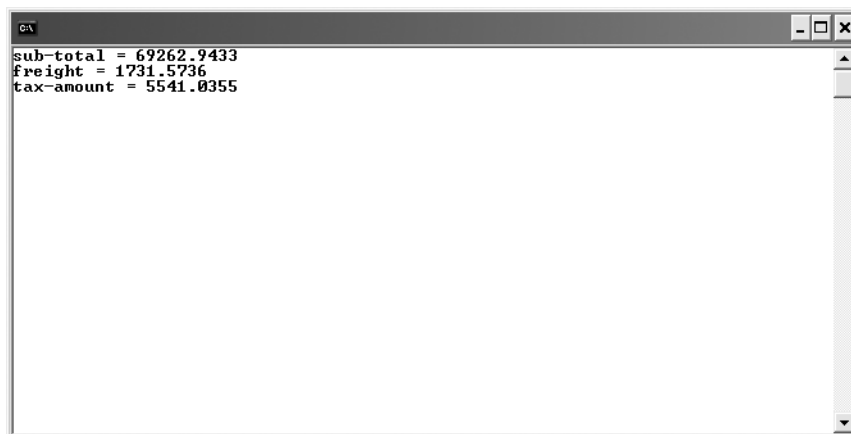


Figure 10-6. Result of .NET `SelectNodes` example

Another option for quickly retrieving nodes from your `XmlDocument` is the `XmlNodeReader` class. This class provides fast, forward-only access to a given `XmlNode`. You can use this class to quickly navigate a subtree anywhere in your XML DOM instance. The example in Listing 10-14 demonstrates this method.

Listing 10-14. *.NET XmlNodeReader Sample*

```

using System;
using System.Xml;

namespace Apress.Samples
{
    class XmlNodeReader_Example
    {
        static void Main(string[] args)
        {
            XmlDocument xd = new XmlDocument();
            xd.LoadXml("<?xml version = \"1.0\"?>" +
                "<sales-order>" +
                "  <row id=\"71805\">" +
                "    <order-date>2004-06-01T00:00:00</order-date>" +
                "    <account-number>10-4020-000573</account-number>" +
                "    <order-amounts total-due=\"76535.5524\">" +
                "      <sub-total>69262.9433</sub-total>" +
                "      <freight>1731.5736</freight>" +
                "      <tax-amount>5541.0355</tax-amount>" +
                "    </order-amounts>" +
                "    <contact>" +
                "      <name>" +
                "        <first>Pilar</first>" +
                "        <last>Ackerman</last>" +
                "      </name>" +
                "      <email>pilar1@adventure-works.com</email>" +
                "      <phone>1 (11) 500 555-0132</phone>" +
                "    </contact>" +
                "  </row>" +
                "</sales-order>");

            XmlNode xn = xd.SelectSingleNode("//contact");
            if (xn != null)
            {
                XmlNodeReader xr = new XmlNodeReader(xn);
                while (xr.Read())
                {
                    Console.WriteLine(String.Format("{0} {1}", xr.Name,
                        xr.Value));
                }
            }
        }
    }
}

```

In this example, the `XmlDocument` is populated exactly as before, using an XML string. Then the `SelectSingleNode` method uses XPath to select the single contact node in the XML document.

```
XmlNode xn = xd.SelectSingleNode("//contact");
```

The `XmlNodeReader` is created on the single node contained in the `XmlNode` instance. Then you use the `Read` method of the `XmlNodeReader` to recursively iterate the nodes beneath the contact node and print the `Name` and `Value` of each node to the console. The result is shown in Figure 10-7.

```
if (xn != null)
{
    XmlNodeReader xr = new XmlNodeReader(xn);
    while (xr.Read())
    {
        Console.WriteLine(String.Format("{0} {1}", xr.Name,
            xr.Value));
    }
}
```

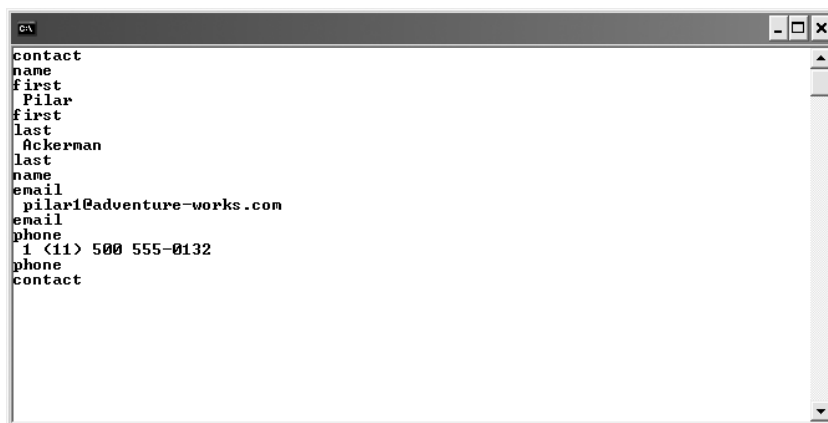


Figure 10-7. Result of iterating XML with `XmlNodeReader`

Notice that the results are recursively nested as the `XmlNodeReader` visits each child node and returns the node name on entry into the node and on exit from the node.

The `System.Xml` namespace is critical to .NET Framework XML parsing, querying, and manipulation in both client applications and in SQLCLR procedures and functions. The `System.Xml` namespace class methods and properties are well documented in the .NET Framework Library documentation, located at <http://msdn2.microsoft.com/en-us/library/system.xml.aspx>.

SqlXml Data Type

The `System.Data.SqlTypes` namespace includes a complete set of client-side SQL Server-specific data types that map directly to native server-side T-SQL types. An important aspect of these SQL Server-specific types is that they are nullable. This means that, unlike other native .NET

data types, like string and integer, these types can be set to SQL NULL. I've used the .NET `System.Data.SqlTypes.SqlXml` data type in several SQLCLR examples throughout this book, and now I'll take a look at some of its capabilities and limitations. The `SqlXml` type maps directly to the SQL Server `xml` data type, which makes it a no-brainer to transfer XML data between SQL Server and .NET in most cases. There are some exceptions, however, which I'll address later in this section.

Caution SQLCLR functions and procedures and client-side .NET code that use the `SqlTypes` usually require special handling for NULL values (although not always). If you don't include special handling, .NET will handle operations on NULL values through the standard exception-handling mechanisms.

While the `SqlXml` type maps to the SQL Server `xml` data type, there's no direct conversion to the .NET `System.Xml.XmlDocument` data type. You've encountered examples of this type of conversion throughout the code samples in this book so far. Fortunately, there are several methods available for converting your `SqlXml` data to an `XmlDocument`. One option is to use the `Load()` method provided by the .NET `XmlDocument` class and the `CreateReader()` method provided by the `SqlXml` class, as shown in Listing 10-15.

Listing 10-15. *Casting `SqlXml` to `XmlDocument` with `CreateReader()`*

```
private static XmlDocument CastSqlXml_XmlDoc_Reader(SqlXml xml)
{
    XmlDocument xd = new XmlDocument();
    xd.Load(xml.CreateReader());
    return xd;
}
```

Tip It's often useful to put these conversions into private functions in your SQLCLR or client-side code, especially if you need to perform more than one conversion between `SqlXml` and `XmlDocument` in your .NET code.

Alternatively you can make use of the `System.Text` encoding classes and the `System.IO.MemoryStream` classes to populate the `XmlDocument`. This also relies on the `SqlXml` type's `Value()` method, which returns its XML contents as a string. This method is shown in Listing 10-16.

Listing 10-16. *Casting `SqlXml` to `XmlDocument` with `MemoryStream`*

```
private static XmlDocument CastSqlXml_XmlDoc_MemStream(SqlXml xml)
{
    XmlDocument xd = new XmlDocument();
    byte[] buffer = UTF8Encoding.UTF8.GetBytes(xml.Value);
```

```

    MemoryStream ms = new MemoryStream(buffer);
    xd.Load(ms);
    return xd;
}

```

Casting an `XmlDocument` back to a `SqlXml` instance is similar. The first method is to create a `System.Xml.XmlTextReader` and pass it to the `SqlXml` constructor as shown in Listing 10-17.

Listing 10-17. *Casting from XmlDocument to SqlXml with XmlTextReader*

```

private static SqlXml CastXmlDoc_SqlXml_XmlTextReader(XmlDocument xmldoc)
{
    byte[] buffer = System.Text.UTF8Encoding.UTF8.GetBytes(xmldoc.OuterXml);
    MemoryStream ms = new MemoryStream(buffer);
    XmlTextReader xr = new XmlTextReader(ms);
    SqlXml x = new SqlXml(xr);
    return x;
}

```

You can also eliminate the `XmlTextReader` creation step and construct a `SqlXml` instance from the `MemoryStream`, as shown in Listing 10-18.

Listing 10-18. *Casting from XmlDocument to SqlXml with MemoryStream*

```

private static SqlXml CastXmlDoc_SqlXml_MemStream(XmlDocument xmldoc)
{
    byte[] buffer = System.Text.UTF8Encoding.UTF8.GetBytes(xmldoc.OuterXml);
    MemoryStream ms = new MemoryStream(buffer);
    SqlXml x = new SqlXml(ms);
    return x;
}

```

If you want to add special handling for SQL NULL arguments, above and beyond the built-in .NET exception handling, you can use the `IsNull` method of the `SqlXml` data type. Listing 10-19 modifies the previous Listing 10-15 to include special NULL handling. In this example, if a SQL NULL is passed in, the `XmlDocument` returned by the function is set to C# `null`.

Listing 10-19. *SqlXml to XmlDocument Conversion with NULL Handling*

```

private static XmlDocument CastSqlXml_XmlDoc(SqlXml xml)
{
    XmlDocument xd = new XmlDocument();
    if (xml.IsNull)
    {
        xd = null;
    }
    else
    {
        xd.Load(xml.CreateReader());
    }
}

```

```

    }
    return xd;
}

```

There are situations where the `SqlXml` type is not the correct choice for passing XML data from SQL Server to .NET code. An excellent example is the DTD validation function in the “XML Validation” section at the beginning of this chapter. The SQL Server `xml` data type strips DTDs from its XML content, so a DTD will never be passed to .NET via the `SqlXml` type. To get around this problem in situations where the raw XML data is required, use a `SqlString` type parameter as I did in Listing 10-3 of this chapter.

SqlCommand Options

The `System.Data.SqlClient.SqlCommand` class provides an interesting set of methods for reading XML data from SQL Server. The `ExecuteXmlReader`, `BeginExecuteXmlReader`, and `EndExecuteXmlReader` all support reading XML results. They are the XML equivalent to the `ExecuteScalar` method of the `SqlCommand` class.

Note If you use the `ExecuteScalar` or `ExecuteReader` methods to retrieve XML data from SQL Server, the results are returned in chunks of 2,033 bytes each. `ExecuteXmlReader` doesn’t suffer from this problem.

The `ExecuteXmlReader` method retrieves the XML result of a query and returns a .NET `XmlReader` that you can use to load an `XmlDocument` or otherwise manipulate. Listing 10-20 demonstrates using the `ExecuteXmlReader` method on a `SqlCommand` to retrieve the results of a FOR XML query, which is then used to populate an `XmlDocument`. The results are shown in Figure 10-8.

Listing 10-20. *ExecuteXmlReader Example Query*

```

using System;
using System.Data.SqlClient;
using System.Xml;
using System.IO;

namespace Apress.Samples
{
    class ExecuteXmlReader_Example
    {
        static void Main(string[] args)
        {
            string constr = "SERVER=SQL2008;INITIAL CATALOG=AdventureWorks;" +
                "INTEGRATED SECURITY=SSPI;";
            using (SqlConnection sqlcon = new SqlConnection(constr))
            {
                sqlcon.Open();

```

```

        using (SqlCommand sqlcmd = new SqlCommand("SELECT TOP (3) * " +
            "FROM Production.Product " +
            "FOR XML AUTO, ELEMENTS, ROOT('root');", sqlcon))
        {
            using (XmlReader xr = sqlcmd.ExecuteXmlReader())
            {
                XmlDocument xd = new XmlDocument();
                xd.Load(xr);
                Console.WriteLine(xd.OuterXml);
            }
        }
    }
    Console.WriteLine("=== Press any key to end. . . ===");
    Console.ReadKey(false);
}
}
}
}

```



Figure 10-8. *ExecuteXmlReader sample result*

The `BeginExecuteXmlReader` and `EndExecuteXmlReader` methods are the asynchronous versions of the `ExecuteXmlReader` method. `BeginExecuteXmlReader` begins asynchronous execution of the `SqlCommand`, and `EndExecuteXmlReader` ends the execution, returning an `XmlReader` with the result. Listing 10-21 demonstrates the asynchronous methods in action.

Listing 10-21. *Executing an XmlReader Asynchronously*

```

using System;
using System.Data.SqlClient;
using System.Xml;
using System.IO;

```

```

namespace Appress.Samples

```

```

{
    class AsyncExecuteXmlReader_Example
    {

        static void Main(string[] args)
        {
            string constr = "SERVER=SQL2008;INITIAL CATALOG=AdventureWorks;" +
                "INTEGRATED SECURITY=SSPI;ASYNCHRONOUS PROCESSING=true;";
            string cmdstr = "SELECT TOP (3) * " +
                "FROM Production.Product " +
                "FOR XML AUTO, ELEMENTS, ROOT('root');";
            Console.WriteLine("Async Call Start. . .");
            RunAsync(cmdstr, constr);
            Console.WriteLine("Async Call End. . .");
            Console.WriteLine("=== Press any key to end. . . ===");
            Console.ReadKey(false);
        }

        static private void RunAsync(string cmdstr, string constr)
        {
            using (SqlConnection sqlcon = new SqlConnection(constr))
            {
                sqlcon.Open();
                using (SqlCommand sqlcmd = new SqlCommand(cmdstr, sqlcon))
                {
                    IAsyncResult result = sqlcmd.BeginExecuteXmlReader();
                    int count = 0;
                    while (!result.IsCompleted)
                    {
                        Console.WriteLine("Waiting. . . ({0})", count++);
                        System.Threading.Thread.Sleep(1);
                    }
                    XmlReader xr = sqlcmd.EndExecuteXmlReader(result);
                    ShowXml(xr);
                }
            }
        }

        private static void ShowXml(XmlReader xr)
        {
            XmlDocument xd = new XmlDocument();
            xd.Load(xr);
            Console.WriteLine(xd.OuterXml);
        }
    }
}

```

In this example, the `BeginExecuteXmlReader` method is called to start asynchronous execution of the SQL statement. While the SQL statement is being executed, the program continues running, as demonstrated by the “Waiting. . .” messages displayed on the screen. Once the SQL statement has finished, the `EndExecuteXmlReader` method is called to retrieve the results, and then they are displayed in a console window. The results are shown in Figure 10-9.



Figure 10-9. Result of asynchronous `ExecuteXmlReader` methods

Tip In order to run SQL statements asynchronously, be sure to set the `ASYNCHRONOUS PROCESSING` option in your connection string to `true`, as shown in the sample code.

Additional .NET XML Support

As if the `System.Xml` namespace and `SqlXml` class were not enough, the .NET Framework provides XML support via several additional namespaces. Table 10-1 lists these additional XML support namespaces.

Table 10-1. .NET Framework XML Support Namespaces

Namespace	Description
System.Data.SqlClient	Contains classes that implement ADO.NET SQL Server native client connectivity and database access, including support for XML-based data retrieval.
System.IO	Provides basic stream and file-based input and output services, including memory streams and other types of streams that are useful when creating stream readers and writers for your XML data.
System.Net	Allows access to HTTP and other network communication protocols that are key to web services, REST services, and other HTML/XML retrieval requirements over the Web.

Namespace	Description
System.Text	Includes classes and methods for converting string data to arrays of bytes or from one encoding to another. These methods are useful when converting XML from string format to streams or when converting data between different UTF and Unicode encodings.
System.Xml.Linq	Provides classes that implement Microsoft's LINQ to XML in-memory XML processing functionality (.NET Framework 3.5).
System.Xml.Schema	Provides support for the W3C XML Schema recommendation.
System.Xml.Serialization	Contains classes that are used to serialize objects to XML documents or streams.
System.Xml.Serialization.Advanced	Includes classes that support customization of Web Services Description Language (WSDL) documents (.NET Framework 2.0 and higher).
System.Xml.Serialization.Configuration	Composed of internal classes used by the .NET framework to read configuration file information. Classes in this namespace are not meant to be accessed directly by developers.
System.Xml.XPath	Contains classes that allow you to navigate and edit XML data using a cursor-style model.
System.Xml.Xsl	Implements support for the W3C XSLT 1.0 Recommendation.
System.Xml.Xsl.Runtime	Provides internal support to the .NET Framework for XSL transformation processing. Classes in this namespace are not meant to be accessed directly by developers.

Much of the XML support in the .NET Framework is designed for client-side XML manipulation and querying. However, this functionality is very useful for implementing custom server-side XML functionality via the SQLCLR. As you've seen in this and previous chapters, this can be a very useful set of tools for performing XML-specific tasks that were not implemented directly in T-SQL.

Summary

In this chapter, I discussed .NET XML support. SQLCLR support is useful for accessing XML functionality not available directly through T-SQL. I also discussed some situations that lend themselves to SQLCLR functionality. Some of the cases considered include the following:

- **DTD validation.** SQL Server provides only limited support for DTD validation. Basically if an inline DTD is included, SQL Server checks it for syntax errors, expands entity references, and then strips the DTD from the XML. You may need to validate the structure of a document against legacy inline or external DTDs; this is functionality that .NET can provide.
- **XML access on the Web.** Accessing external Web-based XML resources, like RSS feeds and XHTML pages, is possible through SQLCLR. These types of resources can be easily retrieved and stored, manipulated, and queried on SQL Server.
- **Accessing REST services.** REST services provide a convenient and simple alternative to SOAP web services. SQL Server can use SQLCLR to access REST-based services over the Web and retrieve the results.

I also discussed the `SqLXml` data type, which is the .NET equivalent of the SQL Server `xml` type. The nullable `SqLXml` type is the key to passing XML data between SQL Server and .NET. During the discussion, I considered several methods of making your SQL Server XML data accessible to .NET and vice versa. Finally, I discussed the key .NET XML support classes with examples of their operation.

In the next chapter, I will discuss a new application of XML in SQL Server—Geography Markup Language support provided by the new SQL Server spatial data types.



Spatial Data and GML

SQL Server 2008 introduces powerful new data types to store, query, and manipulate spatial data. These new data types, geometry and geography, are designed to represent spatial data in accordance with the Open Geospatial Consortium (OGC) XML-based Geography Markup Language standard (GML). In this chapter, I will discuss the GML features of these new geometry and geography data types.

Spatial Data

The new geometry and geography data types are designed to store spatial data. In its simplest form spatial data is two-dimensional, consisting of a coordinate pair representing a point on a flat grid or, in more complex applications, a point on an ellipsoidal surface, like the earth's surface, as shown in Figure 11-1.

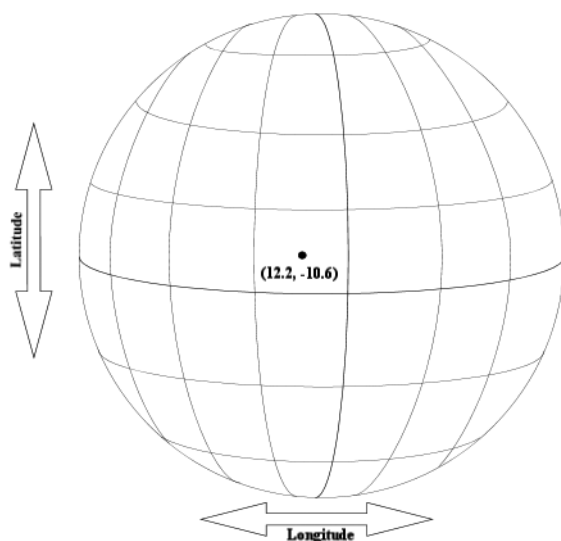


Figure 11-1. *Latitude and longitude pair on the earth's surface*

In addition, spatial data can include a third dimension, representing elevation, and an additional user-defined measure. The third dimension and additional measure are beyond the scope of this chapter, however, since it is not supported by the current GML standard. The commonly used two-dimensional coordinate pair is sufficient for our purposes.

In 2000, the OGC published the GML 1.0 Recommendation, “an XML encoding for the transport and storage of geographic information, including both the spatial and non-spatial properties of geographic features.” GML 1.0 was based on a combination of XML Document Type Definitions (DTDs) and the W3C’s XML Resource Description Framework (RDF), taking advantage of the RDF’s basic data-typing capabilities and simple type inheritance system. The RDF aspect of GML made it somewhat complex to use and slowed down widespread adoption of the standard. In 2001, the reworked and improved version 2.0 of GML was released. GML 2.0 is based on XML Schema, completely removing the RDF dependencies from the equation. The most recent version of the standard, GML 3.2.1, was published in 2007.

Realizing that geospatial database applications would be a very important aspect of the GML standard, the OGC published an additional standard defining SQL options for GML simple feature access. SQL Server implements version 1.1.0 of this standard with the geometry data type. The geometry data type is used to represent two-dimensional geospatial data, the planar or “flat earth” model. The geography data type stores data in an ellipsoidal or “round earth” model. Either of these types can be populated from three source formats: the well-known text (WKT) representation, well-known binary (WKB) representation, or the XML-based GML representation. I will focus on the GML representation of spatial data in this chapter.

Populating Spatial Data

The example in Listing 11-1 declares a geometry instance and populates it using the static `STGeomFromText()` method, using a WKT representation. Specifically this polygon represents the borders of the state of Delaware. The WKT declares a polygon geometric object defined by a set of comma-separated latitude/longitude (lat, long) coordinate pairs. The `AsGml()` method returns the GML representation of the geometric object defined.

Tip The geometry and geography data types include both instance and static methods. The instance methods act upon an instance (variable, column, etc.) of the data type, while the static methods are invoked directly from the data type. Instance type methods are invoked using the dot format, [e.g., `@instance.method()`], while static methods are called using the double colon format [e.g., `data_type::method()`].

Listing 11-1. Delaware Borders Defined

```
DECLARE @Delaware geometry;
```

```
SET @Delaware = geometry::STGeomFromText('POLYGON( (-75.70742 38.557476,
-75.71106 38.649551,-75.724937 38.83017,-75.752922 39.141548,
-75.761658 39.247753,-75.764664 39.295849,-75.772697 39.383007,
-75.791435 39.723755,-75.775269 39.724442,-75.745934 39.774818,
-75.695114 39.820347,-75.644341 39.838196,-75.583794 39.840008,
```

```
-75.470345 39.826435,-75.42083 39.79887,-75.412117 39.789658,
-75.428009 39.77813,-75.460754 39.763248,-75.475128 39.741718,
-75.476334 39.719971,-75.489639 39.714745,-75.610725 39.612793,
-75.562996 39.566723,-75.590187 39.463768,-75.515572 39.36694,
-75.402481 39.257637,-75.397728 39.073036,-75.324852 39.012386,
-75.307899 38.945911,-75.190941 38.80867,-75.083138 38.799812,
-75.045998 38.44949,-75.068298 38.449963,-75.093094 38.450451,
-75.350204 38.455208,-75.69915 38.463066,-75.70742 38.557476) ', 0);
```

```
SELECT @Delaware.AsGml() AS DelawareAsGml;
```

(X, Y) OR (LAT, LONG)?

In the WKT format, two-dimensional positions are defined as (x, y) pairs. This necessarily means that the longitude value (defined as the x axis) appears before the latitude value (defined as the y axis) in these pairs. This only applies to the geometry data type, however.

For the geography data type, coordinates are actually defined as (lat, long) pairs. This means that the y coordinate comes first, which is the exact opposite of the geometry data type. While this (lat, long) order might be more intuitive for beginners, it can introduce complications when trying to share spatial data with third-party applications that use (x, y) coordinate addressing exclusively or when trying to use the same GML data with both the geography and geometry data types. According to the SQL Server spatial data team, some potential changes to this are being actively considered.

The result of Listing 11-1 is the GML document shown in Figure 11-2.

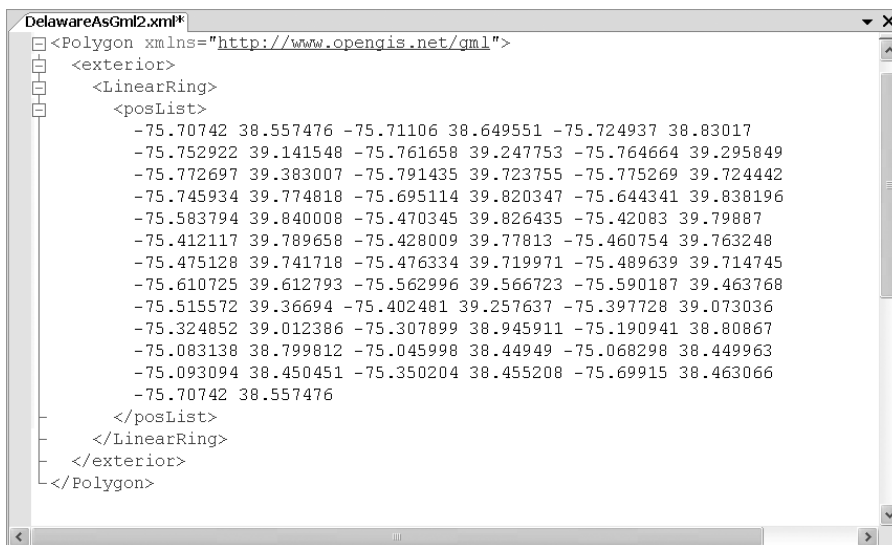


Figure 11-2. GML for the state of Delaware borders

You can likewise generate a geometry instance from a GML document using the static `GeomFromGml()` method, as shown in Listing 11-2.

Listing 11-2. *Populating a geometry Instance from GML*

```
DECLARE @Delaware geometry;

SET @Delaware = geometry::GeomFromGml(N'<gml:Polygon
  xmlns:gml = "http://www.opengis.net/gml">
  <gml:exterior>
    <gml:LinearRing>
      <gml:posList>
        -75.70742 38.557476 -75.71106 38.649551 -75.724937 38.83017
        -75.752922 39.141548 -75.761658 39.247753 -75.764664 39.295849
        -75.772697 39.383007 -75.791435 39.723755 -75.775269 39.724442
        -75.745934 39.774818 -75.695114 39.820347 -75.644341 39.838196
        -75.583794 39.840008 -75.470345 39.826435 -75.42083 39.79887
        -75.412117 39.789658 -75.428009 39.77813 -75.460754 39.763248
        -75.475128 39.741718 -75.476334 39.719971 -75.489639 39.714745
        -75.610725 39.612793 -75.562996 39.566723 -75.590187 39.463768
        -75.515572 39.36694 -75.402481 39.257637 -75.397728 39.073036
        -75.324852 39.012386 -75.307899 38.945911 -75.190941 38.80867
        -75.083138 38.799812 -75.045998 38.44949 -75.068298 38.449963
        -75.093094 38.450451 -75.350204 38.455208 -75.69915 38.463066
        -75.70742 38.557476
      </gml:posList>
    </gml:LinearRing>
  </gml:exterior>
</gml:Polygon>', 0);

SELECT @Delaware.ToString();
```

The `ToString()` method returns the contents of the geometry instance in WKT format, as shown in Figure 11-3.

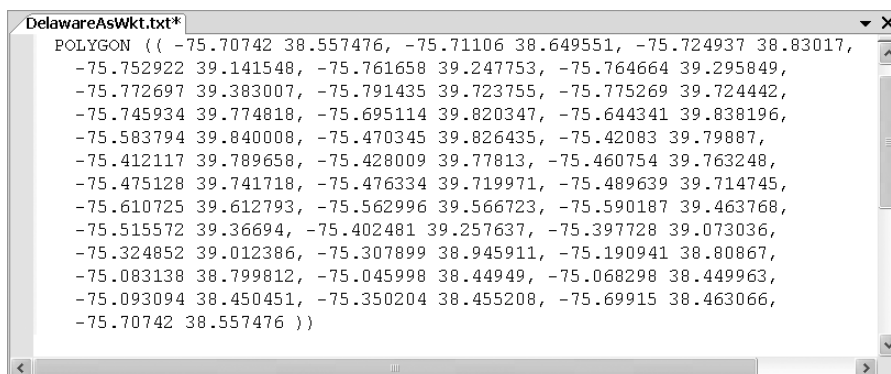


Figure 11-3. *Delaware state borders in WKT format*

Note The geometry and geography data types are SQL Common Language Runtime (SQLCLR) data types, so the method names of these data types are case sensitive. The `STGeomFromText()` method, for instance, will not be recognized as `stgeomfromtext()`.

Once you have your geometry data type instance declared and populated, you can use other data type methods on it as well. While these other data type methods are not the focus of this chapter, they can help you validate your instance data. Listing 11-3 builds on Listing 11-2 by checking to see if various predefined points are contained within the borders of the state of Delaware. For this example, I've decided to check the coordinates of the Delaware state capitol building and the coordinates of the Liberty Bell.

Listing 11-3. *Validating the geometry Instance Data with the `STContains()` Method*

```
DECLARE @Delaware geometry;

SET @Delaware = geometry::GeomFromGml(N'<gml:Polygon
  xmlns:gml="http://www.opengis.net/gml">
    <gml:exterior>
      <gml:LinearRing>
        <gml:posList>
          -75.70742 38.557476 -75.71106 38.649551 -75.724937 38.83017
          -75.752922 39.141548 -75.761658 39.247753 -75.764664 39.295849
          -75.772697 39.383007 -75.791435 39.723755 -75.775269 39.724442
          -75.745934 39.774818 -75.695114 39.820347 -75.644341 39.838196
          -75.583794 39.840008 -75.470345 39.826435 -75.42083 39.79887
          -75.412117 39.789658 -75.428009 39.77813 -75.460754 39.763248
          -75.475128 39.741718 -75.476334 39.719971 -75.489639 39.714745
          -75.610725 39.612793 -75.562996 39.566723 -75.590187 39.463768
          -75.515572 39.36694 -75.402481 39.257637 -75.397728 39.073036
          -75.324852 39.012386 -75.307899 38.945911 -75.190941 38.80867
          -75.083138 38.799812 -75.045998 38.44949 -75.068298 38.449963
          -75.093094 38.450451 -75.350204 38.455208 -75.69915 38.463066
          -75.70742 38.557476
        </gml:posList>
      </gml:LinearRing>
    </gml:exterior>
  </gml:Polygon>', 0);

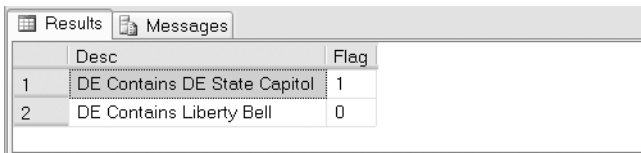
DECLARE @DelawareStateCapitol geometry,
  @LibertyBell geometry;
```

```
SET @DelawareStateCapitol = Geometry::GeomFromGml(N'<gml:Point
  xmlns:gml = "http://www.opengis.net/gml">
  <gml:pos>-75.522864 39.156473</gml:pos>
</gml:Point>', 0);
```

```
SET @LibertyBell = Geometry::GeomFromGml(N'<gml:Point
  xmlns:gml = "http://www.opengis.net/gml">
  <gml:pos>-75.15028 39.95028</gml:pos>
</gml:Point>', 0);
```

```
SELECT 'DE Contains DE State Capitol' AS [Desc],
  @Delaware.STContains(@DelawareStateCapitol) AS Flag
UNION
SELECT 'DE Contains Liberty Bell',
  @Delaware.STContains(@LibertyBell);
```

The results, shown in Figure 11-4, confirm that the Delaware state capitol building is located within the borders of the state of Delaware, while the Liberty Bell is not (in fact, it's located in Philadelphia, PA).



	Desc	Flag
1	DE Contains DE State Capitol	1
2	DE Contains Liberty Bell	0

Figure 11-4. Results of *STContains()* method test

GML

GML is the OGC standard for representation of geospatial data in XML format. SQL Server provides support for a subset of GML, allowing you to create geometry and geography data type instances from GML (or convert them to GML) using built-in data type methods. You saw simple examples of this type of GML conversion in the previous section. In this section, I'll look at the SQL Server implementation of GML and its features and limitations.

Geometric Objects

Geometric objects with varying degrees of complexity form the basis of GML. These data types include zero-, one-, and two-dimensional geometric objects. SQL Server GML supports seven concrete geometric objects that can be instantiated through GML. The SQL Server geometric object hierarchy is shown in Figure 11-5.

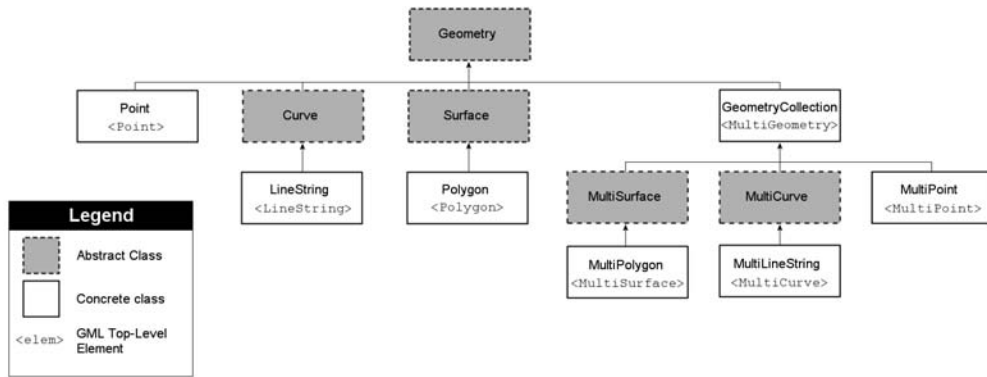


Figure 11-5. GML geometric object hierarchy

Each of the concrete geometric objects can be declared using the specified top-level GML element to create a geometry instance. Notice that some of the top-level elements have different names from the actual geometric objects they are used to declare; the `MultiPolygon` must be declared in GML by using the `MultiSurface` element, for instance. Examples of the SQL Server geometric objects are shown on a flat planar surface in Figure 11-6.

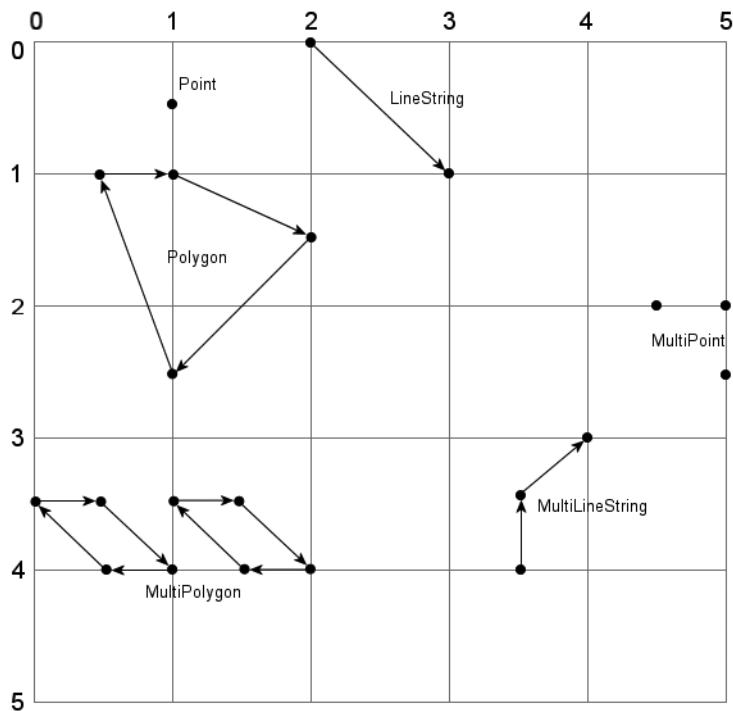


Figure 11-6. Geometric object examples

The example shows the characteristics of the different geometric objects. The following is a quick summary of the different types of objects shown:

- The **Point** object is a zero-dimensional object with no width, length, or surface area.
- The **MultiPoint** object is a collection of zero or more **Point** objects. Notice that the points in the **MultiPoint** collection are not connected to one another.
- The **LineString** object is a single straight line defined by a start and an end point. It is a one-dimensional object with a defined length but no width or surface area.
- The **MultiLineString** object is composed of zero or more **LineString** objects. In the example, the lines are connected to one another, but they needn't be.
- The **Polygon** object is a two-dimensional object defined by a series of connected points. The **Polygon** object must have a single exterior bounding ring, as shown in the example, and it may also have zero or more interior bounding rings defined. The **Polygon** object's bounding rings must enclose the area of the **Polygon**—that is the end point of the **Polygon** must be the same as the starting point.
- The **MultiPolygon** object is composed of zero or more **Polygon** objects.

Though not explicitly labeled in the example, the **GeometryCollection** object can be composed from a collection of one or more other geometric objects like those shown in Figure 11-6. Listing 11-4 uses GML to create the sample objects shown previously.

Listing 11-4. *Creating Sample Geometric Objects*

```
DECLARE @point geometry,
        @multipoint geometry,
        @linestring geometry,
        @multilinestring geometry,
        @polygon geometry,
        @multipolygon geometry,
        @multigeometry geometry;

/* Defining a Point */
SET @point = geometry::GeomFromGml(N'<Point
  xmlns="http://www.opengis.net/gml">
    <pos>0.5 1.0</pos>
</Point>', 0);

/* Defining a MultiPoint */
SET @multipoint = geometry::GeomFromGml('<MultiPoint
  xmlns="http://www.opengis.net/gml">
    <pointMembers>
      <Point>
        <pos>4.5 2</pos>
      </Point>
    <Point>
```



```

        <pos>5 2</pos>
    </Point>
    <Point>
        <pos>5 2.5</pos>
    </Point>
</pointMembers>
</MultiPoint>', 0);

/* Defining a LineString */
SET @linestring = geometry::GeomFromGml(N'<LineString
    xmlns="http://www.opengis.net/gml">
    <posList>2.0 0.0 3.0 1.0</posList>
</LineString>', 0);

/* Defining a MultiLineString */
SET @multilinestring = geometry::GeomFromGml('<MultiCurve
    xmlns="http://www.opengis.net/gml">
    <curveMembers>
        <LineString>
            <posList>3.5 4 3.5 3.5</posList>
        </LineString>
        <LineString>
            <posList>3.5 3.5 4 3</posList>
        </LineString>
    </curveMembers>
</MultiCurve>', 0);

/* Defining a Polygon */
SET @polygon = geometry::GeomFromGml(N'<Polygon
    xmlns="http://www.opengis.net/gml">
    <exterior>
        <LinearRing>
            <posList>0.5 1.0 1.0 1.0 2.0 1.5 1.0 2.5 0.5 1.0</posList>
        </LinearRing>
    </exterior>
</Polygon>', 0);

/* Defining a MultiPolygon */
SET @multipolygon = geometry::GeomFromGml(N'<MultiSurface
    xmlns="http://www.opengis.net/gml">
    <surfaceMembers>
        <Polygon>
            <exterior>
                <LinearRing>
                    <posList>1 3.5 1.5 3.5 2 4 1.5 4 1 3.5</posList>
                </LinearRing>
            </exterior>

```

```

    </Polygon>
    <Polygon>
      <exterior>
        <LinearRing>
          <posList>0 3.5 0.5 3.5 1 4 0.5 4 0 3.5</posList>
        </LinearRing>
      </exterior>
    </Polygon>
  </surfaceMembers>
</MultiSurface>', 0);

/* Defining a MultiGeometry containing a Polygon and MultiPoint */
SET @multigeometry = geometry::GeomFromGml('<MultiGeometry
xmlns="http://www.opengis.net/gml">
<geometryMembers>
  <Polygon>
    <exterior>
      <LinearRing>
        <posList>1 3.5 1.5 3.5 2 4 1.5 4 1 3.5</posList>
      </LinearRing>
    </exterior>
  </Polygon>
  <Polygon>
    <exterior>
      <LinearRing>
        <posList>0 3.5 0.5 3.5 1 4 0.5 4 0 3.5</posList>
      </LinearRing>
    </exterior>
  </Polygon>
  <MultiPoint>
    <pointMembers>
      <Point>
        <pos>5 2.5</pos>
      </Point>
      <Point>
        <pos>5 2</pos>
      </Point>
      <Point>
        <pos>4.5 2</pos>
      </Point>
    </pointMembers>
  </MultiPoint>
</geometryMembers>
</MultiGeometry>', 0);

```

Of course, you can also define these objects using the `Parse()`, `STGeomFromText()`, and other object-specific “FromText” methods available to the geometry and geography data types. (There are several methods that end in “FromText,” which are all covered by this statement,

for example, `STGeomFromText`, `STPointFromText`, `STMPolyFromText`, `STGeomCollFromText`, etc.). These methods use the WKT format to define the objects, however, which is beyond the scope of this discussion of GML. Once you have created a spatial data instance, regardless of the method used to create it, you can use the `AsGml()` method to retrieve the instance data in GML format.

POLYGONS, INTERIOR BOUNDING, AND MULTIPOLYGONS

As I mentioned earlier, `Polygon` objects can consist of one exterior bounding ring and zero or more interior bounding rings. The exterior bounding ring defines the outside of the `Polygon`, and the interior bounding rings remove additional surface area from the inside of a `Polygon`. While the OGC standards and Books Online provide some examples of `Polygon` objects with interior bounding rings, the real-world examples for non-GIS experts are lacking. With that in mind, I'm providing more concrete examples here, beginning with a simplified map of the state of Utah in Figure 11-7. Utah provides a real-world example of a `Polygon` with an exterior bounding ring and an interior bounding ring. Please note that this map is highly simplified to demonstrate the point.

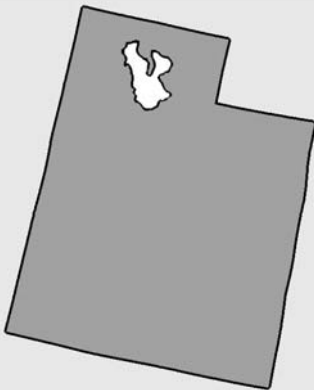


Figure 11-7. *State of Utah (simplified) as a Polygon*

The exterior bounding ring of the state of Utah is simply the state borders as defined by law. I chose Utah because it is a relatively simple `Polygon`, and it features a well-pronounced interior bounding ring, namely the Great Salt Lake. In Figure 11-7, the interior land area of the `Polygon` is shaded gray with a solid black exterior bounding ring. The interior bounding ring around the Great Salt Lake is also solid black. By defining an interior bounding ring around the Great Salt Lake, I am specifically excluding its area from the total shaded area of the `Polygon`. The area within the interior bounding ring is not considered part of the `Polygon` object. Some restrictions are imposed on interior bounding rings, namely that they cannot cross the exterior bounding ring, and they cannot cross other interior bounding rings.

`MultiPolygon` objects are useful for representing disjointed geographic areas. Geopolitical areas associated but disconnected from each other, like coastal territories and countries with islands under their possession, are prime examples. The state of Michigan is an excellent example of a `MultiPolygon` because it is composed of two peninsulas that are physically separated by Lakes Michigan and Huron. These two peninsulas, known as the Upper Peninsula and Lower Peninsula, respectively, are shown in the simplified map in Figure 11-8. Note that Lake Michigan is also shown to accentuate the physical separation of the peninsulas, but it would not be included as part of the `MultiPolygon` definition in this example.



Figure 11-8. *State of Michigan (simplified) as a MultiPolygon*

As you can see, Polygon objects are extremely useful for geospatial calculation and mapping applications. The concept of the interior bounding ring and MultiPolygon objects increases their usefulness by allowing you to easily create complex logical representations of geographic locations.

Elements of GML

As you saw in the previous section, GML defines many XML elements to allow you to create geometric objects. In this section, I'll discuss these XML elements in greater detail beginning with the Point element.

The Point element defines a Point geometric object. As I mentioned in the previous section, the Point object defines a solitary zero-dimensional point in coordinate space. GML supports points on a two-dimensional plane defined by a single (x, y) coordinate pair. The top-level GML element for a Point object is Point. Following are the coordinates of the Point defined within a nested pos element of the GML data:

```
<Point xmlns = "http://www.opengis.net/gml">
  <pos>x-coordinate y-coordinate</pos>
</Point>
```

Listing 11-5 shows a sample GML Point definition for the tallest mountain in the United States, Mount McKinley in Alaska.

Listing 11-5. *GML Point Definition for Mount McKinley*

```
DECLARE @g geometry;
SET @g = geometry::GeomFromGml ('<Point
  xmlns = "http://www.opengis.net/gml"> <!-- Mount McKinley, Alaska -->
  <pos>-151.006347 63.069042</pos>
</Point>', 0);
```

THREE DIMENSIONS AND BEYOND

GML does not support three-dimensional points, but the geometry and geography data types do. In addition to the (x, y) two-dimensional coordinates, the spatial data types support the third dimension of elevation (z). The third dimension can be included in your geometric object definitions by using the WKT format. To include the elevation of Mount McKinley in the previous `Point` example, you could use WKT to define the `Point`, as in the following example:

```
DECLARE @g geometry;
SET @g = geometry::STGeomFromText ('POINT(-151.006347 63.069042 6194)', 0);
```

The `Point` definition now includes an elevation (z) coordinate of 6,194 meters. Note that if you use the `AsGml()` method on this geometry instance, it will strip out the elevation coordinate again, leaving you with a two-dimensional `Point`. Using WKT, you can also include a fourth value in your `Point` definitions, known as a *measure*. The measure can be any user-defined measure. Examples of useful measures might include temperature, barometric pressure, wind speed, or other measurable physical attributes. As with the elevation coordinate, measures are not supported by GML. These limitations of GML make it less useful if you are working with spatial data requiring three-dimensional representation.

The `MultiPoint` object is a collection of zero-dimensional points, each point being defined by an (x, y) coordinate point. The top-level element for the `MultiPoint` is the `MultiPoint` element. The `MultiPoint` element contains a `pointMembers` element, which in turn contains a `Point` element for each `Point` object. And of course the `Point` element contains `pos` elements that define the (x, y) coordinate pair for each point. Following is the format of the `MultiPoint` element:

```
<MultiPoint xmlns = "http://www.opengis.net/gml">
  <pointMembers>
    <Point>
      <pos>
        x-coordinate y-coordinate
      </pos>
    </Point>
    . . .
  </pointMembers>
</MultiPoin
```

Listing 11-6 shows a sample `MultiPoint` GML document with a collection of points that define the three tallest mountains in the United States.

Listing 11-6. *MultiPoint Example with Tallest US Mountains*

```
DECLARE @g geometry;
SET @g = geometry::GeomFromGml ('<MultiPoint
  xmlns = "http://www.opengis.net/gml">
    <pointMembers>
      <Point> <!-- Mount McKinley, Alaska -->
        <pos>-151.006347 63.069042</pos>
      </Point>
```

```

<Point> <!-- Mount St. Elias, Alaska -->
  <pos>-140.930741 60.292682</pos>
</Point>
<Point> <!-- Mount Foraker, Alaska -->
  <pos>-151.399814 62.960408</pos>
</Point>
</pointMembers>
</MultiPoint>', 0);

```

The next object up is the `LineString`, which is defined by a set of points. The `LineString` has a length defined by these points, but no width or surface area, so it is considered a one-dimensional object. The `LineString` object is defined by a top-level `LineString` GML element containing a `posList` element. The `posList` is a space-separated list of (x, y) coordinates. A simple `LineString` object can consist of two points—a start point and an end point. But more complex `LineString` objects are possible, consisting of multiple line segments that are joined one to the next. Following is the GML format for the `LineString`:

```

<LineString xmlns = "http://www.opengis.net/gml">
  <posList>
    x-coordinate1 y-coordinate1 x-coordinate2 y-coordinate2 . . .
  </posList>
</LineString>

```

As an example, imagine a `LineString` that runs from Philadelphia, PA, to Trenton, NJ, and then on to New York City, NY. This sample `LineString` is shown in Figure 11-9. The GML definition of this `LineString` is shown in Listing 11-7.



Figure 11-9. *LineString from Philadelphia to Trenton to New York City*

Listing 11-7. *LineString Example*

```

DECLARE @g geometry;
SET @g = geometry::GeomFromGml ('<LineString
  xmlns = "http://www.opengis.net/gml">
  <!-- Straight line from Philadelphia to Trenton to New York City -->
  <posList>-75.25 39.88 -74.82 40.28 -73.98 40.77</posList>
</LineString>', 0);

```

The `MultiLineString` is a collection of `LineString` objects. Like the `LineString` object, the `MultiLineString` is a one-dimensional object with no width and no surface area. The `MultiLineString` object is defined in GML with a top-level `MultiCurve` element. The `MultiCurve` element contains a `curveMembers` subelement, and the `curveMembers` subelement in turn contains a series of `LineString` elements. As I previously discussed, the `LineString` elements contain `posList` elements with space-separated (x, y) pairs of coordinates defining each `LineString` object. Following is the format for the `MultiCurve` element:

```

<MultiCurve xmlns = "http://www.opengis.net/gml">
  <curveMembers>
    <LineString>
      <posList>
        x-coordinate1 y-coordinate1 x-coordinate2 y-coordinate2 . . .
      </posList>
    </LineString>
    . . .
  </curveMembers>
</MultiCurve>

```

When defining a `MultiLineString` object, you can define several individual `LineString` objects that are not necessarily connected from point to point. In the previous `LineString` example, the three sets of coordinates indicate two separate line segments, one from Philadelphia to Trenton and another from Trenton to New York City. Both line segments share a common coordinate for the city of Trenton. Trenton represents the end of one line segment and the beginning of the next.

The `MultiLineString` object allows you to create sets of `LineString` objects that may be disconnected from one another. Consider the example in Figure 11-10, which shows a `MultiLineString` object composed of two disconnected `LineString` objects. One of the `LineString` objects runs from Philadelphia to Pittsburgh and the second runs from Trenton to New York City. The sample `MultiLineString` in Figure 11-10 is shown in GML format in Listing 11-8.



Figure 11-10. *MultiLineString* example

Listing 11-8. *MultiLineString* GML Example

```
DECLARE @g geometry;
SET @g = geometry::GeomFromGml('<MultiCurve
  xmlns = "http://www.opengis.net/gml">
  <curveMembers>
    <LineString> <!-- Trenton to New York City -->
      <posList>-74.82 40.28 -73.98 40.77</posList>
    </LineString>
    <LineString> <!-- Philadelphia to Pittsburgh -->
      <posList>-75.25 39.88 -80.22 40.50</posList>
    </LineString>
  </curveMembers>
</MultiCurve>', 0);
```

The Polygon object is a step up from the zero- and one-dimensional objects you've seen so far. The Polygon is an enclosed area that is bounded by a single exterior bounding ring. As discussed previously, Polygon objects can include zero or more interior bounding rings to exclude a portion of the area inside the Polygon's exterior bounding ring from the Polygon's area.

The top-level GML element for a Polygon object is Polygon. The Polygon element contains an exterior subelement to define the exterior bounding ring and zero or more interior elements to define the interior bounding rings. The interior and exterior elements contain a LinearRing subelement, which includes the posList subelement containing the set of space-separated (x, y) coordinate pairs. The last coordinate pair in a LinearRing object must be the same as the first coordinate pair, ensuring that the LinearRing object forms an enclosed geometric object. Following is the format for defining a Polygon:


```

<Polygon xmlns = "http://www.opengis.net/gml">
  <exterior>
    <LinearRing>
      <poslist>
        x-coordinate1 y-coordinate1 x-coordinate2 y-coordinate2 . . .
      </poslist>
    </LinearRing>
  </exterior>
  <interior>
    <LinearRing>
      <poslist>
        x-coordinate1 y-coordinate1 x-coordinate2 y-coordinate2 . . .
      </poslist>
    </LinearRing>
  </interior>
  . . .
</Polygon>

```

Figure 11-11 shows a Polygon with an exterior bounding ring and a single interior bounding ring. The shaded area is the interior of the Polygon object, while the white area is considered the Polygon object's exterior.

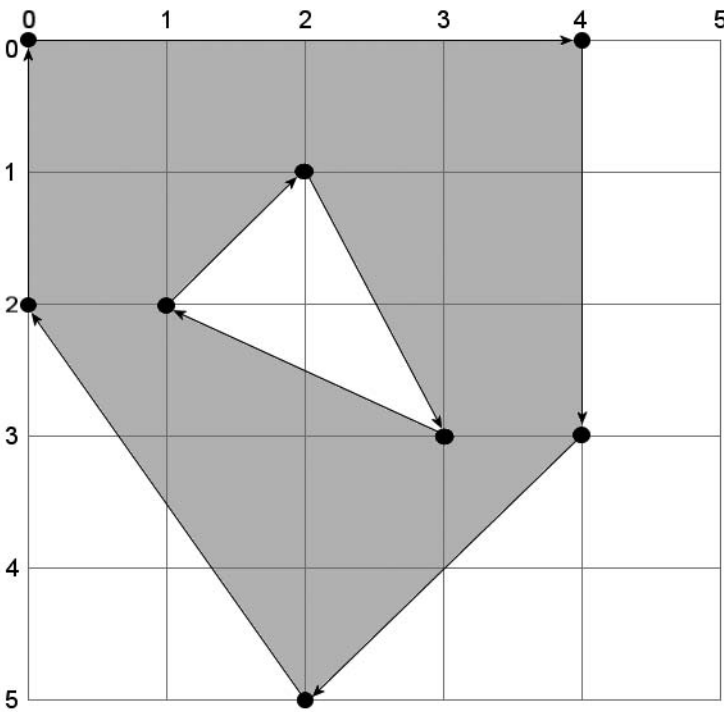


Figure 11-11. Polygon with single interior bounding ring

Listing 11-9 shows the GML document used to create the Polygon shown in Figure 11-11.

Listing 11-9. *Polygon GML Example*

```
DECLARE @g geometry;
SET @g = geometry::GeomFromGml(N'<Polygon
  xmlns="http://www.opengis.net/gml">
    <exterior>
      <LinearRing>
        <posList>0 0 4 0 4 3 2 5 0 2 0 0</posList>
      </LinearRing>
    </exterior>
    <interior>
      <LinearRing>
        <posList>2 1 3 3 1 2 2 1</posList>
      </LinearRing>
    </interior>
  </Polygon>', 0);
```

The MultiPolygon is composed of a collection of Polygon objects. The top-level element of a MultiPolygon is the MultiSurface element. The MultiSurface element contains a surfaceMembers subelement, which in turn contains Polygon elements, as shown in the following format:

```
<MultiSurface xmlns = "http://www.opengis.net/gml">
  <surfaceMembers>
    <Polygon>
      . . .
    </Polygon>
    . . .
  </surfaceMembers>
</MultiSurface>
```

The final geometric object, the GeometryCollection, is composed of a collection of other geometric objects. The top-level GML element for the GeometryCollection object is the MultiGeometry element. This element contains a subelement named geometryMembers, which in turn contains a collection of other GML geometric shape definitions, like Polygon and Point, as shown in the following format:

```
<MultiGeometry xmlns = "http://www.opengis.net/gml">
  <geometryMembers>
    . . .
  </geometryMembers>
</MultiGeometry>
```

Summary

The new SQL Server spatial data types, geometry and geography, promise to change the way SQL Server developers store, manipulate, and access spatial data in SQL Server. GML is one facet of the spatial data type implementation standard that can prove particularly useful for exchanging SQL Server spatial data with other systems. With the popularity of geospatial and mapping applications, SQL Server spatial data type support is a highly anticipated new feature.

In this chapter, I gave an overview of the SQL Server spatial data types and spatial data, including a discussion of the different types of geometric objects supported by the SQL Server spatial data types. I also discussed real-world examples using geographic locations to demonstrate the use of GML to populate geometry data types.

In the next chapter, I will discuss SQLXML support in SQL Server 2008.



SQLXML

Since the 2000 release, SQL Server has supported XML integration via SQLXML. SQLXML was designed as a “bridge” technology between XML and relational data. Much of the functionality available through SQLXML is now exposed by the SQL Server `xml` data type and other T-SQL and .NET functionality. In fact, Microsoft has announced plans to remove SQLXML from the SQL Server 2008 installer in the production release. SQLXML is still an important topic, however, because there are many applications that have been built on this functionality which will require ongoing support. In this chapter, I will discuss SQLXML features, functionality, and options.

SQLXML added functionality in support of SQL Server’s original implementation of XML support circa SQL Server 2000. SQLXML allows developers to perform a variety of tasks, including converting relational data to XML format on the client, querying and updating relational data using XPath and XML, and bulk loading XML data into relational tables. Although SQLXML was originally designed to work in unmanaged code through OLEDB (Object Linking and Embedding Database), the .NET Framework provides managed wrapper classes for SQLXML. In this chapter, I’ll give an overview of some of the important .NET Framework SQLXML support functionality in this chapter.

Note The .NET SQLXML example code introduced in this chapter relies on some external files. The source code looks for these files in the root directory of the C: drive. You will need to copy the `contactupdg.xsd`, `AWCustGeo.xsd`, and `AWCustGeo.xml` files to the `C:\` directory to run these examples, or you can change the source code to point to a directory of your choice and copy the files there.

Querying

One feature provided by SQLXML is the ability to perform a query and format the result set as XML on the client side. This is useful functionality if you want to run a stored procedure that generates a relational result set, but you want the result in XML format. Client-side relational-to-XML formatting is also a useful method for relieving some of the load on the server.

SQLXML queries are performed via the `Microsoft.Data.SqlXml.SqlXmlCommand` object. The `SqlXmlCommand` object exposes properties and methods for accessing SQLXML functionality from .NET. Because this class is a wrapper around the OLEDB/COM-based SQLXML functionality, the `SqlXmlCommand` connection string requires you to use the `SQLOLEDB` provider. When performing client-side formatting, SQLXML executes your query on the server, sans the `FOR XML` clause. It then uses the `FOR XML` clause to format the result on the client.

Figure 12-1 shows the results of the SQLXML Example application, which performs a SQLXML query. The sample application executes the AdventureWorks `uspGetEmployeeManagers` stored procedure on the server and formats the results on the client using the `FOR XML NESTED` clause. To run the sample query, enter an AdventureWorks employee ID in the appropriate text box and click the Query button.

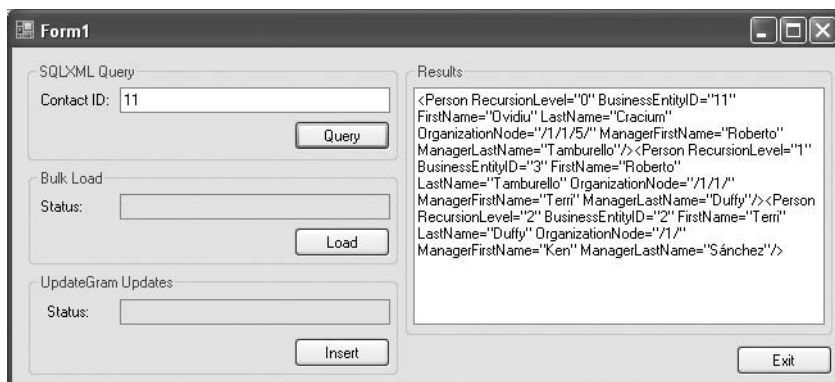


Figure 12-1. SQLXML sample query result

The following results, when 11 is entered into the Contact ID text box, are shown in a more easily readable formatted form:

```
<Person RecursionLevel="0"
  BusinessEntityID="11"
  FirstName="Ovidiu"
  LastName="Cracium"
  OrganizationNode="/1/1/5/"
  ManagerFirstName="Roberto"
  ManagerLastName="Tamburello"/>
```

```
<Person RecursionLevel="1"
  BusinessEntityID="3"
  FirstName="Roberto"
  LastName="Tamburello"
  OrganizationNode="/1/1/"
  ManagerFirstName="Terri"
  ManagerLastName="Duffy"/>
```

```
<Person RecursionLevel="2"
  BusinessEntityID="2"
  FirstName="Terri"
  LastName="Duffy"
  OrganizationNode="/1/"
  ManagerFirstName="Ken"
  ManagerLastName="Sánchez"/>
```

The code that performs this query is behind the Query button's Click event, as shown in Listing 12-1.

Listing 12-1. *Sample SQLXML Query Code*

```
string SqlXmlConnString = "Provider=SQLOLEDB;" +
    "Server=(local);" +
    "Database=AdventureWorks;" +
    "Integrated Security=SSPI";

private void btnQuery_Click(object sender, EventArgs ev)
{
    SqlXmlCommand cmd = new SqlXmlCommand(SqlXmlConnString);
    cmd.CommandText = "EXEC dbo.uspGetEmployeeManagers ? " +
        "FOR XML NESTED";
    cmd.ClientSideXml = true;

    SqlXmlParameter p;
    p = cmd.CreateParameter();
    p.Value = txtContactID.Text;

    try
    {
        Stream strm = cmd.ExecuteStream();
        strm.Position = 0;
        using (StreamReader sr = new StreamReader(strm))
        {
            txtResult.Text = sr.ReadToEnd();
        }
    }
    catch (SqlXmlException e)
    {
        e.ErrorStream.Position = 0;
        string s = new StreamReader(e.ErrorStream).ReadToEnd();
        System.Console.WriteLine(s);
    }
}
```

The first thing to notice is, as I mentioned previously, the Provider in the connection string is set to SQLOLEDB. Modify the other settings in the connection string as appropriate for your server.

```
string SqlXmlConnString = "Provider=SQLOLEDB;" +
    "Server=(local);" +
    "Database=AdventureWorks;" +
    "Integrated Security=SSPI";
```

An instance of the `SqlXmlCommand` object is created with the appropriate connection string, and the `CommandText` attribute is set to a parameterized query that calls the `uspGetEmployeeManagers` stored procedure with the `FOR XML NESTED` clause. The `ClientSideXml` attribute is also set to `true` to ensure that the XML formatting is performed client-side.

```
SqlXmlCommand cmd = new SqlXmlCommand(SqlXmlConnectionString);
cmd.CommandText = "EXEC dbo.uspGetEmployeeManagers ? " +
    "FOR XML NESTED";
cmd.ClientSideXml = true;
```

SQLXML can accept four types of `FOR XML` clauses: `FOR XML NESTED`, `FOR XML RAW`, `FOR XML EXPLICIT`, and `FOR XML AUTO`. `FOR XML AUTO` queries are performed on the server; all others are performed client-side. The employee ID is added to the `SqlXmlCommand` as a parameter via a `SqlXmlParameter` object.

```
SqlXmlParameter p;
p = cmd.CreateParameter();
p.Value = txtContactID.Text;
```

The actual execution takes place in the `try . . . catch` block by using the `SqlXmlCommand`'s `ExecuteStream` method. The results are returned by this function as a `Stream`, which is read by a `StreamReader` and converted to a string. The results are displayed in a `TextBox` on the form. Alternatively you can use the `SqlXmlCommand`'s `ExecuteXmlReader`, `ExecuteToStream`, or `ExecuteNonQuery` methods.

```
Stream strm = cmd.ExecuteStream();
strm.Position = 0;
using (StreamReader sr = new StreamReader(strm))
{
    txtResult.Text = sr.ReadToEnd();
}
```

The remaining code consists of some basic error reporting code. Note that in this example I've kept the error-handling code to a minimum to keep the logic easy to follow. In a production application, you would want to incorporate more thorough error handling, as well as defensive coding techniques.

Updategrams

Since SQL Server 2000, SQLXML has offered the ability to conduct data updates through XML via *updategrams*. Updategrams were initially implemented to support remote data manipulations including updates, inserts, and deletions over the Web. The idea behind updategrams is relatively simple. You present the server with XML-formatted “before” and “after” snapshots of given records, and SQLXML takes care of the rest.

The updategram is based on the annotated mapping schema. These are XML Data-Reduced (XDR) or XML Schema Definition (XSD) documents with additional annotations and attributes that define the XML-to-relational mappings of your updategrams. The SQLXML Example application includes a simple mapping schema called `contactupdg.xsd`, which is shown in Listing 12-2.

Tip SQLXML supports both XDR and XSD mapping schemas, but XDR support is deprecated. Use XSD mapping schemas instead, and start preparing to convert your old XDR schemas to XSD.

Listing 12-2. *Sample updategram Mapping Schema*

```
<?xml version = "1.0"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
  xmlns:sql = "urn:schemas-microsoft-com:mapping-schema"
  elementFormDefault = "qualified"
  attributeFormDefault = "unqualified"
  xmlns:msdata = "urn:schemas-microsoft-com:mapping-schema">

  <xs:annotation>
    <xs:appinfo>
      <msdata:relationship name = "Contact-Person"
        parent = "Person.BusinessEntity"
        parent-key = "BusinessEntityID"
        child = "Person.Person"
        child-key = "BusinessEntityID"/>
    </xs:appinfo>
  </xs:annotation>

  <xs:element name = "Contact"
    sql:relation = "Person.BusinessEntity">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "Id"
          type = "xs:integer"
          sql:field = "BusinessEntityID"
          sql:identity = "useValue"/>
        <xs:element name = "Person"
          sql:relation = "Person.Person"
          sql:relationship = "Contact-Person">
          <xs:complexType>
            <xs:attribute name = "id"
              type = "xs:string"
              use = "optional"
              sql:field = "BusinessEntityID" />
            <xs:attribute name = "person-type"
              type = "xs:string"
              use = "required"
              sql:field = "PersonType"/>
            <xs:attribute name = "name-style"
              type = "xs:integer"
              use = "required"
```

```

        sql:field = "NameStyle"/>
      <xs:attribute name = "last-name"
        type = "xs:string"
        use = "required"
        sql:field = "LastName"/>
      <xs:attribute name = "first-name"
        type = "xs:string"
        use = "required"
        sql:field = "FirstName"/>
      <xs:attribute name = "promotion"
        type = "xs:integer"
        use = "optional"
        sql:field = "EmailPromotion"/>
    </xs:complexType>
  </xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

```

The first thing to notice is the namespace declarations for the standard XML Schema `xs` namespace and the Microsoft mapping schema `sql` namespace.

```

<?xml version = "1.0"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
  xmlns:sql = "urn:schemas-microsoft-com:mapping-schema"
  elementFormDefault = "qualified"
  attributeFormDefault = "unqualified"
  xmlns:msdata = "urn:schemas-microsoft-com:mapping-schema">
  . . .
</xs:schema>

```

The annotation element contains an `appinfo` element that defines a relationship between the `Person.BusinessEntity` and `Person.Person` tables, which are related via their respective `BusinessEntityID` columns.

```

<xs:annotation>
  <xs:appinfo>
    <msdata:relationship name = "Contact-Person"
      parent = "Person.BusinessEntity"
      parent-key = "BusinessEntityID"
      child = "Person.Person"
      child-key = "BusinessEntityID"/>
    </xs:appinfo>
  </xs:annotation>

```

The outer element declaration uses the `sql:relation` attribute to map the `Contact XML` element to the `Person.Contact` table in the `AdventureWorks` database.

```
<xs:element name = "Contact"
  sql:relation = "Person.BusinessEntity">
</xs:element>
```

The Contact element consists of an anonymous complex type that is made up of two elements. The first element is the Id element, which identifies the BusinessEntityID being assigned to this person. An interesting feature is the sql:identity attribute, which is set to useValue on the BusinessEntityID column. This setting allows you to explicitly update an identity column via your XML source data.

```
<xs:complexType>
  <xs:sequence>
    <xs:element name = "Id"
      type = "xs:integer"
      sql:field = "BusinessEntityID"
      sql:identity = "useValue"/>
    . . .
  </xs:sequence>
</xs:complexType>
```

The second element is the Person element, also consisting of an anonymous complex type. The Person element maps its attribute columns of the Person.Person table. Notice the relationship between the Person.Person table and the Person.BusinessEntity tables is assigned to this element with the sql:relationship attribute. The complexType declaration contains the XML attribute to SQL column mappings. The sql:field attributes define the names of the target SQL Server column names.

```
<xs:element name = "Person"
  sql:relation = "Person.Person"
  sql:relationship = "Contact-Person">
  <xs:complexType>
    <xs:attribute name = "id"
      type = "xs:string"
      use = "optional"
      sql:field = "BusinessEntityID" />
    <xs:attribute name = "person-type"
      type = "xs:string"
      use = "required"
      sql:field = "PersonType"/>
    . . .
  </xs:complexType>
</xs:element>
</xs:sequence>
```

Note This example mapping schema uses a largely attribute-centric format—that is, most of the data is passed into the updategram via XML attributes. You can also use an element-centric format, or a combined format, if you choose.

The XSD annotated mapping schema format supports several annotations, all of which are detailed in Table 12-1.

Table 12-1. *XSD Mapping Schema Annotations*

Annotation	Description
<code>sql:encode</code>	Indicates that a URL should be returned instead of the value of a Binary Large Object (BLOB) field. Possible values are <code>url</code> or <code>default</code> .
<code>sql:field</code>	Maps an XML element or attribute to a database column. Set the value to the name of a SQL column.
<code>sql:guid</code>	Specifies whether an autogenerated GUID (Globally Unique Identifier) should be used in the column or an explicit value from the XML data. Can be set to <code>generate</code> or <code>useValue</code> .
<code>sql:hide</code>	Hides the element or attribute specified in the schema in the resulting XML. This option doesn't actually "hide" anything but rather renames the resulting element or attribute. Acceptable values are <code>0</code> , <code>1</code> , <code>true</code> , or <code>false</code> .
<code>sql:identity</code>	Specifies whether an autogenerated identity value should be used in the column or an explicit value from the XML data. Can be set to <code>ignore</code> or <code>useValue</code> .
<code>sql:inverse</code>	Tells the updategram to inverse its logical interpretation of the parent-child relationship specified in the <code>sql:relationship</code> element. Valid values are <code>0</code> , <code>1</code> , <code>true</code> , or <code>false</code> .
<code>sql:is-constant</code>	Creates a constant top-level or container element in the result XML that does not map to any table. Acceptable values are <code>0</code> , <code>1</code> , <code>true</code> , or <code>false</code> .
<code>sql:key-fields</code>	Specifies one or more columns that uniquely identify the rows in a table. Accepts a space-delimited list of column names.
<code>sql:limit-field/sql:limit-value</code>	Identifies a SQL column that contains a limiting value and a value to use in limiting the results. The <code>sql:limit-field</code> accepts a SQL column name, and the <code>sql:limit-value</code> is the value to use in limiting.
<code>sql:mapped</code>	Allows you to exclude schema items from the result. Valid values are <code>0</code> , <code>1</code> , <code>true</code> , or <code>false</code> .
<code>sql:max-depth</code>	Limits the depth of a hierarchical recursive relationship specified in the schema. Valid values are integers between <code>1</code> and <code>50</code> .
<code>sql:overflow-field</code>	Specifies a column to store unconsumed (overflow) data. This attribute should be set to a SQL column name.
<code>sql:prefix</code>	Specifies an attribute to be an ID, IDREF, or IDREFS type attribute. This should be set to the prefix you wish to assign to the ID.
<code>sql:relation</code>	Maps an XML node in the schema to a SQL table. The value should be set to the name of a SQL table.
<code>sql:relationship</code>	A <code>sql:relationship</code> element can be defined in the <code>appinfo</code> element of the XSD schema to define parent-child relationships between tables. The <code>sql:relationship</code> attribute specifies the <code>sql:relationship</code> element to use in defining relationships.
<code>sql:use-cdata</code>	Allows you to specify CDATA sections for certain elements in the XML document. Valid values are <code>0</code> , <code>1</code> , <code>true</code> , or <code>false</code> .

You can define three types of updategram documents with this mapping schema—an insert updategram, an update updategram, and a delete updategram. The insert updategram defines two namespaces: `sql:` points at Microsoft's `xml-sql` schema and `updg:` points at the `xml-updategram` schema. The insert updategram that uses this mapping schema is shown in Listing 12-3.

Listing 12-3. *Sample insert Updategram*

```
<?xml version = "1.0"?>
<ROOT xmlns:sql = "urn:schemas-microsoft-com:xml-sql"
  xmlns:updg = "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync mapping-schema = "c:\contactupdg.xsd">
    <updg:before>
    </updg:before>
    <updg:after>
      <Contact>
        <Id>21000</Id>
        <Person person-type = "IN"
          name-style = "O"
          last-name = "Braff"
          first-name = "Zach"
          promotion = "O" />
      </Contact>
    </updg:after>
  </updg:sync>
</ROOT>
```

The `ROOT` element encompasses the `updg:sync` element, which is the content of the updategram. The `updg:sync` element has a `mapping-schema` attribute that points at the annotated mapping schema. The `updg:sync` element contains an `updg:before` element that contains a snapshot of the “before” data and an `updg:after` element that contains an “after” snapshot of the data.

Inserts

For the insert updategram, the `updg:before` element is empty. In this example, the `updg:after` element contains a single `Contact` element, which will insert a new contact row into the `AdventureWorks Person.Person` table.

This insert updategram is the one used by the SQLXML Example application. To see it in action, click the Insert button to test the insert updategram functionality. The result is shown in Figure 12-2.

Figure 12-2. *Updategram insert row*

You can verify the new row was added with a `BusinessEntityID` of 21000 via a simple query in SQL Server Management Studio (SSMS), as shown in Figure 12-3.

Note `BusinessEntityID` 21000 is used because it's higher than the highest primary key value in the AdventureWorks database by default.

	BusinessEntity...	PersonTy...	NameSt...	Title	FirstNa...	MiddleNa...	LastNa...	Suffix	Email
1	21000	IN	0	NULL	Zach	NULL	Braff	NULL	0

Figure 12-3. *Verifying updategram insert*

Updates

After the updategram insert is complete, the Insert button name changes to Update. Clicking the Update button will execute an update via updategram. The update updategram is similar to the insert updategram, except that it contains data in both the `updg:before` and `updg:after` elements, as shown in Listing 12-4.

Listing 12-4. *Sample update Updategram*

```
<?xml version = "1.0"?>
<ROOT xmlns:sql = "urn:schemas-microsoft-com:xml-sql"
  xmlns:updg = "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync mapping-schema = "c:\contactupdg.xsd">
    <updg:before>
      <Contact>
        <Id>21000</Id>
        <Person/>
      </Contact>
    </updg:before>
    <updg:after>
      <Contact>
        <Id>21000</Id>
        <Person last-name="Braff"
          first-name="Zachary" />
      </Contact>
    </updg:after>
  </updg:sync>
</ROOT>
```

The `updg:before` element contains a `Contact` element with an `Id` element set to 21000. This indicates that the row in the `Person.Person` table that I previously inserted, with `BusinessEntityID` 21000, will be updated. The `updg:after` element contains a `Contact` element with the “after” data, in this case I am performing a simple change—the contact’s first name will be updated from “Zach” to “Zachary.” After pressing the Update button, the screen shows the updategram that is sent to the server. The result is shown in Figure 12-4.

Figure 12-4. *Updategram update row*

As before, you can verify the results in SSMS with a simple SELECT query, as shown in Figure 12-5.

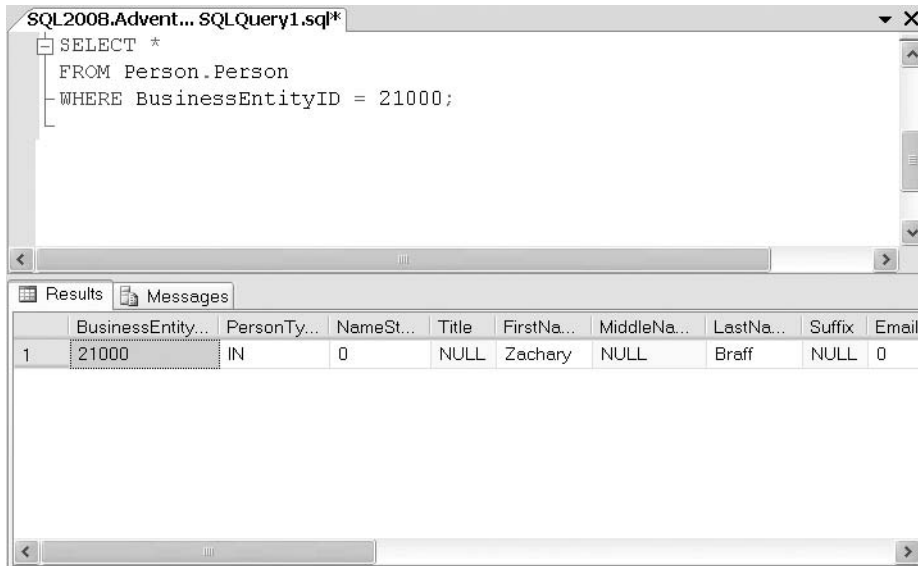


Figure 12-5. *Verifying updategram update*

Deletes

Updategrams also support operations to delete rows from tables. The delete updategram is similar in form to the previous updategrams. The difference is that the `updg:before` element contains Contact elements you wish to delete, while the `updg:after` element does not contain any corresponding elements. Listing 12-5 shows the delete updategram used in the SQLXML Example application.

Listing 12-5. *Sample delete Updategram*

```
<?xml version = "1.0"?>
<ROOT xmlns:sql = "urn:schemas-microsoft-com:xml-sql"
  xmlns:updg = "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync mapping-schema = "c:\contactupdg.xsd">
    <updg:before>
      <Contact>
        <Id>21000</Id>
        <Person last-name = "Braff"
          first-name = "Zachary" />
      </Contact>
    </updg:before>
    <updg:after>
    </updg:after>
  </updg:sync>
</ROOT>
```


After the update operation in the SQLXML Example completes, the Update button changes to Delete. Clicking the Delete button removes the previously added row with ContactID 21000 from the Person.Person table. The result is shown in Figure 12-6.

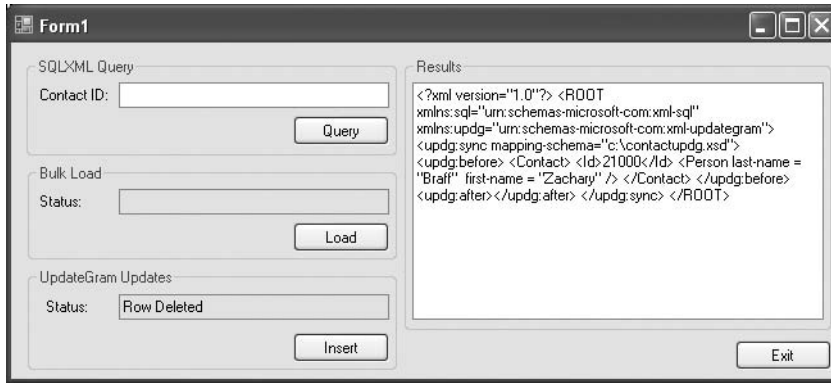


Figure 12-6. *Updategram delete row*

You can again verify that the row has been deleted in SSMS with a simple query, as shown in Figure 12-7.

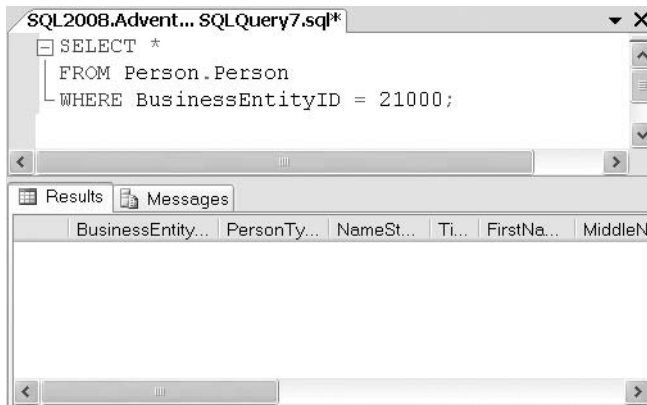


Figure 12-7. *Verifying updategram delete*

Executing Updategrams with SqlXmlCommand

In the SQLXML Example application, I used the `SqlXmlCommand` class to execute the sample updategrams against the database. The sample code uses the .NET managed `SqlXmlCommand` class to execute the updategrams, as shown in Listing 12-6. Note that some portions of the code have been abbreviated to better focus on the `SqlXmlCommand` class and its methods.

Listing 12-6. *SqlXmlCommand Updategram Sample Code*

```
// The SqlXmlConnString is the SQLXML connection string
string SqlXmlConnString = "Provider=SQLOLEDB;" +
    "Server=(local);" +
    "database=AdventureWorks;" +
    "Integrated Security=SSPI";

// The Updategrams hashtable contains all three sample updategrams,
// the sample Insert updategram, sample Update updategram, and sample
// Delete updategram.
Hashtable Updategrams = new Hashtable(3);

. . .

// The sample updategrams are added to the hashtable below
Updategrams.Add("Insert", "<?xml version='1.0'> " +
    "<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql' " +
    "xmlns:updg='urn:schemas-microsoft-com:xml-updategram'> " +
    . . .
    "</ROOT>");

Updategrams.Add("Update", "<?xml version='1.0'> " +
    "<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql' " +
    "xmlns:updg='urn:schemas-microsoft-com:xml-updategram'> " +
    . . .
    "</ROOT>");

Updategrams.Add("Delete", "<?xml version='1.0'> " +
    "<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql' " +
    "xmlns:updg='urn:schemas-microsoft-com:xml-updategram'> " +
    . . .
    "</ROOT>");

. . .

SqlXmlCommand cmd = new SqlXmlCommand(SqlXmlConnString);
cmd.CommandType = SqlXmlCommandType.UpdateGram;
cmd.RootTag = "ROOT";

cmd.CommandText = Updategrams[btnInsertUpdateDelete.Text].ToString();
txtResult.Text = Updategrams[btnInsertUpdateDelete.Text].ToString();
cmd.ExecuteNonQuery();
```

After some initial setup, including defining the SQLXML connection string and the sample updategrams, the code creates a `SqlXmlCommand` object and sets the `CommandType` to `SqlXmlCommandType.UpdateGram`. The `RootTag` attribute is set to `ROOT` to indicate the root element of the updategrams.

```
SqlXmlCommand cmd = new SqlXmlCommand(SqlXmlConnString);
cmd.CommandType = SqlXmlCommandType.UpdateGram;
cmd.RootTag = "ROOT";
```

Next the `CommandText` attribute is set to the actual updategram to be executed. This depends on the current button caption. Each sample updategram is also displayed in a text box so you can see what's being executed. Finally, the `ExecuteNonQuery` method is called on the `SqlXmlCommand` to send each updategram to the server for execution.

```
cmd.CommandText = Updategrams[btnInsertUpdateDelete.Text].ToString();
txtResult.Text = Updategrams[btnInsertUpdateDelete.Text].ToString();
cmd.ExecuteNonQuery();
```

SQLXMLCOMMAND AND OLEDB/COM

The `SqlXmlCommand` class is a wrapper around the SQLXML OLEDB/COM interface. As such, accessing SQLXML through the .NET class can incur a performance penalty every time it instantiates the necessary COM (Component Object Model) and OLEDB objects. SQLXML updategrams are useful for implementing middle-tier applications or web-based database updates from remote systems that don't necessarily have the ability to create direct connections to your SQL Server. JavaScript applications and custom-built non-.NET disconnected data management controls come to mind. Keep in mind, however, that for single updates you can get better performance by avoiding OLEDB/COM in favor of other tools, like SQL Server's native HTTP Simple Object Access Protocol (SOAP) endpoints.

Using `SqlXmlCommand` from .NET requires you to add a reference to the `Microsoft.Data.SqlXml` namespace, as shown in Figure 12-8.

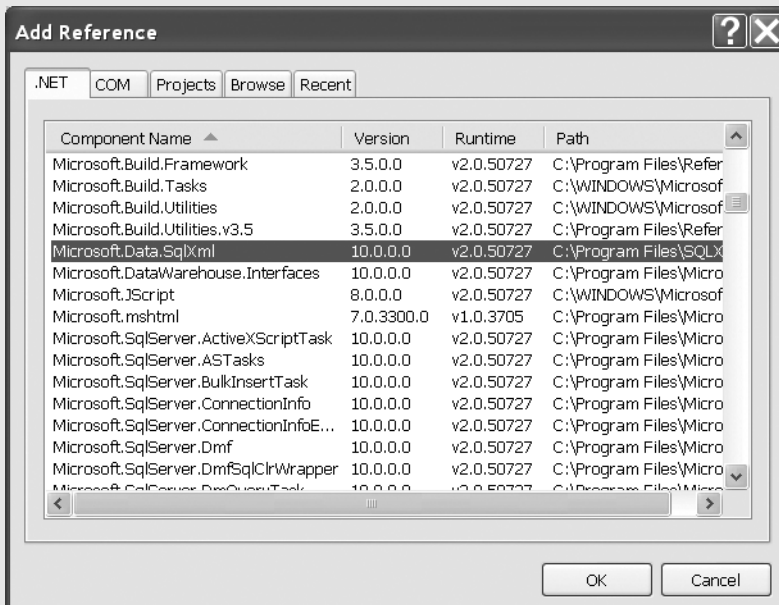


Figure 12-8. Adding `Microsoft.Data.SqlXml` reference

Diffgrams

Diffgrams provide another XML format for performing database updates. Diffgrams were introduced as a specialized format to support .NET disconnected data sets. Diffgrams have an updated format, but they perform a function similar to updategrams. While updategrams support both XDR and XSD schemas, diffgrams support only XSD schemas. Possibly because they are not as specialized as diffgrams, have been around longer, and have undergone several iterations of improvement over the years, updategrams are considerably more flexible than diffgrams. Because of their more specialized purpose, you will normally encounter diffgrams when “peeking under the hood” of .NET disconnected data sets.

Bulk Loading

A useful feature of SQLXML is the Bulk Load feature. This feature is implemented by a stand-alone COM object, which is implemented separately from the other SQLXML functionality. Accessing this functionality from .NET requires you to add a reference to the Microsoft SQLXML BulkLoad 4.0 Type Library COM object, as shown in Figure 12-9.

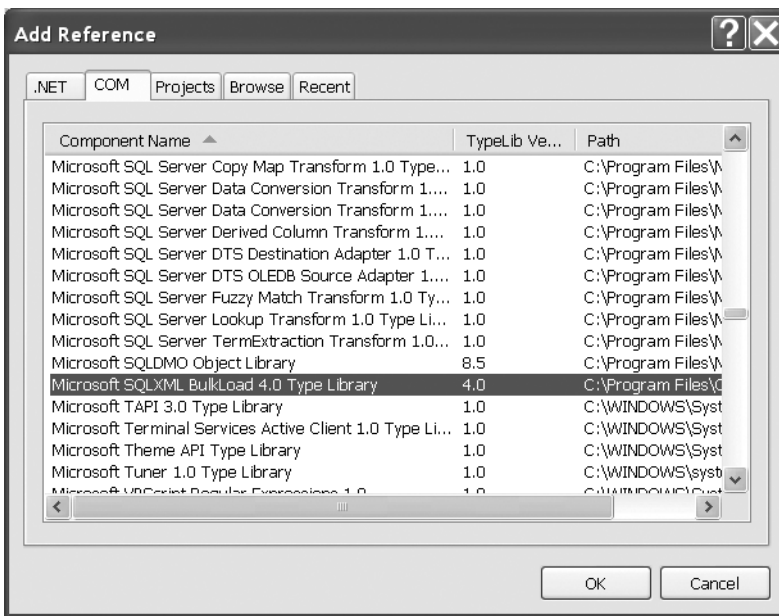


Figure 12-9. Adding a reference to the SQLXML COM object

The COM object reference creates a .NET wrapper called `SQLXMLBULKLOADLib`. The `SQLXMLBulkLoad4Class` exposes the SQLXML Bulk Load functionality. The SQLXML Example application contains a sample of SQLXML Bulk Load in action. When you click the Load button, as shown in Figure 12-10, the example application creates a simple table in the AdventureWorks database and bulk loads XML-formatted geolocation data for business customers into the table.

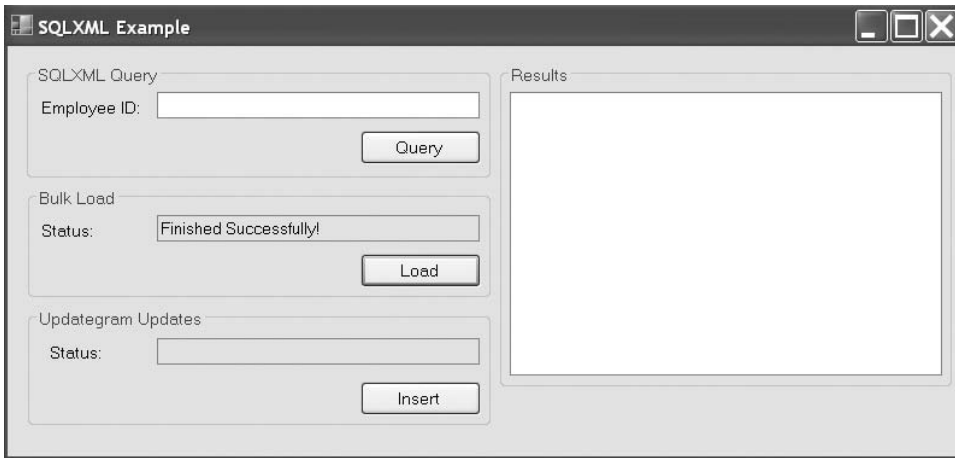


Figure 12-10. *SQLXML Bulk Load*

The `SQLXMLBulkLoad4Class` requires two XML files. One is an annotated mapping schema document with the XML-to-relational mappings, and the second is the XML file containing the actual XML-formatted data to import. The `AWCustGeo.xsd` document is the annotated mapping schema document, shown in Listing 12-7.

Listing 12-7. *SQLXML Bulk Load Mapping Schema*

```
<?xml version = "1.0"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  xmlns:sql = "urn:schemas-microsoft-com:mapping-schema">

  <xsd:element name = "CustomerCoordinates"
    sql:relation = "Sales.CustomerGeography">

    <xsd:complexType>

      <xsd:sequence>

        <xsd:element name = "Resolution"
          sql:field = "Resolution"
          type = "xsd:string" />

        <xsd:element name = "Latitude"
          sql:field = "Latitude"
          type = "xsd:decimal" />

        <xsd:element name = "Longitude"
          sql:field = "Longitude"
          type = "xsd:decimal" />
```

```

    </xsd:sequence>

    <xsd:attribute name = "CustomerID"
        sql:field = "CustomerID"
        type = "xsd:integer" />

</xsd:complexType>

</xsd:element>

</xsd:schema>

```

The table that contains the customer geolocation data is named `Sales.CustomerGeography`. This table is dropped and recreated before the load begins using a `CREATE TABLE` statement, like the one shown in Listing 12-8.

Listing 12-8. *Sales.CustomerGeography Table*

```

CREATE TABLE Sales.CustomerGeography (
    CustomerID INT NOT NULL PRIMARY KEY,
    Latitude NUMERIC (15, 6),
    Longitude NUMERIC (15, 6),
    Resolution VARCHAR(20)
);

```

A sample of the source XML data in the `AWCustGeo.xml` file is shown in Listing 12-9.

Listing 12-9. *SQLXML Bulk Load Source Data*

```

<?xml version = "1.0"?>
<ROOT xmlns:sql = "urn:schemas-microsoft-com:xml-sql">
    <CustomerCoordinates CustomerID = "1">
        <Resolution>best</Resolution>
        <Latitude>47.611744</Latitude>
        <Longitude>-122.347698</Longitude>
    </CustomerCoordinates>
    <CustomerCoordinates CustomerID = "2">
        <Resolution>best</Resolution>
        <Latitude>47.475590</Latitude>
        <Longitude>-122.204569</Longitude>
    </CustomerCoordinates>
    <CustomerCoordinates CustomerID = "3">
        <Resolution>best</Resolution>
        <Latitude>32.935160</Latitude>
        <Longitude>-97.017039</Longitude>
    </CustomerCoordinates>
    <CustomerCoordinates CustomerID = "4">
        <Resolution>good</Resolution>
        <Latitude>30.302679</Latitude>
    </CustomerCoordinates>

```

```

    <Longitude>-97.761980</Longitude>
  </CustomerCoordinates>
  . . .
</ROOT>

```

Each CustomerCoordinates element contains a CustomerID, a geocoding Resolution element, and Latitude and Longitude coordinate pairs. The C# procedure that performs the actual bulk load, named BulkLoadData, is shown in Listing 12-10.

Listing 12-10. *Bulk Load Procedure*

```

[STAThread()]
private void BulkLoadData()
{
    SQLXMLBULKLOADLib.SQLXMLBulkLoad4Class bulkloader =
        new SQLXMLBULKLOADLib.SQLXMLBulkLoad4Class();
    bulkloader.ConnectionString = SqlXmlConnString;
    bulkloader.ErrorLogFile = "c:\\bulkloadererrors.xml";
    bulkloader.KeepIdentity = false;
    bulkloader.Transaction = false;
    bulkloader.Execute("c:\\AWCustGeo.xsd", "c:\\AWCustGeo.xml");
}

```

The first thing to notice is the [STAThread()] attribute on the procedure. The SQLXML Bulk Load COM object must be executed in single thread mode, or it will throw an exception. The first step inside the procedure is the creation of an instance of the SQLXMLBulkLoad4Class. Then the procedure sets the ConnectionString attribute to point at the AdventureWorks database.

The ErrorLogFile attribute sets the location for error messages to be output in XML format. The KeepIdentity and Transaction attributes are also both set to false to turn off their respective Bulk Load options. Finally, the Execute method accepts the file names of an annotated schema and a source XML data document.

You can verify the results of the SQLXML Bulk Load process by selecting from the newly created Sales.CustomerGeography table in the AdventureWorks database, as shown in Figure 12-11.

	CustomerID	Latitude	Longitude	Resolution	
1	1	47.811744	-122.347698	best	;
2	2	47.475590	-122.204569	best	;
3	3	32.935160	-97.017039	best	;
4	4	30.302679	-97.761980	good	;
5	5	37.484880	-121.928669	best	;
6	6	34.229270	-118.995709	best	;
7	7	40.759525	-111.888219	medium	;
8	8	25.791350	-80.319219	best	;

Figure 12-11. *Result of SQLXML Bulk Load*

The SQLXML Bulk Load feature is a handy utility for loading XML data into your relational database fairly quickly. Because it's COM-based, however, you may find it to be less efficient than a native .NET solution. It is useful if you don't want to spend time developing your own custom solution.

Querying SQLXML with XPath

SQLXML also provides the capability to query relational data via XPath queries. This functionality is provided by using annotated schemas to expose an XML view of your relational data. Listing 12-11 is an annotated schema representing the hierarchical relationship between the `Sales.Customer` and `Person.Person` tables, and between the `Sales.Customer` and `Sales.SalesOrderHeader` table.

Listing 12-11. *Hierarchical Annotated Schema*

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  xmlns:msdata = "urn:schemas-microsoft-com:mapping-schema">
  <xsd:annotation>
    <xsd:appinfo>
      <msdata:relationship name = "Customer-Person"
        parent = "Sales.Customer"
        parent-key = "PersonID"
        child = "Person.Person"
        child-key = "BusinessEntityID"/>
      <msdata:relationship name = "Customer-SalesOrderHeader"
        parent = "Sales.Customer"
        parent-key = "CustomerID"
        child = "Sales.SalesOrderHeader"
        child-key = "CustomerID"/>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:element name = "Customer"
    msdata:relation = "Sales.Customer"
    msdata:key-fields = "CustomerID"
    type = "CustomerType"/>
  <xsd:complexType name = "CustomerType">
    <xsd:sequence>
      <xsd:element name = "Person"
        msdata:relation = "Person.Person"
        msdata:key-fields = "BusinessEntityID"
        msdata:relationship = "Customer-Person">
        <xsd:complexType>
          <xsd:attribute name = "LastName" type = "xsd:string"/>
          <xsd:attribute name = "MiddleName" type = "xsd:string"/>
          <xsd:attribute name = "FirstName" type = "xsd:string"/>
        </xsd:complexType>
      </xsd:element>
```



```

<xsd:element name = "SalesOrderHeader"
  msdata:relation = "Sales.SalesOrderHeader"
  msdata:key-fields = "SalesOrderID"
  msdata:relationship = "Customer-SalesOrderHeader">
  <xsd:complexType>
    <xsd:attribute name = "SalesOrderID" type = "xsd:integer"/>
    <xsd:attribute name = "SubTotal" type = "xsd:decimal"/>
    <xsd:attribute name = "TotalDue" type = "xsd:decimal"/>
    <xsd:attribute name = "OrderDate" type = "xsd:date"/>
  </xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:attribute name = "CustomerID" type = "xsd:integer"/>
</xsd:complexType>
</xsd:schema>

```

SQLXML XPATH LIMITATIONS

SQLXML supports a subset of XPath queries. Following is a list of limitations imposed on the SQLXML XPath implementation:

- The only axis specifiers supported are `child`, `parent`, `attribute`, and `self`.
- Only element and attribute node types are supported.
- Only predicates that return a boolean result are supported. Numeric predicates like `[1]` are not supported.
- Only the three XPath data types are supported: string, number, and boolean.
- SQLXML 4 does not support the root query (specified by a forward slash `/`), or descendant-or-self queries (specified by double slashes `//`). Every query must begin at a top-level schema element type.
- SQLXML does not support queries that generate a Cartesian product.
- The `mod` and `union (|)` operators, string functions, and numeric functions are not supported.

Despite these limitations, you can still create useful XPath expressions to query your relational data, as you will see in the code sample for this section.

The code in Listing 12-12 performs an XPath query against the annotated schema in Listing 12-11. The sample program is designed to allow you to enter your own XPath query on a form and execute the query against the XML view of the AdventureWorks database.

Listing 12-12. SQLXML XPath Query Sample

```

using System;
using System.Windows.Forms;
using Microsoft.Data.SqlXml;
using System.IO;

```

```

namespace SQLXML_Path
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnQuery_Click(object sender, EventArgs e)
        {
            SqlXmlCommand sqlcom = new
                SqlXmlCommand("PROVIDER=SQLOLEDB;SERVER=SQL2008;" +
                    "INITIAL CATALOG=AdventureWorks;INTEGRATED SECURITY=SSPI;");
            sqlcom.CommandText = txtQuery.Text;
            sqlcom.CommandType = SqlXmlCommandType.XPath;
            sqlcom.SchemaPath = "c:\\SQLXML-Path.xml";
            Stream s = sqlcom.ExecuteStream();
            StreamReader sr = new StreamReader(s);
            txtResult.Text = sr.ReadToEnd();
            sr.Dispose();
        }
    }
}

```

As in the previous examples in this chapter, the `SqlXmlCommand` class is the basis of the SQLXML sample. You generate a new instance by passing its constructor function a `SQLOLEDB` connection string. Then you set the `CommandText` to the XPath query, the `CommandType` to `XPath`, and the `SchemaPath` to the path where the XML mapping document is located.

```

SqlXmlCommand sqlcom = new
    SqlXmlCommand("PROVIDER=SQLOLEDB;SERVER=SQL2008;" +
        "INITIAL CATALOG=AdventureWorks;INTEGRATED SECURITY=SSPI;");
sqlcom.CommandText = txtQuery.Text;
sqlcom.CommandType = SqlXmlCommandType.XPath;
sqlcom.SchemaPath = "c:\\SQLXML-Path.xml";

```

Then you execute the XPath query with the `ExecuteStream` method of the `SqlXmlCommand` class and use a `StreamReader` to retrieve the results.

```

Stream s = sqlcom.ExecuteStream();
StreamReader sr = new StreamReader(s);
txtResult.Text = sr.ReadToEnd();
sr.Dispose();

```

In the sample application, I've used a simple XPath query to return all `Customer` elements that contain a `Person` element with their `LastName` attribute equal to "Adina". The actual XPath query looks like the following:

```
/Customer[Person/@LastName = "Adina"]
```

The results are returned in XML format, as shown in Figure 12-12.

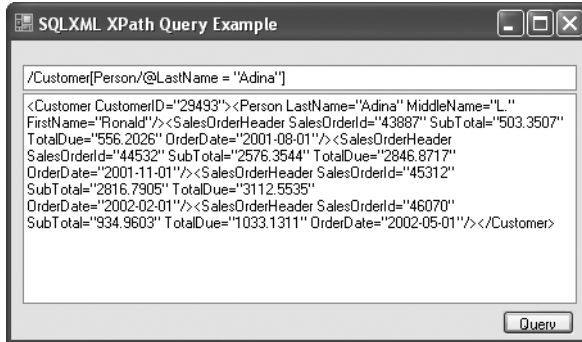


Figure 12-12. Result of sample SQLXML XPath query example

The SQLXML XPath query functionality can be a useful tool for remote querying of relational data, although SQL Server's xml data type and XQuery functionality provide a powerful alternative to this particular SQLXML functionality.

Summary

SQLXML has been available since SQL Server 2000. SQLXML functionality is still useful for specialized requirements like remote web-based SQL Server data manipulations, client-side XML formatting of relational query results, and bulk loading of XML data into SQL Server. SQLXML is OLEDB/COM-based, making it less efficient than other solutions (like a native .NET solution) might be, but the fact that it already exists can help you get a jump on specialized XML-based relational data querying and manipulation.

In this chapter, I discussed the main tools available through SQLXML, including querying relational data in XML format via SQLXML, manipulating data via updategrams, and bulk loading XML formatted data to relational format. Much of SQLXML's additional functionality has been supplanted by newer functionality introduced and expanded since the release of SQL Server 2005.

In the next chapter, I will discuss .NET's exciting new Language Integrated Query (LINQ) functionality. Specifically I'll consider the new LINQ to XML functionality that allows you to perform declarative SQL-like queries against XML data directly in your C# or VB code.



LINQ to XML

Language-Integrated Query (LINQ) is an exciting new .NET Framework feature that provides a standard set of operators to modify, query, and manipulate data without regard to the source. LINQ to XML, or XLinQ, provides a variety of XML-based query and manipulation tools that are built right into the C# and Visual Basic (VB) languages. LINQ to XML has a fast, lightweight, in-memory XML Application Programming Interface (API). In this chapter, I'll discuss some of the features of LINQ to XML.

Note The code in this chapter takes advantage of C# language extensions that are not available in Visual Studio 2005, but they are available in Visual Studio 2008.

Functional Construction

One of the features that LINQ to XML provides is *functional construction*. LINQ to XML supports building your trees in a bottom-up XML (Document Object Model) DOM-like manner, but it also provides support for a new top-down functional construction approach. Functional construction allows you to populate an entire XElement with a single statement. Using functional construction, you can easily create XML documents in memory using a coding style that more accurately reflects the structure and content of the resultant XML document. Listing 13-1 demonstrates this using functional construction to generate XML.

Listing 13-1. Creating XML Using Functional Construction

```
using System;
using System.Xml.Linq;

namespace Apress.Samples
{
    class Linq_FuncConst_Example
    {
        static void Main(string[] args)
        {
            XElement market_summary = new XElement("market-summary",
                new XElement("index",
```

```

        new XAttribute("symbol", "DJIA"),
        new XAttribute("name", "Dow Jones Industrial Average"),
        new XElement("daily-summary",
            new XElement("date", "2008-01-11T05:00:00Z"),
            new XElement("open", "12850.74"),
            new XElement("high", "12863.34"),
            new XElement("low", "12495.91"),
            new XElement("close", "12606.30"),
            new XElement("volume", "4495840000"),
            new XElement("adjusted-close", "12606.30")
        )
    );
    Console.WriteLine(market_summary.ToString());
}
}
}

```

The `XElement` and `XAttribute` classes represent XML elements and attributes, respectively. In fact, LINQ to XML provides several classes that map to XML nodes and other constructs necessary to build, query, and manipulate XML data. Figure 13-1 shows the object hierarchy of the major LINQ to XML classes provided in the `System.Xml.Linq` namespace. You must add your own reference to this namespace in your code, as I did in Listing 13-1, in order to use these classes.

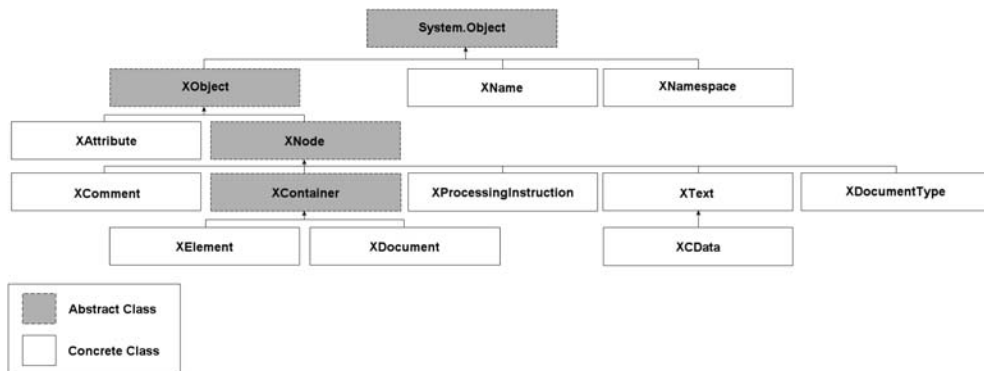


Figure 13-1. Main LINQ to XML classes object hierarchy

The sample uses the `XElement` and `XAttribute` class constructors to generate the XML result shown in Figure 13-2. Notice how closely the source code for the functional construction resembles the resulting XML.

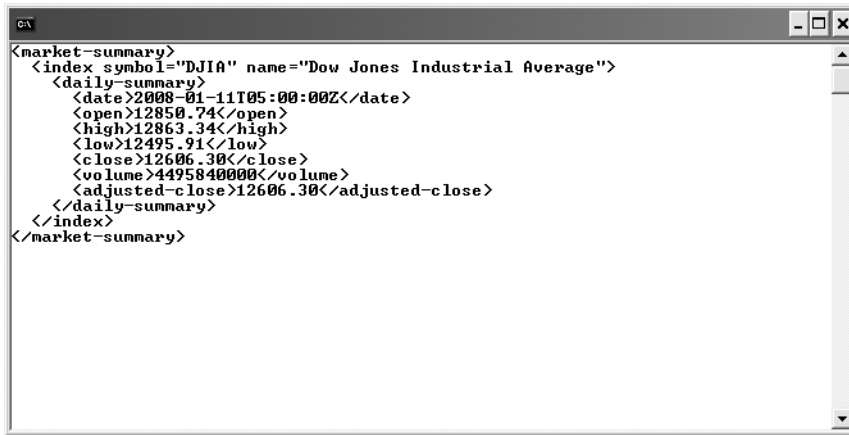


Figure 13-2. XML functional construction result

Loading XML from Other Sources

Functional construction is useful for creating XML instances efficiently within code, and it is also useful for applying transformations, as you will see later. However, it's very common to retrieve XML from outside sources, so LINQ to XML provides static methods to populate `XElement` and `XDocument` instances from files, strings, and `XmlReader` instances. This provides opportunities to populate LINQ to XML instances from any available source, like a SQL Server.

Loading XML with the `XmlReader`

The `XmlReader` is the standard method for loading just about any non-filesystem data, including XML data retrieved from a local area network, the Web, SQL Server, or any type of text stream. Listing 13-2 is a simple demonstration of populating an `XElement` with a SQL Server source query and the `ExecuteXmlReader` method of the `SqlCommand` object.

Listing 13-2. Using `ExecuteXmlReader` to Populate an `XElement`

```
using System;
using System.Xml.Linq;
using System.Data.SqlClient;
using System.Xml;

namespace Apress.Samples
{
    class Linq_Db_Xml_Example
    {
        static void Main(string[] args)
        {
            XElement xml;
            string constr = "SERVER=SQL2008;INITIAL CATALOG=AdventureWorks;" +
```

```

        "INTEGRATED SECURITY=SSPI;";
        using (SqlConnection sqlcon = new SqlConnection(constr))
        {
            sqlcon.Open();
            using (SqlCommand sqlcmd = new SqlCommand("SELECT Resume " +
                "FROM HumanResources.JobCandidate " +
                "WHERE JobCandidateID = 1;", sqlcon))
            {
                using (XmlReader xr = sqlcmd.ExecuteXmlReader())
                {
                    xml = XElement.Load(xr,
                        LoadOptions.PreserveWhitespace);
                }
            }
        }
        Console.WriteLine(xml.ToString());
    }
}
}

```

The example in Listing 13-2 queries the AdventureWorks sample database, retrieving the XML resume for a job candidate. The XML resume is returned as an `XmlReader`, which is then used to populate the `XElement` instance. The result is shown in Figure 13-3.

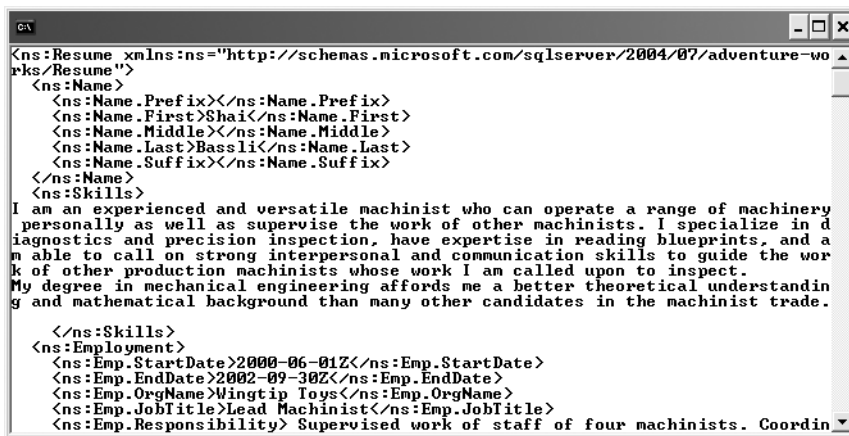


Figure 13-3. Loading an `XElement` instance via the `ExecuteXmlReader` method

Querying with LINQ to SQL

Alternatively, you can go with an all-LINQ solution to perform the same task. With this solution, I will use LINQ to SQL to query the SQL Server database and populate the LINQ to XML XElement with the job candidate resume returned. The first step is to use the LINQ to SQL Object/Relational (O/R) designer. The designer generates C# classes that represent your database objects, quickly and easily. To use the O/R designer, you need to add a new LINQ to SQL Classes item to your project, as shown in Figure 13-4.

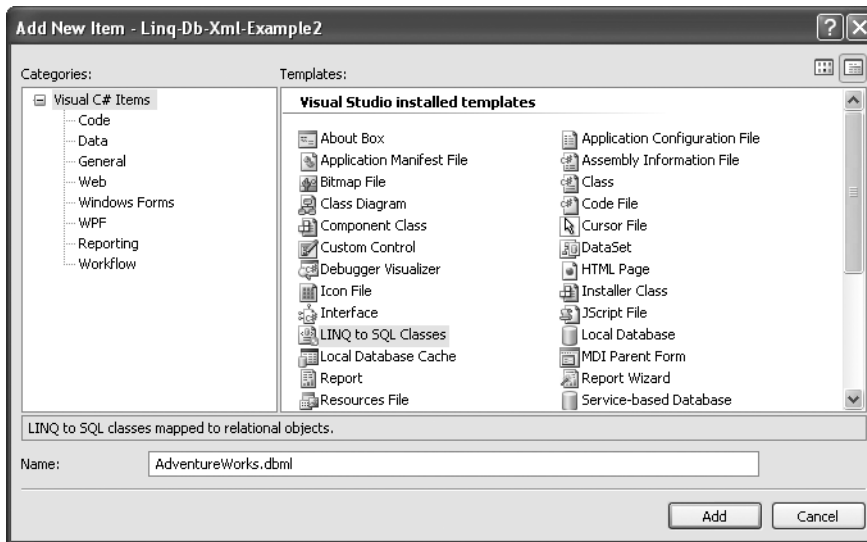


Figure 13-4. Adding the LINQ to SQL Classes item to your project

Tip When you add the LINQ to SQL Classes item, the file name you choose is important because the name of the data context class generated will use the name. In the example, I chose to call it `AdventureWorks.dbml`, so the `DataContext` class generated is `AdventureWorksDataContext`.

Once you've added the LINQ to SQL Classes item, you need to add a data connection to your project. Specifically, you need to add a data connection that uses the .NET Framework Data Provider for SQL Server. Once you've added the data connection, it will show up in the Server Explorer window, as shown in Figure 13-5.

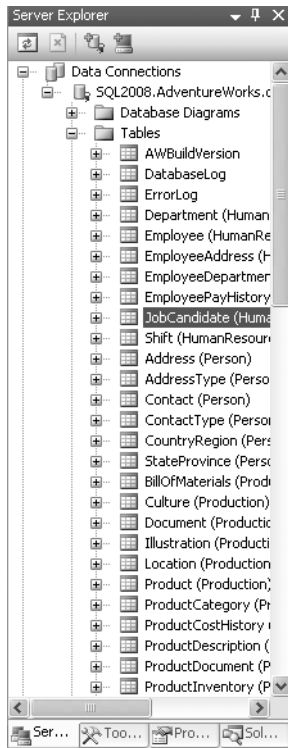


Figure 13-5. *Viewing the data connections in the Server Explorer window*

Note At the time of this writing, LINQ to SQL did not support SQL Server 2008 connectivity, so this example was created with SQL Server 2005. LINQ to SQL support for SQL Server 2008 is scheduled to be added with the release of Visual Studio 2008 Service Pack 1. The estimated date is sometime in the second quarter of 2008.

After you have your LINQ to SQL Classes designer and a data connection added to your project, simply drag and drop tables from the Server Explorer window onto the designer surface, as shown in Figure 13-6.



Figure 13-6. *Dropping a table onto the LINQ to SQL Classes designer surface*

The designer automatically generates the appropriate classes to map your tables to C# objects. The main class in this instance is called `AdventureWorksDataContext`. This class manages the data context, including connection, transaction, and other relevant information.

Once you have LINQ to SQL set up, it's time to start coding. Listing 13-3 creates an instance of the `AdventureWorksDataContext` and performs a LINQ query against the `AdventureWorks.HumanResources.JobCandidate` table. Then you iterate the result set (it will always contain exactly one row), assign the contents of the xml data type `Resume` column to an `XElement`, and display the results on the console.

Listing 13-3. *Using LINQ to SQL to Populate a LINQ to XML XElement*

```
using System;
using System.IO;
using System.Xml;
using System.Linq;
using System.Xml.Linq;
using System.Data.SqlClient;

namespace Apress.Samples
{
    class Linq_Db_Xml_Example2
    {
        static void Main(string[] args)
        {
            AdventureWorksDataContext db =
                new AdventureWorksDataContext();

            db.Log = Console.Out;
```

```

var JobCandidates = from JobCandidate in db.JobCandidates
                    where JobCandidate.JobCandidateID == 1
                    select JobCandidate;

foreach (var JobCandidate in JobCandidates)
{
    XElement xml = JobCandidate.Resume;
    Console.WriteLine(xml.ToString());
}
}
}
}

```

You might notice how much cleaner and easier to understand this code is than the previous version where I used the `ExecuteXmlReader` method of the `SqlCommand`. This object-oriented approach to relational database access works well for this particular task. Notice also that I've added a line setting `db.Log` equal to `Console.Out`. This tells LINQ to SQL to output the actual parameterized SQL `SELECT` query it generates based on my LINQ query. The output of Listing 13-3 is shown in Figure 13-7.

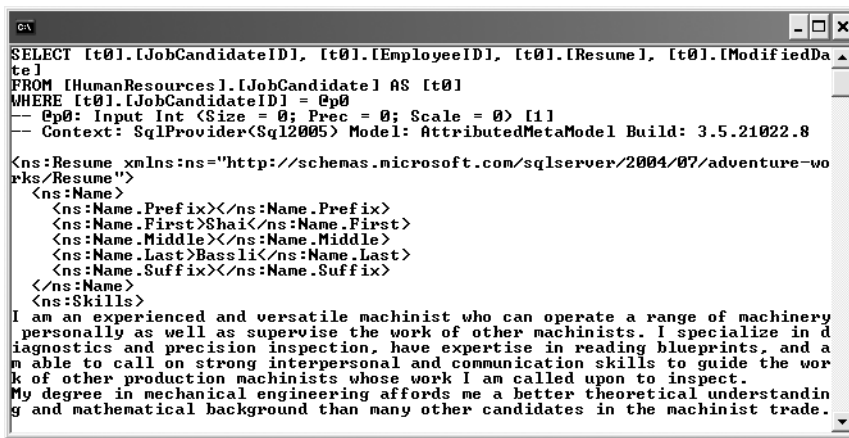


Figure 13-7. Results of using LINQ to SQL to populate an `XElement`

Loading XML from the File System

Although you can use the `XmlReader` to populate `XElement` and `XDocument` instances from files stored on the local file system, both classes offer the static `Load` method as a convenience for this very common requirement. Just pass the `Load` method a file name and optional `LoadOptions`. Listing 13-4 demonstrates the `XDocument Load` method in action.

Listing 13-4. Loading an `XDocument` from a File

```

using System;
using System.Xml.Linq;

```

```

namespace Apress.Samples
{
    class Linq_Load_Example
    {
        static void Main(string[] args)
        {
            XDocument xd = XDocument.Load("c:\\market-summary.xml",
                LoadOptions.PreserveWhitespace);
            Console.WriteLine(xd.ToString());
        }
    }
}

```

The example loads the `market-summary.xml` file into an `XDocument` instance, preserving all white space (even insignificant white space) because of the `LoadOptions.PreserveWhitespace` option. The result is written to the console, as shown in Figure 13-8.



Figure 13-8. Result of populating an `XDocument` with an XML file

Loading XML from a String

As another convenience method, you can also populate `XElement` and `XDocument` objects with string objects via the static `Parse` method. The `Parse` method accepts a string containing XML content and an optional `LoadOptions` argument. This can be useful if you have XML content in strings and want to query or manipulate it with LINQ to XML, or if you want to convert an `XmlDocument` to an `XElement` or `XDocument`. Listing 13-5 shows how to load an `XDocument` from a string.

Listing 13-5. Loading an `XDocument` from a string

```

using System;
using System.Xml.Linq;

```

```

namespace Apress.Samples
{
    class Linq_Load_String_Example
    {
        static void Main(string[] args)
        {
            XDocument xd = XDocument.Parse("<ms:market-summary " +
                "xmlns:ms=\"apress:sample:market-summary:urn\">" +
                "<ms:index symbol=\"DJIA\" " +
                "name=\"Dow Jones Industrial Average\">" +
                "<daily-summary>" +
                "<date>2008-01-11T05:00:00Z</date>" +
                "<open>12850.74</open>" +
                "<high>12863.34</high>" +
                "<low>12495.91</low>" +
                "<close>12606.3</close>" +
                "<volume>4495840000</volume>" +
                "<adjusted-close>12606.3</adjusted-close>" +
                "</daily-summary>" +
                "</ms:index>" +
                "</ms:market-summary>");

            Console.WriteLine(xd.ToString());
        }
    }
}

```

Figure 13-9 shows the results of populating the XDocument with XML data stored in a string.



Figure 13-9. Result of populating an XDocument with a string

Loading XML via HTTP

You can use the Load method to retrieve files over the network via HTTP. This example revisits a previous example from Chapter 10. In that example, I used SQLCLR to retrieve an RSS feed from the Web. In Listing 13-6, I perform a similar task with LINQ to XML. In this example, I retrieve the same RSS feed from the SQL Server Central web site, query it via LINQ to XML, and display the results on screen.

Listing 13-6. *Loading an XDocument from the Web*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace Apress.Samples
{
    class Linq_Rss_Xml_Example
    {
        static void Main(string[] args)
        {
            XDocument rss = XDocument.Load(
@"http://www.sqlservercentral.com/Xml/Rss/Articles/SQL+Server+2008");

            var rssfeed = from item in rss.Elements("rss")
                           .Elements("channel")
                           .Elements("item")
                           select new {
                               title = item.Element("title").Value,
                               description = item.Element("description").Value,
                               guid = item.Element("guid").Value,
                               pubDate = item.Element("pubDate").Value,
                               link = item.Element("link").Value
                           };

            foreach (var item in rssfeed)
            {
                Console.WriteLine("title: {0}\nguid: {1}\npub. date: {2}" +
                                   "\nlink: {3}\ndescription: {4}",
                                   item.title,
                                   item.guid,
                                   item.pubDate,
                                   item.link,
                                   item.description);
                Console.WriteLine("=====");
            }
        }
    }
}
```

```

    }
}

```

The key to this example is passing the URL to the network or web-based file in the Load method. The file is retrieved and queried—I'll discuss the querying aspect further in the "Querying XML" section of this chapter. Finally the results are iterated and displayed on the console. The result is shown in Figure 13-10.

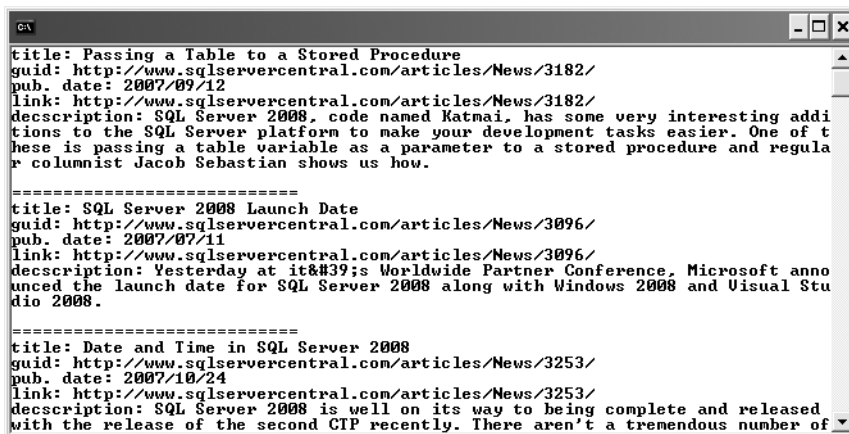


Figure 13-10. Result of XQuery querying an RSS feed

Querying XML

One of the powerful features of LINQ is the ability to perform SQL-like declarative queries against any supported data source. I performed a simple LINQ query against a SQL Server data source in Listing 13-3. The query used in that example is shown here in Listing 13-7.

Listing 13-7. Simple LINQ Query Used in Listing 13-3

```

var JobCandidates = from JobCandidate in db.JobCandidates
                    where JobCandidate.JobCandidateID == 1
                    select JobCandidate;

```

SQL developers will recognize some of the keywords used and should have a general idea of how the query works. Let's look at the composition of this query for a moment:

- In the first line, you'll notice that the result of the entire query is being assigned to the anonymously typed `JobCandidates` variable. The anonymous type is indicated with the `var` keyword (see the "Anonymous Types" sidebar).
- Also in the first line you have the `from...in` keywords indicating the data source. In this instance, you are querying the `db.JobCandidates` entity, and each row is returned in the `JobCandidate` variable.

- The next line of the query contains the where keyword, which is used to limit the results returned. Notice the C# comparison operators are used in the where predicate, and they are not necessarily the same as the SQL operators. In this case, only the JobCandidate whose JobCandidateId property is equal to 1 will be returned.
- Finally, the select keyword comes last and determines what is returned to the JobCandidates collection. In this instance, you are returning a complete JobCandidate object for every matching job candidate.

ANONYMOUS TYPES

In Listing 13-7, the `var` keyword is used to anonymously type the `JobCandidates` variable. Anonymous types, introduced in C# 3.0, are types that the compiler can statically infer at compile time. This is extremely useful in LINQ because it means you don't have to create a new class to support different result sets from every LINQ query you run. The `var` keyword and anonymous types can only be used locally and can only be declared within the body of a method. Also it is very important to realize that anonymous types are not `Object` or variant types. An anonymously typed variable is strongly typed just like any other C# variable; the only difference is that you are leaving it up to the compiler to determine the type at compile time.

LINQ supports several standard query operators to perform querying, restriction, ordering, grouping, partitioning, and many other result set manipulation tasks. These standard query operators are the same regardless of the data source. This means that you can query an XML file using the same operators and similar syntax as you can when you query a SQL database. Listing 13-8 shows a simple LINQ to XML query against an XML data source.

Listing 13-8. Simple LINQ to XML Query

```
using System;
using System.IO;
using System.Xml.Linq;
using System.Collections.Generic;
using System.Linq;

namespace Apress.Samples
{
    class Linq_Query_Example
    {
        static void Main(string[] args)
        {
            XDocument xd = XDocument.Load("c:\\market-summary.xml",
                LoadOptions.PreserveWhitespace);
            XNamespace ms = "apress:sample:market-summary:urn";

            var query = from node in xd.Elements(ms + "market-summary")
                        .Elements(ms + "index")
                        .Elements("daily-summary")
```

```

        where (DateTime)node.Element("date") >=
            DateTime.Parse("2008-01-10T05:00:00Z")
        orderby (DateTime)node.Element("date"),
            (string)node.Parent.Attribute("symbol")
        select node;

    foreach (var node in query)
    {
        Console.WriteLine("Date: {0} Index: {1} ",
            node.Element("date").Value,
            node.Parent.Attribute("symbol").Value);
        Console.WriteLine("Open: {0} High: {1} Low: {2}",
            node.Element("open").Value,
            node.Element("high").Value,
            node.Element("low").Value);
        Console.WriteLine("=====");
    }
}
}
}

```

In this example, an XML document is loaded from the file `market-summary.xml`, which contains summary data from various stock market indexes, like the Dow and the NASDAQ Composite Index.

```

XDocument xd = XDocument.Load("c:\\market-summary.xml",
    LoadOptions.PreserveWhitespace);

```

Once the XML data is loaded into an `XDocument`, you create an `XNamespace` to match the namespace declaration used in the sample file:

```

XNamespace ms = "apress:sample:market-summary:urn";

```

LINQ TO XML NAMESPACES

LINQ to XML provides namespace support through the `XNamespace` object. To create a namespace, just assign the Uniform Resource Identifier (URI) to an `XNamespace` variable. To use the namespace, just concatenate it to the element name when querying. LINQ to XML automatically expands the namespace-qualified element name out to its fully qualified name internally, so you don't have to worry about it. In the example, the element name `ms + "index"` is expanded to its full `{apress:sample:market-summary:urn}index` automatically.

The next step is to perform the actual query. If I were using XPath notation, I would specify a path that looked like this: `ms:market-summary/ms:index/daily-summary`. I would also be restricting the results to matching nodes that contain a date element with a date greater than or equal to 2008-01-10. I am ordering the results by date value and then by the `symbol` attribute of the parent

ms:index element. Finally, the results are returned as a LINQ `IOrderedEnumerable<XElement>`, essentially an enumerable ordered `XElement` collection, similar to an XQuery node sequence. I am using the `var` keyword to declare the query variable that will contain the results, so the compiler will automatically infer this type at compile time—so I don't have to worry about it.

```
var query = from node in xd.Elements(ms + "market-summary")
            .Elements(ms + "index")
            .Elements("daily-summary")
            where (DateTime)node.Element("date") >=
                DateTime.Parse("2008-01-10T05:00:00Z")
            orderby (DateTime)node.Element("date"),
                (string)node.Parent.Attribute("symbol")
            select node;
```

As you can see, the syntax is familiar to SQL developers, even though I am querying an XML document. The remainder of the code simply uses `foreach` to iterate the results and display them on the console.

```
foreach (var node in query)
{
    Console.WriteLine("Date: {0} Index: {1} ",
        node.Element("date").Value,
        node.Parent.Attribute("symbol").Value);
    Console.WriteLine("Open: {0} High: {1} Low: {2}",
        node.Element("open").Value,
        node.Element("high").Value,
        node.Element("low").Value);
    Console.WriteLine("=====");
}
```

The results of the LINQ to XML query against the sample data are shown in Figure 13-11.

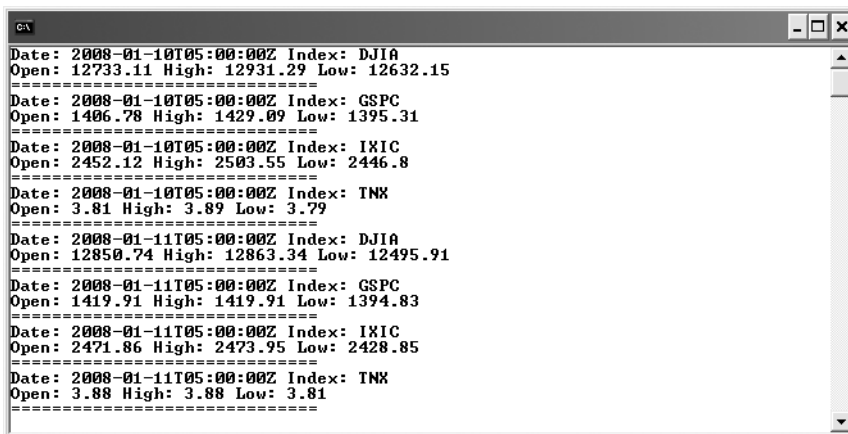


Figure 13-11. Result of simple LINQ to XML query

As I mentioned previously, LINQ supports a large number of standard query operators for querying data and manipulating the results. LINQ to XML supports them all. Some of the most important and useful LINQ standard query operators are shown in Table 13-1.

Table 13-1. *Useful LINQ Standard Query Operators*

Function	Keyword	Description
Restriction	where	The restriction operator restricts/filters the results returned by a query, returning only the items that match the where predicate condition. You can think of this as equivalent to the WHERE clause in SQL.
Projection	select	The projection operator is used to define/restrict the attributes that should be returned in the result collection. The select keyword approximates the SQL SELECT clause.
Join	join	The join operator performs an inner join of two sequences based on matching keys from both sequences. This is equivalent to the SQL INNER JOIN clause.
	join...into	The join...into operator can accept an into clause to perform a left outer join. This format of the join keyword is equivalent to the SQL LEFT OUTER JOIN clause.
Ordering	orderby	The orderby keyword accepts a comma-separated list of keys to sort your query results. Each key can be followed by the ascending or descending keyword. The ascending keyword is the default order. This is equivalent to the SQL ORDER BY clause.
Grouping	group	The group keyword allows you to group your results by a specified set of key values. You can use the group...into syntax if you want to perform additional query operations on the grouped results. The behavior of this keyword approximates the SQL GROUP BY clause.
Subexpressions	let	The let keyword in a query allows you to store subexpressions in a variable during the query. You can use the variable in subsequent query clauses. SQL doesn't have an equivalent for this statement, although subqueries can approximate the behavior in some instances. The best equivalent for this keyword is the XQuery FLWOR (for-let-where-order-by-return) expression let keyword.

LINQ includes several other standard query operators, including partitioning, set, element, aggregation, and other operators. Most of these operators do not have dedicated C# keywords, but instead are invoked as methods of the `System.Query.Sequence` static class.

Listing 13-9 demonstrates a more complex query that queries the same XML document, groups the results by market symbol, and calculates the average open and close value for each index.

Listing 13-9. *LINQ to XML Query with Grouping and Aggregation*

```
using System;
using System.IO;
using System.Xml.Linq;
using System.Collections.Generic;
```

```

using System.Linq;

namespace Apress.Samples
{
    class Linq_Query_Example
    {
        static void Main(string[] args)
        {
            XDocument xd = XDocument.Load("c:\\market-summary.xml",
                LoadOptions.PreserveWhitespace);
            XNamespace ms = "apress:sample:market-summary:urn";

            var query = from node in xd.Elements(ms + "market-summary")
                        .Elements(ms + "index")
                        .Elements("daily-summary")
                        group node by node
                        .Parent
                        .Attribute("symbol") into g
                        select new
                        {
                            Index = g.Key,
                            AverageOpen = g.Average(p =>
(float)p.Element("open")),
                            AverageClose = g.Average(p =>
(float)p.Element("close"))
                        };

            foreach (var node in query)
            {
                Console.WriteLine("Date: {0}, Avg Open: {1}, Avg Close: {2}",
                    node.Index,
                    node.AverageOpen,
                    node.AverageClose);
                Console.WriteLine("=====");
            }
        }
    }
}

```

This code sample begins like the previous one, by populating an `XDocument` from the `market-summary.xml` file and declaring an `XNamespace`.

```

XDocument xd = XDocument.Load("c:\\market-summary.xml",
    LoadOptions.PreserveWhitespace);
XNamespace ms = "apress:sample:market-summary:urn";

```

The LINQ to XML query contains a `group...into` clause, this time to group the results by the `symbol` attribute of the `ms:index` element. The `select` clause creates a new anonymous type structure containing the index symbol and the average of the open and close elements for each group. Notice that I've eliminated the `where` and `orderby` clauses for this example.

```

var query = from node in xd.Elements(ms + "market-summary")
            .Elements(ms + "index")
            .Elements("daily-summary")
            group node by node
            .Parent
            .Attribute("symbol") into g
            select new
            {
                Index = g.Key,
                AverageOpen = g.Average(p => (float)p.Element("open")),
                AverageClose = g.Average(p => (float)p.Element("close"))
            };

```

The results of the preceding code are shown in Figure 13-12.

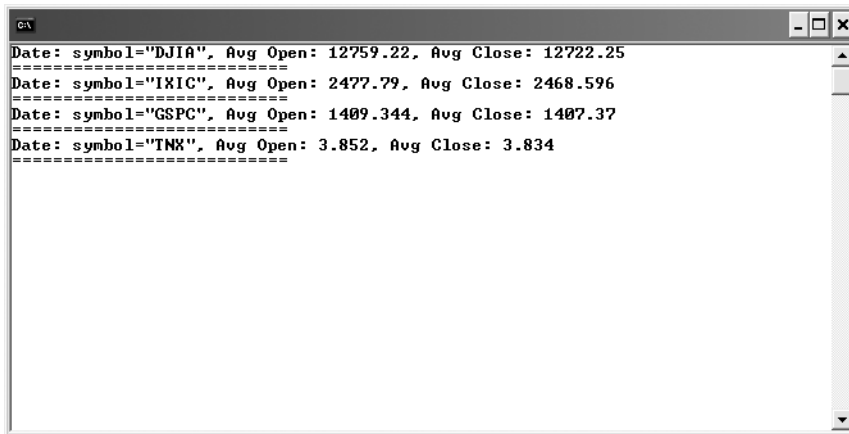


Figure 13-12. Result of LINQ to XML grouping and aggregation example

LAMBDA EXPRESSIONS

C# 2.0 provided a shorthand method of creating methods and instantiating delegates all in one construct, known as *anonymous delegates*. C# 3.0 expanded on this idea with a more compact syntactical construct for creating anonymous methods known as *lambda expressions*. On its most basic level, a lambda expression is simply an anonymous function or method. In Listing 13-8, I've used lambda expressions in two places as arguments to the `Average` method. The lambda expressions I used were as follows:

```

p => (float)p.Element("open")
p => (float)p.Element("close")

```

In both of the preceding lambda expressions, the anonymous function is both declared and instantiated in one expression. Query methods, like the `Average` method, can accept anonymous functions as parameters—a very powerful feature. Lambda expressions offer many advantages (apart from terseness of code) that .NET 2.0 anonymous delegates do not provide:


```

        new XAttribute("symbol",
            node.Parent.Attribute("symbol").Value),
        new XAttribute("date",
            node.Element("date").Value),
        new XAttribute("open",
            node.Element("open").Value),
        new XAttribute("high",
            node.Element("high").Value),
        new XAttribute("low",
            node.Element("low").Value),
        new XAttribute("close",
            node.Element("close").Value),
        new XAttribute("volume",
            node.Element("volume").Value),
        new XAttribute("adjusted-close",
            node.Element("adjusted-close").Value)
    );

    foreach (var x in query)
    {
        Console.WriteLine(x.ToString());
    }
}
}
}

```

This example takes the market-summary.xml file I've used in previous examples and converts its elements to attributes, as shown in Figure 13-13.



Figure 13-13. Result of XML transformation through functional construction

The key to this simple transformation is the functional construction in the select clause of the LINQ query. This select clause maps all the elements of the source document to attributes in a newly constructed XElement.


```
select new XElement("summary",
    new XAttribute("symbol", node.Parent.Attribute("symbol").Value),
    new XAttribute("date", node.Element("date").Value),
    new XAttribute("open", node.Element("open").Value),
    new XAttribute("high", node.Element("high").Value),
    new XAttribute("low", node.Element("low").Value),
    new XAttribute("close", node.Element("close").Value),
    new XAttribute("volume", node.Element("volume").Value),
    new XAttribute("adjusted-close", node.Element("adjusted-close").Value)
);
```

This transformation is a simple example of a very powerful concept. LINQ's functional construction capabilities use your current knowledge of C# as a building block to help you write XML transformations, all without the need to learn XSLT or another transformation language. LINQ's functional construction capabilities are a powerful and expressive method for creating and transforming XML data. The learning curve for functional construction is not nearly as steep as it might be for other technologies, assuming you are starting with knowledge of C# or VB.

Summary

LINQ is a powerful new technology built into the .NET Framework 3.5. The C# and VB teams have gone so far as to change the actual languages to better support LINQ features. In this chapter, I gave you a small taste of the power of LINQ to XML, with an overview and samples of its functionality and features. LINQ to XML offers several advantages over traditional XML querying and manipulation technologies:

- LINQ is directly integrated into its host languages, allowing you to take advantage of .NET functionality without any extraordinary efforts to access the LINQ API.
- LINQ allows you to query any supported data source, including XML data, SQL databases, and .NET objects, using a standard set of query operators.
- LINQ's standard set of query operators and syntax are immediately familiar to developers of SQL, the most widespread query language in the world.
- LINQ to XML, in particular, offers powerful and expressive functional construction that makes creating and transforming XML easier than many other XML APIs.

LINQ to XML is a powerful tool in any XML developer's tool kit, and the number of LINQ-based applications is continuing to grow. For instance, Microsoft is currently testing LINQ to XSD (XML Schema Definition) which will allow you to query strongly typed XML documents. Possibly the most important aspect of LINQ is that it can be used to combine and manipulate data from multiple data sources using a common query syntax and semantics. Using LINQ to XML and LINQ to SQL, for instance, you can easily query, manipulate, calculate, and combine relational and XML data and store the results in any format you choose. As LINQ matures, we can expect a large number of third-party applications and tools designed to take advantage of this exciting new technology.

In the next chapter, I will discuss various "support" applications that use XML or aid in XML development. These applications include such tools as Bulk Copy Program (BCP) and XML for Analysis (XMLA).



XML Support Tools

In addition to SQL Server 2008's built-in XML support, there are many development, support, and administration tools designed to take advantage of XML. Many of them come standard with SQL Server, while others are third-party applications. In this chapter, I will give a brief overview of some of the more useful XML-based tools and utilities available.

Bulk Copy Program

The Bulk Copy Program, or BCP for short, is SQL Server's command-line tool for bulk loading data from flat files into SQL Server tables. BCP can use XML format files to specify the format, data types, and flat file-to-database column mappings. BCP can generate XML format files automatically from the command line, with the `format` and `-x` command-line options, as shown in Listing 14-1. The sample XML format file is generated based on the sample `Sales.CustomerGeography` table I created to demonstrate the SQLXML Bulk Load feature in Chapter 12.

Listing 14-1. *Command Line to Generate XML Format File*

```
bcpx AdventureWorks.Sales.CustomerGeography format nul -x -f
c:\AWCustGeo.fmt -S "(local)" -T -c
```

The command in Listing 14-1 specifies that an XML format file named `c:\AWCustGeo.fmt` should be created based on the `AdventureWorks.Sales.CustomerGeography` table. In this command, the `-f` option specifies the name of the format file to generate, `-x` indicates that the format file should be an XML format file, `-c` tells BCP that the source data files will be in character format, and `-S` and `-T` specify a trusted connection to the local SQL Server instance.

Note Refer to SQL Server Books Online for a description of the full set of BCP command-line options and the BCP XML format file structure.

The sample XML format file generated is shown in Listing 14-2.

Listing 14-2. *Sample BCP XML Format File*

```
<?xml version="1.0"?>
<BCPFORMAT
  xmlns = "http://schemas.microsoft.com/sqlserver/2004/bulkload/format"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
<RECORD>
  <FIELD ID = "1" xsi:type = "CharTerm" TERMINATOR = "\t" MAX_LENGTH = "12"/>
  <FIELD ID = "2" xsi:type="CharTerm" TERMINATOR = "\t" MAX_LENGTH = "41"/>
  <FIELD ID = "3" xsi:type="CharTerm" TERMINATOR = "\t" MAX_LENGTH = "41"/>
  <FIELD ID = "4" xsi:type = "CharTerm" TERMINATOR = "\r\n" MAX_LENGTH = "20"
    COLLATION = "Latin1_General_CI_AS"/>
</RECORD>
<ROW>
  <COLUMN SOURCE = "1" NAME = "CustomerID" xsi:type = "SQLINT"/>
  <COLUMN SOURCE = "2" NAME = "Latitude" xsi:type = "SQLNUMERIC"
    PRECISION = "15" SCALE = "6"/>
  <COLUMN SOURCE = "3" NAME = "Longitude" xsi:type = "SQLNUMERIC"
    PRECISION = "15" SCALE = "6"/>
  <COLUMN SOURCE = "4" NAME = "Resolution" xsi:type = "SQLVARYCHAR"/>
</ROW>
</BCPFORMAT>
```

The XML format file can be used by BCP to bulk load data into a SQL Server table using a command like the one shown in Listing 14-3.

Listing 14-3. *Command Line to Bulk Load a File*

```
bcp AdventureWorks.Sales.CustomerGeography in c:\AWCustGeo.txt -f
c:\AWCustGeo.fmt -T
```

Note The sample code in this chapter is available in the sample downloads.

XML format files also work with the T-SQL BULK INSERT statement and the SQL Server Integration Services (SSIS) Bulk Insert Task, which generates BULK INSERT statements behind the scenes. The BULK INSERT statement shown in Listing 14-4 is the T-SQL equivalent of the BCP command given previously.

Listing 14-4. *Equivalent BULK INSERT Statement*

```
BULK INSERT Sales.CustomerGeography
FROM 'c:\AWCustGeo.txt'
WITH
(
```

```
FORMATFILE = 'c:\AWCustGeo.fmt'
);
```

XML for Analysis

XML for Analysis, or XMLA, is a Simple Object Access Protocol (SOAP)–based XML protocol designed for “universal data access to any standard multidimensional data source.” XMLA was originally proposed by Microsoft as a replacement for their OLEDB for OLAP connectivity technology. XMLA has since become an industry standard with support from companies like Hyperion, IBM, SAS, and SAP. XMLA is important to SQL Server 2008 specifically because it is the primary method of communicating with SQL Server Analysis Services (SSAS).

XMLA allows you to create and manage dimensions and cubes, process cubes, and perform administrative tasks on your OLAP databases. While a complete discussion of SSAS is beyond the scope of this book, I show the sample XMLA script in Listing 14-5 to create the AdventureWorksAS sample database Dim Scenario dimension.

Note AdventureWorksAS is a sample Analysis Services database that uses the AdventureWorks data warehouse (AdventureWorksDW) database as a data source. Both databases are available from the CodePlex web site in the AdventureWorks Business Intelligence (AdventureWorksBI) download at www.codeplex.com.

Listing 14-5. Sample XMLA Script to Create a Dimension

```
<Create xmlns = "http://schemas.microsoft.com/analysiservices/2003/engine">
  <ParentObject>
    <DatabaseID>AdventureWorksAS</DatabaseID>
  </ParentObject>
  <ObjectDefinition>
    <Dimension xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
      xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ddl2 = "http://schemas.microsoft.com/analysiservices/2003/␣
        engine/2"
      xmlns:ddl2_2 = "http://schemas.microsoft.com/analysiservices/2003/␣
        engine/2/2"
      xmlns:ddl100_100 = "http://schemas.microsoft.com/␣
        analysiservices/2008/engine/100/100">
      <ID>Dim Scenario</ID>
      <Name>Dim Scenario</Name>
      <Source xsi:type = "DataSourceViewBinding">
        <DataSourceViewID>Adventure Works DW</DataSourceViewID>
      </Source>
      <UnknownMember>Visible</UnknownMember>
      <ErrorConfiguration>
        <KeyNotFound>ReportAndStop</KeyNotFound>
        <KeyDuplicate>ReportAndStop</KeyDuplicate>
```

```

        <NullKeyNotAllowed>ReportAndStop</NullKeyNotAllowed>
    </ErrorConfiguration>
    <Language>1033</Language>
    <Collation>Latin1_General_CI_AS</Collation>
    <UnknownMemberName>Unknown</UnknownMemberName>
    <Attributes>
        <Attribute>
            <ID>Scenario Key</ID>
            <Name>Scenario Key</Name>
            <Usage>Key</Usage>
            <KeyColumns>
                <KeyColumn>
                    <NullProcessing>UnknownMember</NullProcessing>
                    <DataType>Integer</DataType>
                    <Source xsi:type = "ColumnBinding">
                        <TableID>dbo_DimScenario</TableID>
                        <ColumnID>ScenarioKey</ColumnID>
                    </Source>
                </KeyColumn>
            </KeyColumns>
            <NameColumn>
                <NullProcessing>ZeroOrBlank</NullProcessing>
                <DataType>WChar</DataType>
                <Source xsi:type = "ColumnBinding">
                    <TableID>dbo_DimScenario</TableID>
                    <ColumnID>ScenarioName</ColumnID>
                </Source>
            </NameColumn>
            <OrderBy>Key</OrderBy>
        </Attribute>
    </Attributes>
    <ProactiveCaching>
        <SilenceInterval>-PT1S</SilenceInterval>
        <Latency>-PT1S</Latency>
        <SilenceOverrideInterval>-PT1S</SilenceOverrideInterval>
        <ForceRebuildInterval>-PT1S</ForceRebuildInterval>
        <Source xsi:type = "ProactiveCachingInheritedBinding" />
    </ProactiveCaching>
</Dimension>
</ObjectDefinition>
</Create>

```

The Dim Scenario dimension essentially consists of a hierarchy of the different reporting scenarios supported by AdventureWorksDW: Actual, Budget, Forecast, and Unknown. The sample XMLA script contains the data source information, processing settings, error-handling configuration, and other metadata required to process relational data into SSAS OLAP dimensional data. Once the dimension has been built and processed, it can be browsed using SQL Server Management Studio (SSMS), as shown in Figure 14-1.

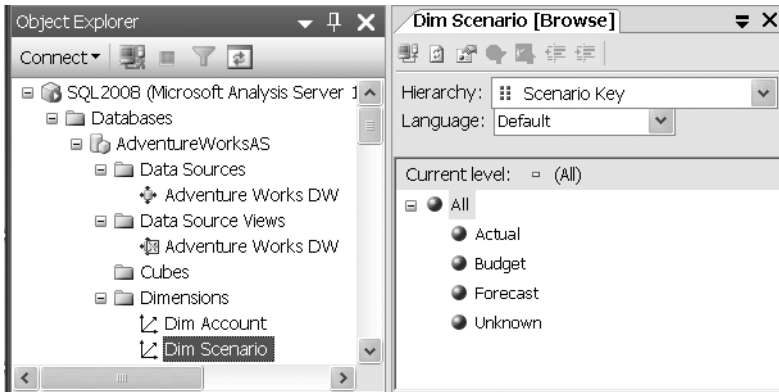


Figure 14-1. *Dim Scenario dimension created with an XMLA script*

SQL Server Integration Services

SQL Server Integration Services (SSIS) provides Extract-Transform-Load (ETL) services for SQL Server. SSIS stores packages in a verbose XML format known as DTSX. Listing 14-6 shows a partial DTSX file listing for a very simple DTSX package.

Listing 14-6. *Partial DTSX File Listing*

```
<?xml version="1.0"?>
<DTS:Executable xmlns:DTS = "www.microsoft.com/SqlServer/Dts"
  DTS:ExecutableType = "SSIS.Package.2">
  . . .
  <DTS:Property DTS:Name = "ObjectName">
    Bulk Insert Task
  </DTS:Property>
  <DTS:Property DTS:Name = "DTSID">
    {D75B6F5C-CAAF-4BC0-BCC3-2630B1E11429}
  </DTS:Property>
  <DTS:Property DTS:Name="Description">Bulk Insert Task</DTS:Property>
  <DTS:Property DTS:Name="CreationName">
    Microsoft.SqlServer.Dts.Tasks.BulkInsertTask.BulkInsertTask, ➤
    Microsoft.SqlServer.BulkInsertTask, Version=10.0.0.0, Culture=neutral, ➤
    PublicKeyToken=89845dcd8080cc91
  </DTS:Property>
  <DTS:Property DTS:Name = "DisableEventHandlers">0</DTS:Property>
  <DTS:ObjectData>
    <BulkInsertTask:BulkInsertTaskData BulkInsertTask:BatchSize = "0"
      BulkInsertTask:CheckConstraints = "True"
      BulkInsertTask:CodePage = "RAW"
      BulkInsertTask:SourceConnectionName =
        "{F4260D5A-ACFB-4E85-ABA6-20E78DFF3187}"
      BulkInsertTask:DestinationConnectionName =
```

```

    "{A3B17064-6B65-4DBF-B806-FE5367455A26}"
    BulkInsertTask:DataFileType = "DTSBulkInsert_DataFileType_Char"
    BulkInsertTask:DestinationTableName =
        "[AdventureWorks].[Sales].[CustomerGeography]"
    BulkInsertTask:FirstRow = "1"
    BulkInsertTask:LastRow = "0"
    BulkInsertTask:UseFormatFile = "True"
    BulkInsertTask:FormatFile = "C:\AWCustGeo.fmt"
    BulkInsertTask:SortedData = ""
    BulkInsertTask:TableLock = "False"
    BulkInsertTask:KeepIdentity = "False"
    BulkInsertTask:KeepNulls = "False"
    BulkInsertTask:FieldTerminator = "Tab"
    BulkInsertTask:RowTerminator = "{CR}{LF}"
    BulkInsertTask:FireTriggers = "False"
    BulkInsertTask:MaximumErrors = "0"
    xmlns:BulkInsertTask =
        "www.microsoft.com/sqlserver/dts/tasks/bulkinserttask" />
</DTS:ObjectData>
. . .
</DTS:Executable>

```

Note Check out MSDN online for details on the DTSX file format. Also, author and SSIS enthusiast Jamie Thomson maintains a great blog on SSIS at <http://blogs.conchango.com/jamiethomson>.

The sample SSIS package shown in Listing 14-6 is a simple ETL package that loads a file into the database via the Bulk Insert Task. Fortunately, Microsoft Business Intelligence Development Studio (BIDS) allows you to create, view, and manipulate SSIS packages using the Visual Studio graphical user interface. The sample SSIS package from Listing 14-6 is shown in graphical format in Figure 14-2.

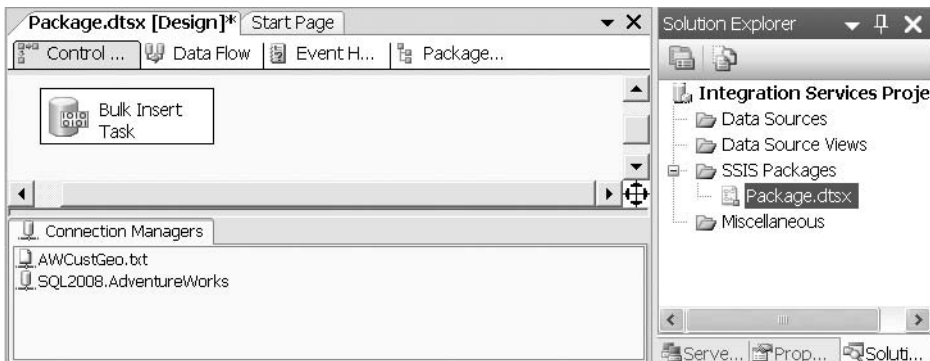


Figure 14-2. SSIS package viewed in BIDS GUI

While most developers don't need to dive into the details of the DTSX file format, it can be useful information for troubleshooting SSIS packages or for specialized applications like automated SSIS package generation.

XML Query Plans

SQL Server provides support for XML-based performance tuning in two key areas: XML query plans and the Database Tuning Advisor (DTA). I'll discuss both in this section. XML query plans provide a window into the SQL query engine's optimization techniques, showing you the logical and physical operators SQL Server applies during the execution of your query. You can generate query plans in several formats—textual, graphical, and XML. A graphical query plan can be saved to a file in XML format with the extension `.sqlplan`. Figure 14-3 shows a simple graphical query plan.

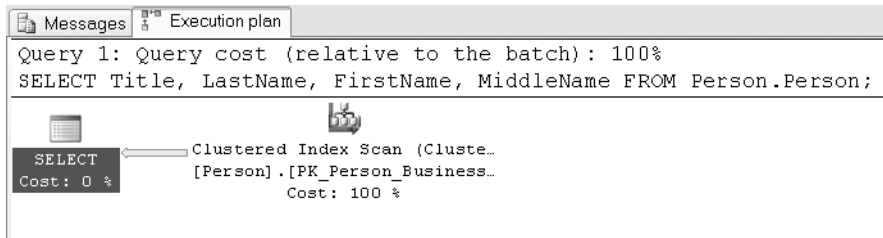


Figure 14-3. Graphical query plan for simple query

The XML equivalent of the graphical query plan shown in Figure 14-3 is presented in Listing 14-7.

Listing 14-7. Simple XML Query Plan

```
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan"
  Version="1.0" Build="10.0.1300.13">
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementText="SELECT Title, LastName, FirstName, MiddleName
&#xD;&#xA;FROM Person.Person;" StatementId="1" StatementCompId="1"
  StatementType="SELECT" StatementSubTreeCost="2.84451"
  StatementEstRows="19972" StatementOptmLevel="TRIVIAL">
          <StatementSetOptions QUOTED_IDENTIFIER="false" ARITHABORT="true"
            CONCAT_NULL_YIELDS_NULL="false" ANSI_NULLS="false"
            ANSI_PADDING="false" ANSI_WARNINGS="false"
            NUMERIC_ROUNDABORT="false" />
          <QueryPlan CachedPlanSize="8" CompileTime="1" CompileCPU="1"
            CompileMemory="64">
            <RelOp NodeId="0" PhysicalOp="Clustered Index Scan"
              LogicalOp="Clustered Index Scan" EstimateRows="19972"
```

```

EstimateIO="2.82238" EstimateCPU="0.0221262" AvgRowSize="175"
EstimatedTotalSubtreeCost="2.84451" TableCardinality="19972"
Parallel="0" EstimateRebinds="0" EstimateRewinds="0">
<OutputList>
  <ColumnReference Database="[AdventureWorks]" Schema="[Person]"
    Table="[Person]" Column="Title" />
  <ColumnReference Database="[AdventureWorks]" Schema="[Person]"
    Table="[Person]" Column="FirstName" />
  <ColumnReference Database="[AdventureWorks]" Schema="[Person]"
    Table="[Person]" Column="MiddleName" />
  <ColumnReference Database="[AdventureWorks]" Schema="[Person]"
    Table="[Person]" Column="LastName" />
</OutputList>
<IndexScan Ordered="0" ForcedIndex="0" NoExpandHint="0">
  <DefinedValues>
    <DefinedValue>
      <ColumnReference Database="[AdventureWorks]" Schema="[Person]"
        Table="[Person]" Column="Title" />
    </DefinedValue>
    <DefinedValue>
      <ColumnReference Database="[AdventureWorks]" Schema="[Person]"
        Table="[Person]" Column="FirstName" />
    </DefinedValue>
    <DefinedValue>
      <ColumnReference Database="[AdventureWorks]" Schema="[Person]"
        Table="[Person]" Column="MiddleName" />
    </DefinedValue>
    <DefinedValue>
      <ColumnReference Database="[AdventureWorks]" Schema="[Person]"
        Table="[Person]" Column="LastName" />
    </DefinedValue>
  </DefinedValues>
  <Object Database="[AdventureWorks]" Schema="[Person]"
    Table="[Person]" Index="[PK_Person_BusinessEntityID]"
    IndexKind="Clustered" />
</IndexScan>
</RelOp>
</QueryPlan>
</StmtSimple>
</Statements>
</Batch>
</BatchSequence>
</ShowPlanXML>

```

I briefly discussed SQL Server's cached XML query plans in Chapter 5 to demonstrate XQuery queries. One of the interesting features of the .sqlplan XML query plan files is that you can have users send them to you to help in remotely determining the cause of slow-running queries. Once received, you can open them up in SSMS and view the graphical query plan. You can also use

XML query plans with the `USE PLAN` query hint to force the optimizer to choose a more efficient query plan than it might come up with. This is particularly useful for situations when a less efficient query plan is generated after applying a SQL Server service pack or hotfix, or when the plan generated is less efficient when running the query on a server with different hardware.

Caution The `USE PLAN` hint should be used with caution because it overrides the optimizer and could adversely affect performance.

You can also use XML query plans to create plan guides that test query performance under different conditions without completely rewriting the query. Overall, XML query plans are an extremely useful XML-based tool for optimizing SQL queries.

Database Tuning Advisor

Another useful tool for performance tuning is the DTA. DTA is the updated replacement for the SQL Server 2000 Index Tuning Wizard. DTA can accept XML input files to help you optimize indexes, indexed views, and partitions. Listing 14-8 shows a sample DTA XML input file.

Listing 14-8. *DTA XML Input File*

```
<?xml version = "1.0" encoding = "utf-8" ?>
<DTAXML xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xmlns = "http://schemas.microsoft.com/sqlserver/2004/07/dta">
  <DTAInput>
    <Server>
      <Name>(local)</Name>
      <Database>
        <Name>AdventureWorks</Name>
        <Schema>
          <Name>Sales</Name>
          <Table>
            <Name>SalesOrderHeader</Name>
          </Table>
        </Schema>
      </Database>
    </Server>
    <Workload>
      <EventString Weight = "100.01">
        SELECT SalesOrderID, SubTotal
        FROM Sales.SalesOrderHeader
        WHERE SubTotal > 100.00
        AND SubTotal < 1000.00
        ORDER BY SubTotal;
      </EventString>
    </Workload>
  </DTAInput>
</DTAXML>
```

```

</Workload>
<TuningOptions>
  <TuningTimeInMin>120</TuningTimeInMin>
  <StorageBoundInMB>1500</StorageBoundInMB>
  <FeatureSet>IDX</FeatureSet>
  <Partitioning>NONE</Partitioning>
  <KeepExisting>NONE</KeepExisting>
  <OnlineIndexOperation>OFF</OnlineIndexOperation>
  <DatabaseToConnect>AdventureWorks</DatabaseToConnect>
</TuningOptions>
</DTAInput>
</DTAXML>

```

Figure 14-4 shows the results DTA produces when the sample XML input file is run through it.

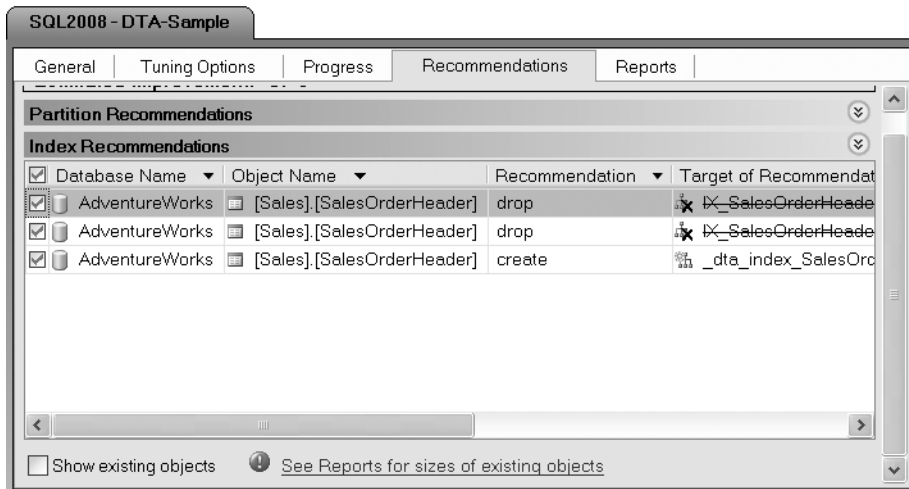


Figure 14-4. Sample XML input file DTA results

XMLSpy

XML was designed to be easy to edit by hand in any word processor or text editing utility, regardless of platform. If your favorite text editor can save documents in a plain text format, you can use it to edit XML. Since XML was introduced, however, better and more powerful editors have been created. I'll look at some popular XML viewing and editing tools in this section.

It's not a Microsoft product, and it doesn't come bundled with SQL Server, but I couldn't let this chapter go without mentioning what may well be the best XML editor on the market. The current release of the Altova® XMLSpy® software, XMLSpy 2008, is the industry standard XML

modeling development environment. XMLSpy makes it very easy to create just about any type of XML document including W3C-compliant XML schemas and XML Stylesheet Language (XSL) transformations. In fact, many of the XML samples, XSL transformations, and XML schemas I created for this book were initially modeled and tested using XMLSpy 2008. Figure 14-5 is a screenshot of an XML schema viewed in XMLSpy.

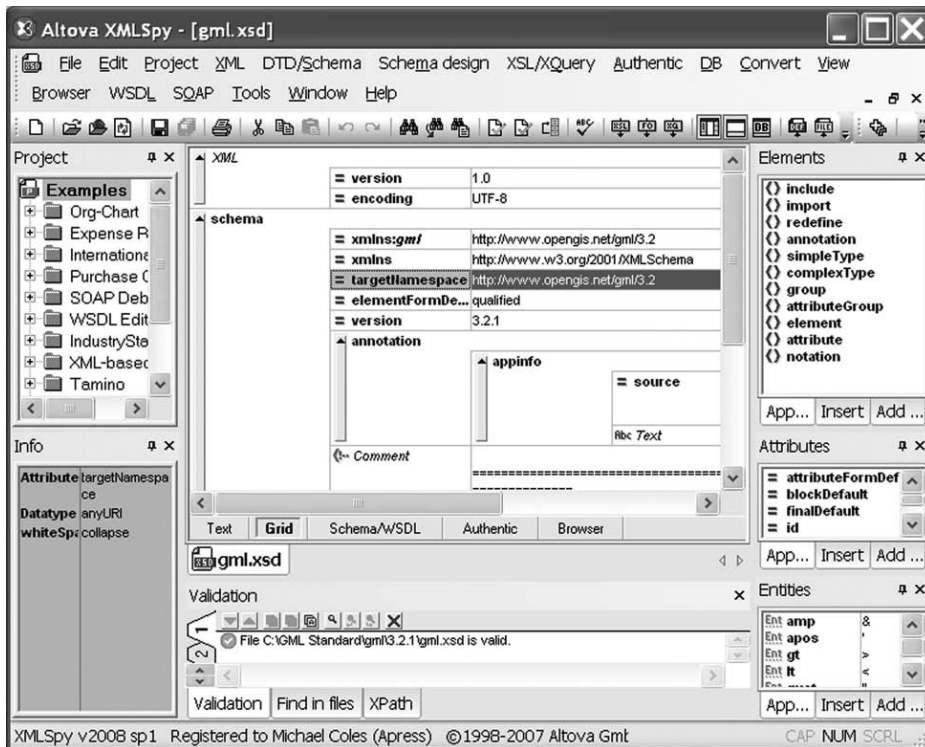


Figure 14-5. Designing an XML schema in XMLSpy 2008

XMLSpy has a lot of features geared toward developers like XML “pretty-print” formatting and XML tag autocomplete, based on XML schemas, to make XML document editing and creation easy. XMLSpy can check your XML documents for well-formedness and validate them against an XML schema or Document Type Definition (DTD). It also includes features for performing XSL transformations, XPath and XQuery creation, testing, profiling, and debugging. Figure 14-6 shows the result of testing an XPath expression against an XML document in XMLSpy.

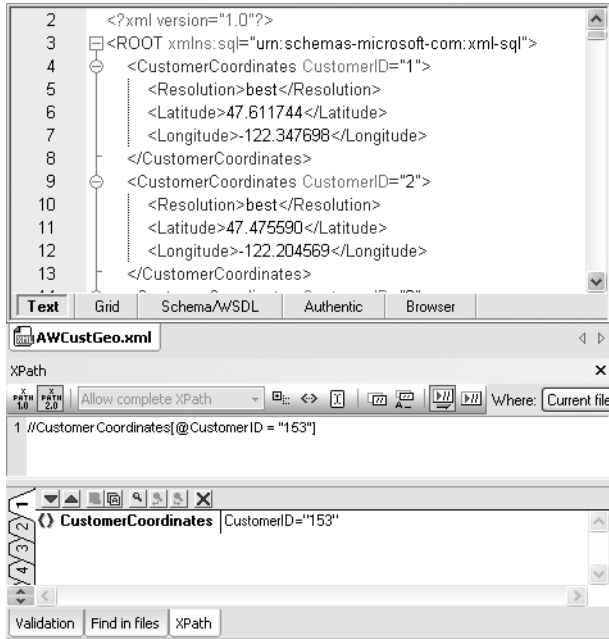


Figure 14-6. Testing an XPath expression in XMLSpy

XMLSpy is a complete Integrated Development Environment (IDE) for designing, testing, and manipulating all XML. Any serious XML developer or data modeler should consider acquiring this software. More information about the current release of XMLSpy can be found at the Altova web site at www.altova.com.

Web Browsers

The most popular modern web browsers include built-in default XSL transformations to transform any well-formed XML document into viewable XHTML format. While the browsers do not allow XML editing, the autocoloring and formatting features make browsers extremely useful for viewing your XML while designing, developing, and testing. Figure 14-7 shows an XML document as viewed in Mozilla Firefox.

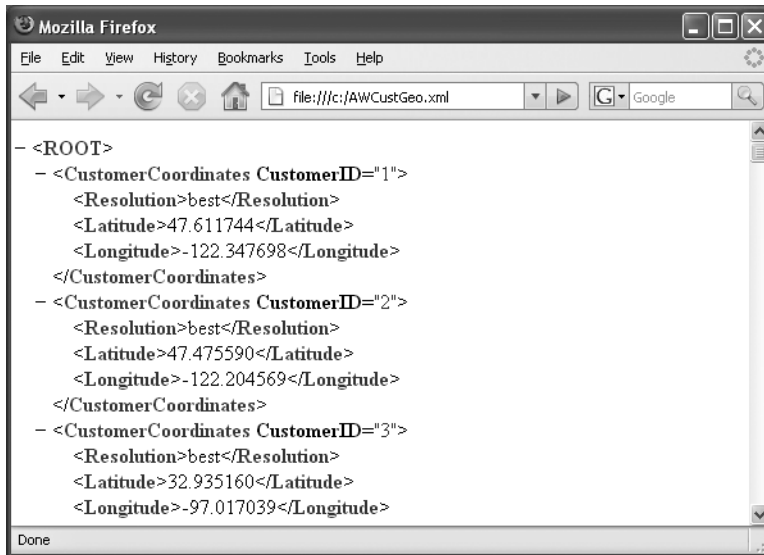


Figure 14-7. Viewing XML in Firefox

Of course, Microsoft's flagship browser product, Internet Explorer (IE), also includes functionality to view XML in the browser. Figure 14-8 shows an XML document as viewed in IE 6.

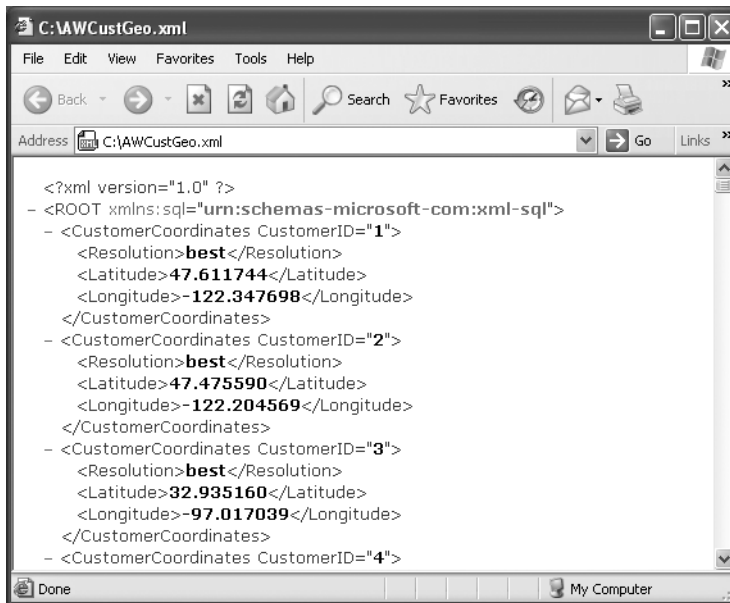


Figure 14-8. Viewing XML in IE

Visual Studio

The VS IDE, including the products that use the VS IDE (the free express editions of Microsoft's Visual languages and BIDS), has built-in support for editing XML documents. The VS IDE can “pretty-print” format your XML to make it more easily readable while editing, and it can automatically generate XML schemas based on a given XML document.

While not as powerful as XMLSpy for XML development, the VS IDE is an economical alternative for simple XML data modeling tasks. Figure 14-9 demonstrates editing an XML document in the VS IDE.

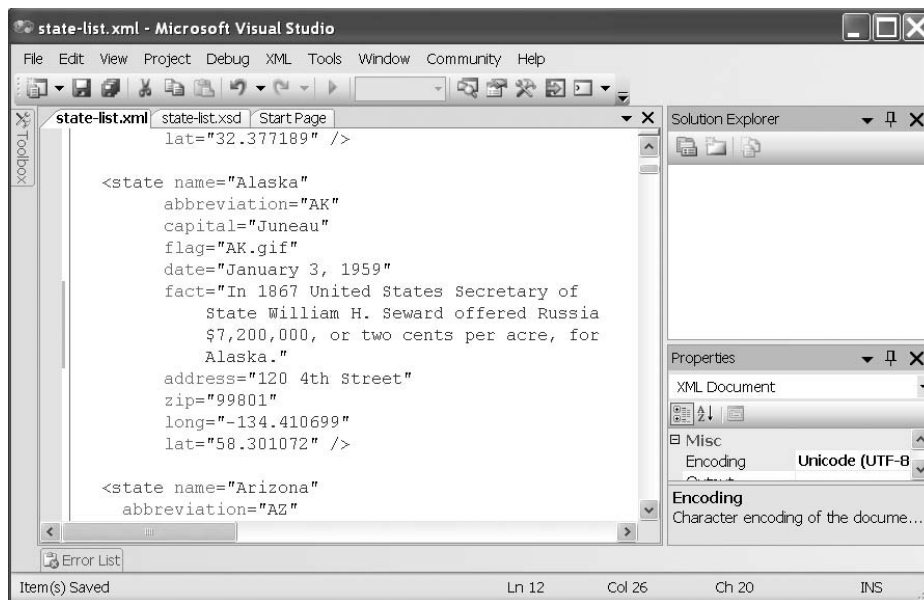


Figure 14-9. *Editing an XML document in the VS IDE*

Summary

Throughout this book, I've talked about SQL Server-specific XML features and functionality. The discussion has mostly centered on T-SQL features, .NET support, and OLEDB/COM-based SQLXML functionality. I decided to finish the book with a survey of additional XML technologies that work in support of SQL Server and XML development. As you can see, there is a wide range of XML-based technologies that work in support of SQL Server development. The features discussed in this chapter include all of the following:

- BCP XML format files for bulk loading data
- XMLA for communicating with OLAP databases
- XML-formatted SSIS packages to support ETL solutions

- Performance tuning SQL Server queries with XML query plans and the Database Tuning Advisor
- Software packages that support viewing and editing XML content, including web browsers, XMLSpy, and the VS IDE

This list is not exhaustive by any means, but it is indicative of the scope of SQL Server XML integration in the SQL Server 2008 release and the variety of tools available to XML data modelers and developers.

At this point, I'd like to take a moment to thank you for reading this book, and I hope you find its content useful in your SQL Server–based XML development efforts. I sincerely hope you've enjoyed the content as much as I've enjoyed bringing it to you, and I wish you well in all your endeavors. Be sure to check my *Pro SQL Server XML* blog at <http://blogs.sqlservercentral.com/prosqlxml> for more XML programming tips and tricks.



W3C and Other References

The World Wide Web Consortium (W3C) has a very specific process in place to fulfill its goal of “developing interoperable technologies” for the Web. In a nutshell, the process goes like this:

1. The W3C initially publishes a *working draft* of a document for review by the community.
2. A document can undergo several rounds of updates, each being released as a new working draft.
3. A working draft that W3C believes has been widely reviewed and satisfies the relevant technical requirements is released as a *candidate recommendation*.
4. A candidate recommendation that has proven technically sound is promoted to a *proposed recommendation*, which is sent to the W3C Advisory Committee for final endorsement.
5. A proposed recommendation that is endorsed by the W3C members and director is disseminated as a *W3C recommendation*.

W3C recommendations are the W3C equivalent of standards published by other organizations, like ISO and ANSI. Throughout this book, I’ve referenced several W3C specifications, which are in various phases of the process previously described. This appendix provides references to these and other relevant documents.

W3C Specifications

The W3C has published several recommendations, including dozens of working drafts. In this section, I’ve listed some of the key recommendations that come in handy for SQL Server XML programming.

Cascading Style Sheets, Levels 1, 2, and 3

This set of W3C recommendations work with the XHTML standard to offer flexible and powerful control over web publishing and presentation. The CSS recommendations are available at www.w3.org/TR/REC-CSS1, www.w3.org/TR/REC-CSS2, and www.w3.org/TR/css3-selectors.

Extensible Markup Language (XML) 1.0

This W3C recommendation is the basis for all work in XML. It explains the design goals, purpose, and specifications for XML 1.0. The recommendation is available at www.w3.org/TR/xml.

Extensible Markup Language (XML) 1.1

The W3C XML 1.1 Recommendation provides some additions to XML 1.0, including expanded support for Unicode, Extended Binary Coded Decimal Interchange Code (EBCDIC), and some slight tweaks to XML naming conventions. It is recommended that you use XML 1.0 if you do not have a need for these new features. The recommendation is available at www.w3.org/TR/xml11.

Namespaces in XML 1.0

This W3C recommendation adds namespace support for qualifying element and attribute names in XML documents. The recommendation is available at www.w3.org/TR/xml-names.

XHTML 1.0: The Extensible HyperText Markup Language, Second Edition

This W3C recommendation reformulates the HTML 4 standard in terms of XML. While similar to HTML, XHTML is cleaner than HTML and can be parsed by any XML parser. The recommendation is available at www.w3.org/TR/xhtml1. The XHTML 1.1 working draft is available at www.w3.org/TR/xhtml11.

XML Fragment Interchange (Candidate Recommendation)

This W3C candidate recommendation has had no active work performed on it recently. Is noteworthy primarily for its definition of XML document fragments (as opposed to well-formed documents), which XML Query (XQuery) language must be able to query and/or return. The recommendation is available at www.w3.org/TR/xml-fragment.

XML Information Set (Infoset)

This W3C recommendation defines an abstract data set known as the Infoset for specifications that need to refer to the information in well-formed XML documents. The recommendation is available at www.w3.org/TR/xml-infoset.

XML Path Language Version 1.0

This W3C recommendation defines the XML Path Language (XPath), which is a language for addressing parts of XML documents. XPath was designed specifically to support the requirements of Extensible Stylesheet Language Transformations (XSLT) and XPointer. XQuery also borrows heavily from XPath. This recommendation is available at www.w3.org/TR/xpath. The XPath 2.0 Recommendation is available at www.w3.org/TR/xpath20.

XML Query 1.0 Use Cases (Working Group Note)

This W3C working group note provides a series of use cases to illustrate important XQuery applications. This working group note is available at www.w3.org/TR/xquery-use-cases.

XML Schema Part 0: Primer, Second Edition

A document designed as an introduction to XML Schema facilities. Highly recommended for those who want to understand, in relatively nontechnical language, what XML Schema offers. The recommendation is available at www.w3.org/TR/xmlschema-0.

XML Schema Part 1: Structures, Second Edition

This W3C recommendation defines XML Schema structures and the XML Schema Definition Language (XSD). SQL Server implements a subset of this standard. The working draft is available at www.w3.org/TR/xmlschema-1. The working draft of XML Schema 1.1 Part 1 is available at www.w3.org/TR/xmlschema11-1.

XML Schema Part 2: Datatypes, Second Edition

This W3C recommendation defines XML Schema data types that can be used when creating XML schemas. SQL Server implements a subset of this standard, as detailed in Appendix B. The recommendation is available at www.w3.org/TR/xmlschema-2. The working draft of XML Schema 1.1 Part 2 is available at www.w3.org/TR/xmlschema11-2.

XQuery 1.0: An XML Query Language

This W3C recommendation defines the XQuery language. The XQuery language is designed to operate on the abstract, logical structure of an XML document (the data model) rather than on its surface syntax. The recommendation is available at www.w3.org/TR/xquery.

XQuery 1.0 and XPath 2.0 Data Model (XDM)

This W3C recommendation defines a method of representing XML documents with a logical structure to enable efficient querying and manipulation. The recommendation is available at www.w3.org/TR/xpath-datamodel.

XQuery Update Facility (Working Draft)

This W3C working draft discusses facilities for updating XDM instances via XQuery extensions. SQL Server implements a proprietary set of XQuery extensions known as XML Data Manipulation Language (XML DML), but this W3C working draft explains the rationale for these extensions and provides some insight into what we might expect in the near future. The working draft is available at www.w3.org/TR/xqupdate.

XSL Transformations

This W3C recommendation defines XSLT 1.0, a language for transforming XML documents into other XML documents. The recommendation is available at www.w3.org/TR/xslt. The XSLT 2.0 Recommendation is available at www.w3.org/TR/xslt20.

Other Useful Documents

There are several other standards bodies out there who have published standards that are useful and relevant for SQL Server and XML programmers. Here I've listed a few of them, but this list is not exhaustive.

Microsoft SQL Server Books Online

The direct link to the SQL Server 2008 Books Online launch page is subject to change, and there are different versions for different languages. The US English language version of Books Online and other SQL Server reference works can be found at <http://msdn2.microsoft.com/en-us/library/default.aspx>.

ISO SQL

The ISO SQL standards, including SQL-92, SQL:2003, and others, are available for purchase at www.iso.org.

Unicode Standards

The Unicode character encoding and collation standards are available from the Unicode Consortium at www.unicode.org/Public.

IANA Language Subtag Registry

The Internet Assigned Numbers Authority (IANA) provides a reference list of language codes for use on the Internet. The IANA registry is the official reference for XML language codes and is located at www.iana.org/assignments/language-subtag-registry.



SQL Server XQuery Data Types

SQL Server supports the data types defined in the XQuery/XPath Data Model (XDM). The supported data types are listed here with their definitions (see Table B-1). The diagram from Chapter 5 showing the relationships between the XDM data types is also reproduced here (see Figure B-1) for quick reference.

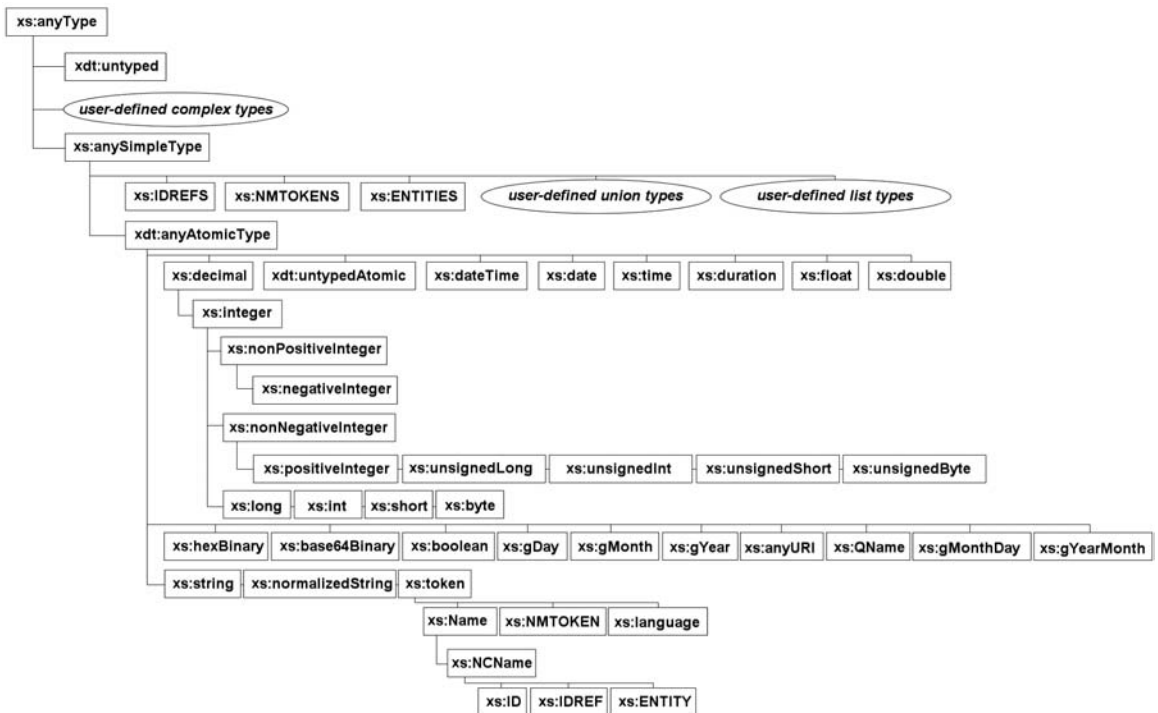


Figure B-1. XDM data type system

Table B-1. *XDM Data Type Descriptions***Base Types**

<code>xs:anySimpleType</code>	This is the base type for all simple built-in types.
<code>xs:anyType</code>	This is the base type for <code>xs:anySimpleType</code> and complex built-in types.

Temporal (Time) Types

<code>xs:date</code>	This type represents a Gregorian calendar–based date value exactly one day in length, represented in the format <code>yyyy-mm-dd[time_zone]</code> . The <i>time_zone</i> can be a capital Z for <i>zero-meridian (UTC)</i> , or in the format <code>+/-hh:mm</code> to represent a UTC offset. An example of a valid <code>xs:date</code> is <code>2006-12-25Z</code> , which represents December 25, 2006, UTC time.
<code>xs:dateTime</code>	This type represents a Gregorian calendar–based date and time value with precision to 1/1000th of a second. The format is <code>yyyy-mm-ddThh:mm:ss.sss[time_zone]</code> . Time is specified using a 24-hour clock. As with <code>xs:date</code> , the <i>time_zone</i> can be a capital Z (UTC) or a UTC offset in the format <code>+/-hh:mm</code> . A valid <code>xs:dateTime</code> value is <code>2006-10-30T13:00:59.500-05:00</code> , which represents October 30, 2006, 1:00:59.5 PM, US Eastern Standard Time. Unlike SQL Server 2005, in SQL Server 2008 the <code>xs:dateTime</code> type maintains the time zone information you assign instead of automatically converting all date/time values to a single time zone. The time zone is also not mandatory in SQL Server 2008.
<code>xs:duration</code>	This type represents a Gregorian calendar–based temporal (time-based) duration, represented as <code>PyyyyYmmMddThhHmMss.ssss</code> . <code>P0010Y03M12DT00H00M00.000S</code> , for instance, represents 10 years, 3 months, 12 days.
<code>xs:gDay</code>	This type represents a Gregorian calendar–based day. The format is <code>---dd[time_zone]</code> (notice the three preceding hyphen - characters.) The <i>time_zone</i> is optional. A valid <code>xs:gDay</code> value is <code>---09Z</code> , which stands for the ninth day of the month, UTC time.
<code>xs:gMonth</code>	This type represents a Gregorian calendar–based month. The format is <code>--mm[time_zone]</code> (notice the two preceding hyphen - characters.) The <i>time_zone</i> is optional. A valid <code>xs:gMonth</code> value is <code>-12</code> , which stands for December.
<code>xs:gMonthDay</code>	This type represents a Gregorian calendar–based month and day. The format is <code>--mm-dd[time_zone]</code> (notice the preceding hyphens --). The <i>time_zone</i> for this data type is optional. A valid <code>xs:gMonthDay</code> value is <code>--02-29</code> for February 29.
<code>xs:gYear</code>	This type represents a Gregorian calendar–based year. The format is <code>yyyy[time_zone]</code> . The <i>time_zone</i> is optional. The year can also have a preceding hyphen character - indicating a negative (BCE) year as opposed to a positive (CE) date. A valid <code>xs:gYear</code> value is <code>-0044</code> for 44 BCE. Notice that all four digits are required in the year representation, even for years that can be normally represented with less than four digits.
<code>xs:gYearMonth</code>	This type represents a Gregorian calendar–based year and month. The format is <code>yyyy-mm[time_zone]</code> . The <i>time_zone</i> for this data type is optional and can be Z or a UTC offset. A valid <code>xs:gYearMonth</code> value is <code>2001-01</code> for January 2001.
<code>xs:time</code>	This type represents a time value with precision to 1/1000th of a second, using a 24-hour clock representation. The format is <code>hh:mm:ss.sss[time_zone]</code> . As with other temporal data types, <i>time_zone</i> can be Z (UTC) or a UTC offset in the format <code>+/-hh:mm</code> . A valid <code>xs:time</code> value is <code>23:59:59.000-06:00</code> , which represents

11:59:59 PM, US Central Standard Time. The canonical representation of midnight in 24-hour format is 00:00:00.

Binary Types

`xs:base64Binary`

This type represents Base64-encoded binary data. Base64-encoding symbols are defined in RFC 2045 (www.ietf.org/rfc/rfc2045.txt) as letters A–Z, a–z, 0–9, +, /, and trailing = signs. White-space characters are also allowed. An example of a valid `xs:base64Binary` value is QVB5ZXNzIEJvb2t2IEFuZCBTUUwgU2VydmdVYIDlWMDU=.

`xs:hexBinary`

This type represents hexadecimal-encoded binary data. The symbols defined for encoding data in hexadecimal format are 0–9, A–F, and a–f. Upper- and lowercase letters A–F are considered equivalent by this data type. A valid `xs:hexBinary` value is 6170726573732E636F6D.

Boolean Type

`xs:boolean`

This type represents a Boolean binary truth value. The values supported are true or false, 0 (false) or 1 (true). An example of a valid `xs:boolean` value is true.

Numeric Types

`xs:byte`

This type represents an 8-bit signed integer in the range -128 to +127.

`xs:decimal`

This type represents an exact decimal value up to 38 digits in length. These numbers can have up to 28 digits before the decimal point and up to 10 digits after the decimal point. A valid `xs:decimal` value is 8372.9381.

`xs:double`

This type represents a double-precision floating point value patterned after the IEEE standard for floating point types. The representation of values is similar to `xs:float` values $nE[+/-]e$, where n is the mantissa followed by the letter E or e and an exponent e . The range of valid values for `xs:double` is approximately -1.79E+308 to -2.23E-308, 0, and +2.23E-308 to +1.79E+308.

`xs:float`

This type represents an approximate single-precision floating point value per the IEEE 754-1985 standard. The format for values of this type is nEe , where n is a decimal mantissa followed by the letter E or e and an exponent. The value represents $n \cdot 10^e$. The range for `xs:float` values is approximately -3.4028e+38 to -1.401298E-45, 0, and +1.401298E-45 to +3.4028e+38. The special values -INF and +INF represent negative and positive infinity. SQL Server does not support the XQuery-specified special value NaN, which stands for “Not a Number.” A valid `xs:float` value is 1.98E+2.

`xs:int`

This type represents a 32-bit signed integer in the range -2147483648 to +2147483647.

`xs:integer`

This type represents an integer value up to 28 digits in length. A valid `xs:integer` value is 76372.

`xs:long`

This type represents a 64-bit signed integer in the range -9223372036854775808 to +9223372036854775807.

`xs:negativeInteger`

This type represents a negative nonzero integer value derived from the `xs:integer` type. It can be up to 28 digits in length.

`xs:nonNegativeInteger`

This type represents a positive or zero integer value derived from the `xs:integer` type. It can be up to 28 digits in length.

`xs:nonPositiveInteger`

This type represents a negative or zero integer value derived from the `xs:integer` type. It can be up to 28 digits in length.

<code>xs:positiveInteger</code>	This type represents a positive nonzero integer value derived from the <code>xs:integer</code> type. It can be up to 28 digits in length.
<code>xs:short</code>	This type represents a 16-bit signed integer in the range -32768 to +32767.
<code>xs:unsignedByte</code>	This type represents an unsigned 8-bit integer in the range 0 to 255.
<code>xs:unsignedInt</code>	This type represents an unsigned 32-bit integer in the range 0 to +4294967295.
<code>xs:unsignedLong</code>	This type represents an unsigned 64-bit integer in the range 0 to +18446744073709551615.
<code>xs:unsignedShort</code>	This type represents an unsigned 16-bit integer in the range 0 to +65535.

String Types

<code>xs:ENTITY</code>	This type is equivalent to the ENTITY type from the XML 1.0 standard. The lexical space has the same construction as an <code>xs:NCName</code> .
<code>xs:ENTITIES</code>	This type is a space-separated list of ENTITY types.
<code>xs:ID</code>	This type is equivalent to the ID attribute type from the XML 1.0 standard. An <code>xs:ID</code> value has the same lexical construction as an <code>xs:NCName</code> .
<code>xs:IDREF</code>	This type represents the IDREF attribute type from the XML 1.0 standard. The lexical space has the same construction as an <code>xs:NCName</code> .
<code>xs:IDREFS</code>	This type is a space-separated list of IDREF attribute types.
<code>xs:language</code>	This type is a language identifier string. This data type represents natural language identifiers as specified by RFC 3066 (www.ietf.org/rfc/rfc3066.txt), and a complete list of language codes is maintained by the IANA registry at www.iana.org/assignments/language-subtag-registry . Language identifiers must conform to the regular expression pattern <code>[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*</code> . An example of a valid language identifier is <code>tlh</code> , which is the identifier for the Klingon language.
<code>xs:Name</code>	This type is an XML name string. A name string must match the XML-specified production for Name. Per the standard, a Name must begin with a letter, an underscore, or a colon and may then contain a combination of letters, numbers, underscores, colons, periods, hyphens, and various other characters designated in the XML standard as <i>combining characters</i> and <i>extenders</i> . Refer to the XML standard at www.w3.org/TR/2000/WD-xml-2e-20000814#NT-Name for specific information about these additional allowable Name characters.
<code>xs:NCName</code>	This type is a noncolonized name. The format for an <code>xs:NCName</code> is the same as for <code>xs:Name</code> , but without colon characters.
<code>xs:NMTOKEN</code>	This type is an NMTOKEN type from the XML 1.0 standard. An <code>xs:NMTOKEN</code> value is composed of any combination of letters, numbers, underscores, colons, periods, hyphens, and XML combining characters and extenders.
<code>xs:NMTOKENS</code>	This type is a space-separated list of <code>xs:NMTOKEN</code> values.
<code>xs:normalizedString</code>	This type is an XML whitespace-normalized string. A whitespace-normalized string is one that does not contain the white-space characters <code>#x9</code> (tab), <code>#xA</code> (line feed), or <code>#xD</code> (carriage return).
<code>xs:string</code>	This type is an XML character string.
<code>xs:token</code>	This type is an XML whitespace-normalized string with the following additional restrictions on <code>#x20</code> (space) characters: (1) it can have no leading or trailing spaces, and (2) it cannot contain any sequences of two space characters in a row.



XML Schema Reference

This appendix provides a quick reference for the W3C XML Schema recommendation, as implemented by SQL Server 2008. SQL Server implements XML Schema in the form of XML schema collections, which allow you to create typed `xml` data type instances as described in Chapter 4. In this appendix, I'll define the components that are used to create an XML schema.

XML Schema is the W3C Recommendation for creating schemas that define syntactic, structural, and value constraints on XML data. XML that conforms to a given XML schema is called *valid XML*. SQL Server 2008 supports typed `xml` data type instances, which are instances associated with XML schema collections. XML schema collections are groups of XML schemas that have been registered with SQL Server. SQL Server uses the `CREATE XML SCHEMA COLLECTION`, `ALTER XML SCHEMA COLLECTION`, and `DROP XML SCHEMA COLLECTION` statements to create and manage XML schema collections.

At the highest level, an XML schema is a container that may contain the following components:

- Type definitions
- Attribute declarations
- Element declarations
- Attribute group declarations
- Model group definitions
- Notation declarations
- Annotations

XML Schema components are defined using element information items, which I will summarize in this appendix. Note that this is a quick reference for XML Schema syntax; XML Schema semantics and behavior are detailed in Chapter 4.

Element Information Items

Element information items are the basic building blocks of XML schemas. The element information item is an XML element that is recognized as part of the XML Schema recommendation. The element information items are used to define and restrict the structure and content of your

XML schemas. In this section, I will discuss each of the supported element information items individually.

FORMATTING CONVENTIONS

The formats for the XML schema element information items in this chapter follow a specific formatting convention, which is adapted from the W3C XML Schema recommendation, though somewhat simplified. I have used *italicized* code font names as placeholders for actual values, such as *ID* in the following example:

```
id = "ID"
```

When you have a choice of predetermined attribute values, the choices are separated by a vertical bar (|). The following example represents a choice between the values *qualified* and *unqualified*.

```
form = "qualified" | "unqualified"
```

When multiple values can be selected from a group, the group is enclosed with parentheses, with choices separated by the vertical bar. A group from which you can choose multiple values is followed by the ellipsis (. . .). The following example demonstrates a choice between the values *#all* or a combination of one or both of *extension* and *restriction*:

```
block = "#all" | ( "extension" | "restriction" ) . . .
```

Finally, if an attribute has a default value, it is represented as a value following the possible attribute value list, separated by a colon (:) and underlined for emphasis. The following example is a choice between 0 and 1 with a default value of 1 if the attribute is omitted:

```
minOccurs = "0" | "1" : "1"
```

all Element

The *all* element specifies that contained elements can appear in any order exactly zero or one time each.

```
<all id = "ID"
  maxOccurs = "1" : "1"
  minOccurs = "0" | "1" : "1" >
  . . .
</all>
```

annotation Element

The *annotation* element information item provides mechanisms for documenting your XML schemas inline. The *annotation* element information item can contain both *appinfo* and *documentation* element information items. The *appinfo* element information item is generally used to include machine-readable documentation, while the *documentation* element information item is usually used for human-readable documentation. Following is the syntax of the *annotation*, *appinfo*, and *documentation* element information items:

```
<annotation&nbsp;id = "ID" >
  . . .
</annotation>
```

any Element

The any element allows you to extend the XML document to use elements not explicitly specified by the XML schema.

```
<any id = "ID"
  maxOccurs = "nonNegativeInteger" | "unbounded" : "1"
  minOccurs = "nonNegativeInteger" : "1"
  namespace = "##any" | "##other" | ( "##local" | "##targetNamespace" | "anyURI" ) ➡
  . . . : "##any"
  processContents = "lax" | "skip" | "strict" : "strict" >
  . . .
</any>
```

anyAttribute Element

The anyAttribute element extends the XML document with attributes not specified by the XML schema.

```
<anyAttribute id = "ID"
  namespace = "##any" | "##other" | ( "##targetNamespace" | "##local" | "anyURI" ) ➡
  . . . : "##any"
  processContents = "lax" | "skip" | "strict" : "strict" >
  . . .
</anyAttribute>
```

appInfo Element

The appInfo element allows you to specify information to be used by the application processing your XML document.

```
<appInfo source = "anyURI">
  . . .
</appInfo>
```

attribute Element

The attribute element information item allows you to define XML attributes in your XML schemas. The attribute element information item may contain annotation and simpleType element information items. Following is the format for the attribute element information item:

```
<attribute default = "string"
  fixed = "string"
  form = "qualified" | "unqualified"
  id = "ID"
```

```

    name = "NCName"
    ref = "QName"
    type = "QName"
    use = "optional" | "prohibited" | "required" : "optional" >
    . . .
</attribute>

```

attributeGroup Element

The attributeGroup element groups a set of attribute declarations for use in complexType definitions.

```

<attributeGroup id = "ID"
  ref = "QName" >
  . . .
</attributeGroup>

```

choice Element

The choice element limits XML content to only one of the contained elements.

```

<choice id = "ID"
  maxOccurs = "nonNegativeInteger" | "unbounded" : "1"
  minOccurs = "nonNegativeInteger" : "1" >
  . . .
</choice>

```

complexContent Element

The complexContent element defines restrictions or extensions on a complexType.

```

<complexContent id = "ID"
  mixed = "boolean" : "false" >
  . . .
</complexContent>

```

complexType Element

The complexType element information item defines XML Schema complex user-defined types. This element information item can contain annotation, simpleContent, complexContent, group, all, choice, sequence, attribute, attributeGroup, and anyAttribute element information items.

```

<complexType
  abstract = "boolean" : "false"
  block = "#all" | ( "extension" | "restriction" ) . . .
  final = "#all" | ( "extension" | "restriction" ) . . .
  id = "ID"

```

```

    mixed = "boolean" : "false"
    name = "NCName" >
    . . .
</complexType>

```

documentation Element

The documentation element allows you to enter human-readable text comments in an XML schema.

```

<documentation source = "anyURI"
  xml:lang = "language" >
  . . .
</documentation>

```

element Element

The element element information item is used to define XML elements in your XML schemas. This element information item may contain annotation, simpleType, complexType, unique, key, and keyref element information items.

```

<element abstract = "boolean" : "false"
  block = "#all" | ( "extension" | "restriction" | "substitution" ) . . .
  default = "string"
  final = "#all" | ( "extension" | "restriction" ) . . .
  fixed = "string"
  form = "qualified" | "unqualified"
  id = "ID"
  maxOccurs = "nonNegativeInteger" | "unbounded" : "1"
  minOccurs = "nonNegativeInteger" : "1"
  name = "NCName"
  nillable = "boolean" : "false"
  ref = "QName"
  substitutionGroup = "QName"
  type = "QName" >
  . . .
</element>

```

extension Element

The extension element information item is used to extend an existing simpleType or complexType element.

```

<extension base = "QName"
  id = "ID" >
  . . .
</extension>

```

group Element

The group element defines a group of elements for use in complexType definitions.

```
<group id = "ID"
  name = "NCName"
  ref = "QName"
  maxOccurs = "nonNegativeInteger" | "unbounded" : "1"
  minOccurs = "nonNegativeInteger" : "1" >
  . . .
</group>
```

import Element

The import element imports additional schemas with a target namespace different from the document.

```
<import id = "ID"
  namespace = "anyURI"
  schemaLocation = "anyURI" >
  . . .
</import>
```

list Element

The list element information item defines a list of items that represent the value space of a simple user-defined XML Schema type. This item may contain annotation and simpleType element information items. The syntax is as follows:

```
<list id = "ID"
  itemType = "QName" >
  . . .
</list>
```

notation Element

The notation element defines the format for non-XML data within your XML document.

```
<notation id = "ID"
  name = "NCName"
  public = "anyURI"
  system = "anyURI" >
  . . .
</notation>
```

restriction Element

The restriction element information item restricts the facets of a user-defined simple type. This element information item may contain annotation, simpleType, and combinations of the following additional element information items: minExclusive, minInclusive, maxExclusive,

maxInclusive, totalDigits, fractionDigits, length, minLength, maxLength, enumeration, whiteSpace, and pattern. The applicability of many of these element information items is dependent on the base data type being restricted. For a numeric user-defined type, for instance, restrictions on whiteSpace are inapplicable. Following is the format for the restriction element information item:

```
<restriction base = "QName"
  id = "ID" >
  . . .
</restriction>
```

schema Element

The schema element information item is the root element for every XML schema. Following is the format for the schema element information item:

```
<schema attributeFormDefault = "qualified" | "unqualified" : "unqualified"
  blockDefault = "#all" | ( "extension" | "restriction" | "substitution" ) . . .
  elementFormDefault = "qualified" | "unqualified" : "unqualified"
  finalDefault = "#all" | ( "extension" | "restriction" | "list" | "union" ) . . .
  id = "ID"
  targetNamespace = "anyURI"
  version = "token"
  xml:lang = "language"
  xmlns:namespace = "anyURI" >
  . . .
</schema>
```

The schema element information item contains all other element information items. XML schemas are often declared with the `xmlns:xsd` or `xmlns:xs` namespace attribute set to the URI <http://www.w3.org/2001/XMLSchema>.

sequence Element

The sequence element specifies that its contained elements must appear in sequence, with each child element occurring zero or more times.

```
<sequence id = "ID"
  maxOccurs = "nonNegativeInteger" | "unbounded" : "1"
  minOccurs = "nonNegativeInteger" : "1" >
  . . .
</sequence>
```

simpleContent Element

The `simpleContent` element information item defines simple (character data) content with attributes. The `simpleContent` element information item can contain annotation and restriction, or extension element information items.

```
<simpleContent id = "ID" >
  . . .
</simpleContent>
```

simpleType Element

The `simpleType` element information item defines simple XML Schema user-defined types that are derived by restriction, list, or union from other simple XML Schema base types. This element information item may contain annotation and restriction, list, or union element information items. The syntax is as follows:

```
<simpleType final = "#all" | ( "list" | "union" | "restriction" ) . . .
  id = "ID"
  name = "NCName" >
  . . .
</simpleType>
```

union Element

The `union` element information item defines a user-defined data type that represents the union of a sequence of simple data types. The `union` element information item may contain annotation and `simpleType` element information items.

```
<union id = "ID"
  memberTypes = "QName-list" >
  . . .
</union>
```

XML Schema Data Type Facets

XML Schema defines several constraining facets for user-derived data types. These facets are accessible as elements within the user-derived data type declarations. Table C-1 lists the constraining facets available in XML Schema.

Table C-1. *Constraining Facet List*

Constraining Facet	Description
enumeration	Constrains the value space of a data type to a given set of values.
fractionDigits	Restricts the value space of a data type to only those values that can be represented with a given set of digits after the decimal point.
length	Specifies the size of the data type in exact units of length—either the number of characters, octets, or list items for list data types.
maxExclusive	Specifies the maximum upper bound (excluding this value) for ordered data types.
maxInclusive	Specifies the minimum lower bound (including this value) for ordered data types.
maxLength	Specifies the maximum number of characters, octets, or list items allowed.
minExclusive	Specifies the minimum lower bound (excluding this value) for ordered data types.
minInclusive	Specifies the minimum lower bound (including this value) for ordered data types.
minLength	Specifies the minimum number of characters, octets, or list items allowed.
pattern	Specifies an exact format for values of this data type, using a regular expression to restrict the values to a specific pattern.
totalDigits	Specifies the exact number of digits allowed.
whiteSpace	Specifies whether white space is normalized or preserved in values.



XQuery/XPath/XML DML Quick Reference

This chapter serves as a quick reference to XQuery, XPath, and XML DML functionality as implemented in SQL Server 2008.

XPath

The `FOR XML PATH` provides support for naming result columns using a subset of XPath-style paths, including node tests. Path expressions in XPath consist of location steps separated by forward slash (/) characters. Each step of the path represents an element or attribute in a hierarchical XML structure. Listing D-1 shows sample `FOR XML PATH` XPath-style paths.

Listing D-1. *Sample Path Expressions in FOR XML PATH*

```
@customer-number  
shipping-address/city  
address/billing-address/@postal-code
```

Path expressions used with `FOR XML PATH` must follow certain rules, including the following:

- Paths cannot begin or end with the / axis step, and cannot contain the // axis step.
- Paths are case sensitive, regardless of database collation settings.
- The @ sign is used to map a column to an XML attribute in the result.
- Consecutive columns with the same prefix are grouped together; nonconsecutive columns with the same prefix are separated in the XML result.

In addition, `FOR XML PATH` supports a set of XPath node tests, which can appear as the last location step in a path. The supported node tests are listed in Table D-1.

Table D-1. *FOR XML PATH Supported Node Tests*

Node Test	Description
text()	Adds the string value of the column to the XML result as a text node.
comment()	Adds the string value of the column to the XML result as a comment node.
node()	Adds the string value of the column inline under the row element.
element/node()	Adds the string value of the column inline under the specified element.
data()	Adds the string value of the column as an atomic value. Spaces are inserted between atomic values in the XML result.
processing-instruction(name)	Adds the string value of the column to the XML result as a processing instruction named <i>name</i> .
*	Adds the string value of the column inline under the row element.
element/*	Adds the string value of the column inline under the specified element.
@name	Adds the string value of the column as an attribute of the row element.
name	Adds the string value of the column as a subelement of the row element.
element/name	Adds the string value of the column to the XML result as a subelement in the specified element hierarchy, under the row element.
element/@name	Adds the string value of the column to the XML result as an attribute of the last element in the specified hierarchy, under the row element.

Listing D-2 demonstrates a sample FOR XML PATH query run within the AdventureWorks database.

Listing D-2. *FOR XML PATH Query Example*

```
SELECT cus.CustomerID AS "@customer-id",
       cus.AccountNumber AS "@account-number",
       con.Title AS "name/title",
       con.FirstName AS "name/first",
       con.LastName AS "name/last",
       con.EmailAddress AS "contact/email-address",
       con.Phone AS "contact/phone"
FROM Sales.Customer cus
INNER JOIN Sales.Individual ind
  ON cus.CustomerID = ind.CustomerID
INNER JOIN Person.Contact con
  ON ind.ContactID = con.ContactID
FOR XML PATH;
```

XQuery

SQL Server 2008 continues the tradition of built-in XQuery support in SQL Server, available since the SQL Server 2005 release. SQL Server supports a subset of the W3C XQuery recommendation. This section is a quick reference to the SQL Server 2008 implementation of XQuery. XQuery maps XML data to XDM instances for data-typing support. The XDM supports seven types of nodes, listed in Table D-2.

Table D-2. *XDM Node Types*

XDM Node Type	Description
Document node	Encapsulates an XML document
Element node	Represents XML elements
Attribute node	Represents XML attributes
Namespace node	Represents namespace bindings
Processing instruction node	Represents XML processing instructions
Comment node	Represents XML comments
Text node	Represents XML character content

XQuery supports XPath expressions with location paths consisting of zero or more location steps separated by forward slash (/) characters. Each location step consists of an axis specifier, a node test, and zero or more predicates enclosed in brackets ([]). A leading single forward slash (/) character in the path indicates the node tree root. Listing D-3 shows examples of valid XQuery path expressions.

Listing D-3. *Sample XQuery Path Expressions*

```
/Root/Address/State
//Contact/Name
/Movie[1]/Actors[1]
Invoice[@invoice-num eq "IN19835"]/Subtotal
```

SQL Server supports a subset of XQuery axis specifiers in path expressions. The supported axis specifiers are listed in Table D-3.

Table D-3. *SQL Server XQuery Supported Axis Specifiers*

Axis Specifier	Abbreviation
attribute::	@
child::	(default)
descendant::	n/a
descendant-or-self::	//
parent::	..
self::	.
/	

In addition, SQL Server XQuery supports the node tests listed in Table D-4.

Table D-4. *SQL Server XQuery Supported Node Tests*

Node Test
node()
text()
comment()
processing-instruction()
processing-instruction(<i>literal</i>)
<i>node-name</i>
<i>namespace-prefix:node-name</i>
*
<i>namespace-prefix:*</i>

XQuery supports several operators for performing mathematical calculations, node comparisons, value comparisons, and logical operations. All of SQL Server’s supported XQuery operators are listed in Table D-5.

Tip SQL Server does not support the XQuery recommended `idiv` operator, which performs integer division. You can simulate the `idiv` operator with the `div` and `cast` operators, using the following expression: `($arg1 div $arg2) cast as xs:integer?`.

Table D-5. *XQuery Operators*

Operator	Description
Math	
+	Addition
-	Unary minus, subtraction
*	Multiplication
div	Division
mod	Modulo arithmetic
Existential Comparison	
<=	Less than or equal
<	Less than
>=	Greater than or equal
>	Greater than
=	Equal
!=	Not equal

Operator	Description
Value Comparison	
eq	Equal
ne	Not equal
gt	Greater than
ge	Greater than or equal
lt	Less than
le	Less than or equal
Logical Operators	
and	Logical AND of two Boolean expressions
or	Logical OR of two Boolean expressions

XQuery returns results as node sequences. You can construct sequences using parentheses and the comma operator. Listing D-4 demonstrates sequence construction using parentheses and the comma operator.

Listing D-4. *Constructing Sample XQuery Sequences*

```
(1, 2, 3)
(3.14159, 2.71828, 0.8346268)
("Jackie", "Tito", "Jermaine", "Marlon", "Michael")
(<singer>Latoya</singer>, <singer>Janet</singer>, <singer>Rebbie</singer>)
(6.0, "six", "VI", "seis", "00000110", "sechs")
(6.02214179)
()
```

SQL Server supports sequences of nodes and sequences of atomic values. SQL Server does not, however, support heterogeneous sequences that combine nodes and atomic values. Trying to create a heterogeneous sequence causes a static type error in SQL Server. XQuery sequences are subject to the following rules:

- The empty sequence is represented by empty parentheses ().
- A sequence containing only one single atomic value is equal to that singleton atomic value.
- A sequence cannot contain other sequences; XQuery flattens nested sequences.

XQuery also supports powerful FLWOR expressions, which consist of the clauses listed in Table D-6.

Table D-6. *FLWOR Expression Clauses*

FLWOR Clause
for <i>\$var</i> in <i>expression</i>
let <i>\$var</i> := <i>expression</i>
where <i>boolean-expression</i>
order by <i>expression</i> [ascending descending]
return <i>expression</i>

XQuery also supports conditional expressions of the form if (*expression*₁) then *expression*₂ else *expression*₃. These expressions are equivalent to the SQL CASE expressions. If the effective Boolean value of *expression*₁ is true, then *expression*₂ is returned, otherwise *expression*₃ is returned.

SQL Server XQuery partially supports the instance of and cast as expressions. The instance of operator determines the runtime type of the value of a specified expression. The cast as operator converts a value to a specified data type. In addition, SQL Server XQuery supports the quantified expressions some and every, in the following formats: some *\$var* in *expression*, . . . satisfies *boolean-expression* and every *\$var* in *expression*, . . . satisfies *boolean-expression*.

SQL Server supports XQuery XML construction using both direct and computed constructors. The computed constructors supported are listed in Table D-7.

Table D-7. *XQuery Computed Constructors*

Computed Constructor
element
attribute
text

SQL Server XQuery supports a subset of the XQuery functions defined in the XPath/XQuery Functions and Operators Recommendation. These functions are detailed in Chapter 6.

XML DML

SQL Server supports XQuery extensions for manipulating XML, referred to as XML DML. XML DML can only be executed by the xml data type modify() method. The XML DML statements are summarized in Table D-8. The SQL Server 2008 implementation of XML DML is described in detail in Chapter 6.

Table D-8. *XML DML Statements*

Statement
insert
delete
replace value of



XSLT 1.0 and XPath 1.0 Reference

This chapter serves as a quick reference to Extensible Stylesheet Language Transformations (XSLT) and XPath 1.0, which I talked about in Chapter 8. The .NET 2.0 `XslCompiledTransform` class supports XSLT 1.0. Here I'll provide a quick reference to the elements and functions supported by XSLT. Table E-1 lists the available XSLT 1.0 elements.

Table E-1. *XSLT Elements*

Element	Attributes	Description
<code>xsl:apply-imports</code>	None	Applies a template rule from an imported stylesheet.
<code>xsl:apply-templates</code>	<code>select</code> , <code>mode</code>	Applies a template rule to the current element or to the current element's child nodes.
<code>xsl:attribute</code>	<code>name</code> , <code>namespace</code>	Adds an attribute to an element.
<code>xsl:attribute-set</code>	<code>name</code> , <code>use-attribute-sets</code>	Creates a named set of XML attributes.
<code>xsl:call-template</code>	<code>name</code>	Calls a named template.
<code>xsl:choose</code>	None	Contains multicondition tests.
<code>xsl:comment</code>	None	Creates an XML comment node.
<code>xsl:copy</code>	<code>use-attribute-sets</code>	Creates a shallow copy of the current element, without copying child nodes or attributes.
<code>xsl:copy-of</code>	<code>select</code>	Creates a deep copy of the specified element, including attributes, namespace nodes, and child nodes.

Continued

Table E-1. *Continued*

Element	Attributes	Description
xsl:decimal-format	name, decimal-separator, grouping-separator, infinity, minus-sign, NaN, percent, per-mille, zero-digit, digit, pattern-separator	Defines format when converting numbers to strings with the format-number function.
xsl:element	name, namespace, use-attribute-sets	Creates an XML element.
xsl:fallback	None	Specifies alternate code to execute for unsupported XSLT elements.
xsl:for-each	select	Loops through the specified node set.
xsl:if	test	Performs a logical test and returns the contents of the xsl:if only if the effective Boolean value of the test is true.
xsl:import	href	Imports a stylesheet into another stylesheet, with lower precedence than the importing stylesheet.
xsl:include	href	Includes a stylesheet in another stylesheet, giving the included stylesheet the same precedence as the including stylesheet.
xsl:key	name, match, use	Declares a named key that can be used with the key function in a stylesheet.
xsl:message	terminate	Writes a message to the output.
xsl:namespace-alias	stylesheet-prefix, result-prefix	Aliases a namespace in the input with a different namespace in the output.
xsl:number	count, level, from, value, format, lang, letter-value, grouping-separator, grouping-size	Inserts a formatted number in the output.
xsl:otherwise	None	Specifies a default action for xsl:choose constructs.
xsl:output	method, version, encoding, omit-xml-declaration, standalone, doctype-public, doctype-system, cdata-section-elements, indent, media-type	Defines the format of the output document.
xsl:param	name, select	Declares a global stylesheet or local template parameter.

Element	Attributes	Description
xsl:preserve-space	elements	Defines a list of elements for which extraneous white space should be preserved.
xsl:processing-instruction	name	Writes an XML processing instruction to the output.
xsl:sort	select, lang, data-type, order, case-order	Sorts the output of xsl:for-each or xsl:apply-templates.
xsl:strip-space	elements	Defines a list of elements for which extraneous white space should be removed.
xsl:stylesheet, xsl:transform	version, extension-element-prefixes, exclude-result-prefixes, id	The root element of a style sheet.
xsl:template	name, mode, match, priority	Defines a template to be instantiated based on specified template rules.
xsl:text	disable-output-escaping	Writes a text node to the output.
xsl:value-of	select, disable-output-escaping	Retrieves the value of a selected node or expression and includes it in the output.
xsl:variable	name, select	Sets a global stylesheet or local template variable. Note that an XSLT variable is analogous to constants in other languages, since its value cannot be changed once it is set.
xsl:when	test	Specifies an expression and one path of action in a multicondition xsl:choose element.
xsl:with-param	name, select	Defines the value of a parameter to pass into a template. Must have a matching xsl:param element in the called template.

The .NET `XslCompiledTransform` object supports XSLT 1.0, which uses the XPath 1.0 functions and some additional XSLT 1.0–specific functions. Table E-2 lists the XPath 1.0 and XSLT 1.0 functions.

Note XSLT 2.0 supports XPath 2.0 functions and operators, which is the same library of functions supported by XQuery 1.0.

Table E-2. *XPath 1.0 and XSLT 1.0 Functions*

Function	Description
Boolean Functions	
<code>boolean(object)</code>	Returns the effective Boolean value of its argument.
<code>false()</code>	Returns false.
<code>lang(string)</code>	Returns true if the <code>xml:lang</code> attribute of the context node is the language, or is a sublanguage, of the language specified by the function parameter.
<code>not(object)</code>	Returns true if its argument is false, and returns false otherwise.
<code>true()</code>	Returns true.
Node Set Functions	
<code>count(node-set)</code>	Returns the number of nodes in the node set passed as a parameter to the function.
<code>id(object)</code>	Selects a node by its unique ID.
<code>last()</code>	Returns a number equal to the context size of the expression evaluation context. Simply put, this returns the number of nodes, or the number of the last node, in the current node set. Note that the first node is always 1.
<code>local-name(node-set), local-name()</code>	Returns the local name of the specified node set or the context node.
<code>name(node-set), name()</code>	Returns the expanded QName of the first node in the node set, or of the context node.
<code>namespace-uri(node-set), namespace-uri()</code>	Returns the namespace URI of the specified node set or the context node.
<code>position()</code>	Returns a number equal to the current context node position. This is the number of the context node in the current node set.
Number Functions	
<code>ceiling(number)</code>	Returns the smallest integer that is not less than the argument.
<code>floor(number)</code>	Returns the largest integer that is not greater than the argument.
<code>number(object), number()</code>	Converts its argument, or the context node, to a number.
<code>round(number)</code>	Returns the integer that is closest to the argument.
<code>sum(node-set)</code>	Converts the nodes in its arguments to numbers and returns the sum of those numbers.
String Functions	
<code>concat(string, string, . . .)</code>	Concatenates two or more strings together into a single string.
<code>contains(string₁, string₂)</code>	Returns true if <code>string₁</code> contains <code>string₂</code> .
<code>normalize-space(string), normalize-space()</code>	Returns the whitespace-normalized version of the string parameter, or of the current context node converted to a string if no string is specified. The whitespace-normalized version strips leading and trailing spaces and collapses multiple consecutive spaces into a single space.
<code>starts-with(string₁, string₂)</code>	Returns true if <code>string₁</code> starts with <code>string₂</code> .
<code>string(object), string()</code>	Converts the specified argument, or the context node, to a string.

Function	Description
<code>string-length(<i>string</i>),</code> <code>string-length()</code>	Returns the number of characters in the string specified, or in the context node converted to a string if no string is specified.
<code>substring(<i>string</i>, <i>start</i>),</code> <code>substring(<i>string</i>, <i>start</i>, <i>length</i>)</code>	Returns the portion of the string specified starting at the given <i>start</i> position and with the specified <i>length</i> . If the <i>length</i> parameter is omitted, the substring is returned from the <i>start</i> position to the end of the string.
<code>substring-after(<i>string</i>₁, <i>string</i>₂)</code>	Returns the portion of <i>string</i> ₁ after the first occurrence of <i>string</i> ₂ is encountered. If <i>string</i> ₂ does not occur in <i>string</i> ₁ , the empty string is returned.
<code>substring-before(<i>string</i>₁, <i>string</i>₂)</code>	Returns the portion of <i>string</i> ₁ before the first occurrence of <i>string</i> ₂ is encountered. If <i>string</i> ₂ does not occur in <i>string</i> ₁ , the empty string is returned.
<code>translate(<i>string</i>₁, <i>string</i>₂, <i>string</i>₃)</code>	The translate function maps characters in <i>string</i> ₁ to the characters in <i>string</i> ₂ , which are translated to characters in <i>string</i> ₃ . This is useful for converting characters to upper- or lowercase in some situations. For instance, <code>translate("abcdef", "ace", "ACE")</code> returns the result "AbCdEf".
XSLT 1.0 Functions	
<code>current()</code>	Returns the current node.
<code>document(<i>object</i>, <i>node-set</i>),</code> <code>document(<i>object</i>)</code>	Allows access to XML documents other than the current source document. If the first argument is a node set, the result is the union of the node set with the current source document. If the first argument is not a node set, it is converted to a string and treated as a URI.
<code>element-available(<i>string</i>)</code>	Returns true if the specified XSLT element is supported by the XSLT processor.
<code>format-number(<i>number</i>, <i>string</i>₁),</code> <code>format-number(<i>number</i>, <i>string</i>₁, <i>string</i>₂)</code>	Converts a number to a formatted string using the format specified by <i>string</i> ₁ and the decimal-format of <i>string</i> ₂ , if specified.
<code>function-available(<i>string</i>)</code>	Returns true if the specified XSLT or XPath function is supported by the XSLT processor.
<code>generate-id(<i>node-set</i>),</code> <code>generate-id()</code>	Returns a string value that uniquely identifies the first node in the node set, or the context node if no node set is specified.
<code>key(<i>string</i>, <i>object</i>)</code>	Returns a node set indexed by <code>xsl:key</code> elements.
<code>system-property(<i>string</i>)</code>	Returns a system property value specified by a QName. Required system properties include <code>xsl:version</code> , <code>xsl:vendor</code> , and <code>xsl:vendor-url</code> .
<code>unparsed-entity-uri(<i>string</i>)</code>	Returns the URI of the unparsed entity.

XPath 1.0 supports axis specifiers, which are used in XPath expressions. The supported axis specifiers are listed in Table E-3.

Table E-3. *XPath 1.0 Axis Specifiers*

Axis Specifier	Description
ancestor	Contains the parent of the context node, the parent's parent, and so on up to the root node.
ancestor-or-self	Contains the context node and all its ancestor nodes up to the root node.
attribute	Contains the attributes of the context node. This axis can be abbreviated with the at sign (@).
child	Contains the children of the context node. This axis is the default if no axis specifier is used.
descendant	Contains the children of the context node, the child's child, and so on.
descendant-or-self	Contains the context node and its descendants. This can be abbreviated as //.
following	Contains all nodes after the context node, in document order, excluding descendants of the context node.
following-sibling	Contains all sibling nodes after the context node, in document order.
namespace	Contains the namespace nodes of the context node.
parent	Contains the parent of the context node. This can be specified using the .. abbreviation.
preceding	Contains all nodes that are before the context node in document order, excluding ancestors of the context node.
preceding-sibling	Contains all sibling nodes before the context node, in document order.
self	Contains the current context node. This can be specified using the period (.) abbreviation.
/	The / operator at the beginning of a path expression specifies the root node of the document.

In XPath 1.0, every node has a principal type. XPath 1.0 supports the node tests listed in Table E-4 to determine the type of a node.

Table E-4. *XPath 1.0 Node Tests*

Node Test	Description
comment()	Returns true for any comment node.
node()	Returns true for any type of node.
processing-instruction()	Returns true for any processing instruction node.
processing-instruction(<i>name</i>)	Returns true for a processing instruction node with the specified name.
text()	Returns true for any text node.

The XPath 1.0 standard supports a selection of comparison operators for use in predicates, logical operators for forming compound predicates, and mathematical operators. All of these operators are listed in Table E-5.

Table E-5. *XPath Operators*

Operator	Description
Comparison Operators	
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal
!=	Not equal
Logical Operators	
and	Logical AND of the result of two expressions; both must be true to return true.
or	Logical OR of the result of two expressions; both must be false to return false.
Math Operators	
+	Addition
-	Subtraction, unary minus
*	Multiplication
div	Division
mod	Modulus

The XPath 1.0/XSLT 1.0 data model is significantly simpler than the XQuery 1.0/XPath 2.0 data model. XSLT 1.0 supports seven types of nodes and four data structures, all shown in Table E-6.

Table E-6. *XPath 1.0 Data Types/Node Types*

Type	Description
Data Types	
boolean	A Boolean value indicated by true or false.
number	An integer or floating point numeric value.
string	A character string.
node set	A set of nodes selected by an XPath expression.
Node Types	
Root	Maps to document root node.
Element	Maps to XML element nodes.
Attribute	Maps to XML attribute nodes.
Namespace	Maps to XML namespace nodes.
Processing Instruction	Maps to XML processing instruction nodes.
Comment	Maps to XML comment nodes.
Text	Maps to XML character data.



Glossary

Throughout this book, I have used several terms that are specific to XML and related W3C recommendations. This appendix serves as a reference for XML-specific definitions.

Annotation

XML Schema annotations are a built-in mechanism for including both human-readable and application-readable documentation in an XML schema.

Atomic Data Types, List Data Types, and Union Data Types

Atomic data types are indivisible data types that derive from the `xs:anyAtomicType` type. Examples include `xs:boolean`, `xs:date`, and `xs:integer`. List data types are types that are constructed of sequences of other types. Union data types are constructed from the *ordered union* of two or more data types or a restricted subset of a data type. The “XML Schema 1.1, Part 2: Datatypes” specification working draft (www.w3.org/TR/xmlschema11-2/#ordinary-built-ins) defines no built-in union data types.

Attribute

Attributes are used in XML to provide metadata, or additional information, about an XML element. Attributes hold a special position in XML, as they do not technically have parent elements in the XML hierarchy, but they are generally associated with particular XML elements. See also *Attribute Nodes*.

Attribute Nodes

Attribute nodes are the XDM equivalent of XML attributes. Because of their special definition in the XML recommendation, attribute nodes do not technically have parent nodes. However, attribute nodes are generally associated with corresponding element nodes. See also *Attribute*.

Axis

An axis specifier indicates the relationship between the nodes selected by the location step and the context node. Examples of axis specifiers include `child`, `parent`, and `ancestor`.

Axis Step

An axis step consists of an optional axis specifying movement direction, a node test specifying criteria to select nodes, and zero or more predicates to filter results. The result of an axis step is a sequence of zero or more nodes.

Base64

Base64 is the base of a counting system with 64 individual digits. Base64 is the largest power-of-two base that can be represented with printable UTF-7 characters, and it has been adopted for representation of binary data in XML Schema. XML Schema uses the IETF RFC 2045 definition Base64 system, which uses the letters a–z, A–Z, 0–9, the plus sign (+), the forward slash (/), and the equal sign (=). RFC 2045 is available at www.ietf.org/rfc/rfc2045.txt.

Character Reference

Character references in XML come in two flavors: character entity references and numeric character references. XML supports five built-in character entity references, such as & for the ampersand (&) character. Numeric character references take the form &#*n*; where *n* is a number indicating a Unicode character code point, or &#x*n*; with *n* representing a Unicode character code point in hexadecimal format. Character references are expanded by the XML parser when nodes are accessed.

Closed Language

A closed language is one in which every single possible expression is guaranteed to be represented by the underlying data model. XQuery 1.0, XPath 2.0, and XSLT 2.0 are considered closed with respect to XDM.

Comma Operator

The XQuery comma operator is used to combine items into a sequence.

Comments

XQuery comments are denoted by the (: and :) delimiters in XQuery queries. XQuery comments are ignored during processing and should not be confused with XML *comment nodes*.

Comment Nodes

XML comment nodes are denoted using the <!-- and --> delimiters in XML data. Comment nodes are not ignored and can be retrieved by XML parsers during processing. Comment nodes should not be used to send instructions to applications, however; for that purpose, use XML processing instructions.

Complex Type

Complex types in XML Schema are types that define the structure of an element including valid attributes, children, and content of the element.

Compound Predicate

Compound predicates result from combining two or more comparison predicates with the *and* and *or* keywords.

Computed Element Constructor

A computed element constructor is an XQuery mechanism for constructing XML by using element constructors that begin with a keyword to identify the type of node to create, the name of the node to create, and the content of the node.

Context Item Expression

In XQuery, this expression evaluates to the context item. Compare *Context Node*.

Context Node

The context node is the node currently being processed. Technically, the context node is the context item when the item is a node. Each node returned by a step in a location path is used in turn as the context node. Subsequent steps define their axes in relation to the current context node. For instance, with the sample XPath expression `/Root/Person/Address`, the Root node is the first context node. All Person nodes returned below Root become the context node in turn, and the Address nodes are retrieved relative to these context nodes, each becoming the context node in turn.

CSV

CSV is the Comma-Separated Values plain text file format. This format was popularized in the mid-1990s with the emergence of popular spreadsheet programs, and it was semi-standardized after that. There is no official CSV standard, and CSV files are often not portable across platforms or even across different applications on the same platform.

Declaration

Declarations are defined in DTDs, and they are automatically expanded by conformant XML parsers.

Direct Element Constructor

An XQuery direct element constructor is a mechanism for constructing XML using a specification in standard XML notation.

Document

A document in XML is XML data that conforms to the following rules for well-formedness: 1) it must have a single root element, 2) all elements must be properly nested, and 3) it must be composed of valid characters and well-formed entities. See also *Document Node*, *Root Node*.

Document Node

Document nodes are defined by the XDM as nodes that encapsulate XML documents. A document node is roughly analogous to an XML root node; however, XQuery can accept as input, and generate as output, XML data that is not well-formed, with multiple document nodes. See also *Document*, *Root Node*.

Document Order

XQuery returns most results in document order by default. Document order is analogous to the order in which nodes appear in the XML data being queried, or to the order items are placed in a sequence when the sequence is created.

DTD

DTDs, or Document Type Definitions, are defined as a standard mechanism for defining the structure of, and constraining the textual content of, XML documents. SQL Server supports a very limited subset of DTDs. To define XML document structure and constrain content, use more powerful and flexible XML schemas instead.

EBCDIC

Extended Binary Coded Decimal Interchange Code is an 8-bit character encoding, commonly used on IBM mainframe systems.

Element

In XML, an element is a logical structure in XML that is delimited by a start tag and an end tag, or by the empty-element tag. An element may have associated attributes and may contain data including additional nested subelements.

Element Nodes

Element nodes are the XQuery equivalent representation of XML elements. See also *Element*.

Empty Sequence

This is an XPath 2.0 and XQuery 1.0 sequence containing zero items. The XQuery notation for an empty sequence is a set of parentheses: ().

Enclosed Expression

An enclosed expression is an expression enclosed in curly braces ({}) in an XQuery element constructor. The enclosed expression is evaluated by the XQuery processor during element construction.

Entity

An entity is an XML construct that can be expanded to a character or sequence of characters by an XML parser. XML has some built-in character entity references, and allows user-created entity declarations within the DTD.

Expression

Expressions are the building blocks of XQuery queries, and XQuery supports several types of expressions including primary expressions, path expressions, mathematical expressions, FLWOR expressions, and others.

External DTD

An external DTD is one that is stored separately from the XML document(s) that use it, often in the local file system or on a network. The SQL Server xml data type does not support external DTDs.

F&O

F&O is used to reference XQuery functions and operators as defined by the W3C XQuery 1.0 and XPath 2.0 Functions and Operators Recommendation, available at www.w3.org/TR/xquery-operators.

Facets

Facets are schema components used to constrain data types. A couple of commonly used facets are `whiteSpace` and `length`, which control how white space in string values is handled and restrict values to a specific number of units in length, respectively.

Filter Expression

This is a primary expression followed by zero or more predicates.

FLWOR Expression

FLWOR is an acronym for the XQuery keywords that are used to create them: `for`, `let`, `where`, `order by`, and `return`. FLWOR expressions support iteration and variable binding.

FOR XML Clause

The `FOR XML` clause is used in SQL Server `SELECT` statements to generate XML from relational data.

Fragment

An XML document fragment is a portion of an XML document. It generally must follow the same rules for well-formedness as an XML document, except that it can have more than a single root element. See also *Document*.

General Comparisons

These are existentially quantified XQuery comparisons that may be applied to operand sequences of any length. In general comparisons, the nodes are atomized and the atomic values of both operands are compared using value comparisons. If any of the value comparisons evaluate to true, the result is true.

General Comparison Operators

The XQuery general comparison operators are `=`, `<`, `>`, `<=`, `>=`, and `!=`.

Heterogeneous Sequence

A heterogeneous sequence in XQuery is a sequence that contains both nodes and singleton atomic values. SQL Server XQuery does not allow heterogeneous sequences and will return an error if you try to create one.

HTTP SOAP

See *SOAP*.

Inline DTD

An inline DTD is one that is included as part of an XML document. See also *External DTD*.

Infoset

An XML Infoset is a nonliteral representation of XML data that conforms to the W3C XML Information Set standard. The XML Infoset removes extraneous white space, expands character and entity references, boundary information, and other information from the XML representation. See also *PSVI*, *XDM*.

ISO SQL:2003 Standard

The ISO SQL standard is the official standard for SQL. ISO SQL:2003 is the standard approved in 2003. This standard is particularly significant because it officially recognized the importance of XML support in relational databases by incorporating the SQL/XML standard into SQL. The official designation for this standard is ISO/IEC 9075-*n*:2003, where *n* is a designated portion of the standard. See also *ISO SQL/XML*.

ISO SQL/XML

The ISO SQL/XML standard is the official standard for XML integration with SQL. This standard was added to the ISO SQL:2003 standard. See also *ISO SQL:2003 Standard*.

LOB

LOB is an abbreviation for Large Object, in reference to SQL Server data types that can hold very large amounts of data. The LOB data types include the deprecated `text`, `ntext`, and `image` data types, as well as the `varchar(max)`, `nvarchar(max)`, `varbinary(max)`, and `xml` data types. LOB data types can hold up to 2.1 gigabytes of data.

Location Paths

A path is an XPath or XQuery expression that addresses a specific subset of nodes in an XML document. The location path is a series of steps separated by the solidus (forward slash) character, evaluated from left to right. Each step generates a sequence of items. Location paths can be relative or absolute. *Absolute* location paths begin with a single solidus character, but *relative* location paths do not.

Namespace

XML supports namespaces, which help avoid naming conflicts in XML data. XML namespaces are prepended to element and attribute names, which allows reuse of these names in XML data. See also *Namespace Nodes*, *Namespace Prefix*.

Namespace Nodes

Each XML element has an associated set of namespace nodes, which define all namespaces that are in scope for that element. See also *Namespace*, *Namespace Prefix*.

Namespace Prefix

Each XML namespace is composed of two parts, a namespace prefix and a namespace URI. The XML namespace prefix can be thought of as the “friendly name” for a namespace. For instance, the predeclared namespace prefix `xmlns` is bound to the namespace URI `http://www.w3.org/2000/xmlns`. See also *Namespace*, *Namespace Nodes*.

Node

XPath 2.0 and XQuery 1.0 treat XML data as a hierarchical tree structure, similar to (but not exactly the same as) the Document Object Model (DOM) that web programmers often use to manipulate HTML and XML. XPath and XQuery XML trees are composed of the seven types of nodes defined in the W3C XQuery 1.0 and XPath 2.0 Data Model (XDM), full descriptions of which are available at www.w3.org/TR/xpath-datamodel/#node-identity. These node types include the following:

- Attribute nodes, which represent XML attributes
- Comment nodes, which encapsulate XML comments
- Document nodes, which encapsulate XML documents
- Element nodes, which encapsulate XML elements
- Namespace nodes, which represent the binding of a namespace URI to a namespace prefix (or the default namespace)
- Processing instruction nodes, which encapsulate processing instructions (PIs)
- Text nodes, which encapsulate XML character content

XPath 1.0 defines the node types it uses in Part 5 of the XPath 1.0 specification. The main difference between XPath 1.0 nodes and XDM nodes is that XPath 1.0 defines the *root node* of a document in place of the *document nodes* of the XDM. Another major difference is that in the XDM, element nodes are either explicitly or implicitly (based on content) assigned type information.

Node Comparison

Node comparisons in XQuery compare nodes by their document order or identity.

Node Comparison Operators

XQuery provides three node comparison operators (`<<`, `>>`, and `is`), which test nodes for order (preceding, following) and node identity equality, respectively.

Node Test

A node test is a condition that must be true for each node generated by a step. A node test can be based on the name of the node, the kind of node, or the type of node.

Optional Occurrence Indicator

The question mark (?) character, when used in conjunction with the `cast` as keywords, is referred to as the optional occurrence indicator. It indicates that the empty sequence is allowed.

Path Expression

See *Location Paths*.

Predicate

A predicate is an expression enclosed in brackets ([]) that is used to filter a sequence. The predicate expressions are generally comparison expressions of some sort (equality, inequality, etc.).

Primary Expression

This is the basic *primitive* of the XQuery language. A primary expression can be a literal, a variable reference, a context item expression, a data type constructor, or a function call.

Processing Instruction

Processing instructions are XML-defined mechanisms for passing instructions to processing applications inside of XML data. Processing instructions provide a tool to help improve automated processing of XML data. See also *Processing Instruction Nodes*.

Processing Instruction Nodes

XQuery processing instruction nodes are the XQuery equivalent of XML processing instructions. See also *Processing Instruction*.

Prolog

The XQuery language supports prologs, which are namespace declarations and definitions positioned before the body of an XQuery expression.

PSVI

The Post-Schema-Validation Infoset (PSVI) is an XML Infoset that is produced after XML Schema validation. The PSVI is an XML Infoset augmented with additional XML Schema type information. See *Infoset*, *XDM*.

Reverse Step Indicator

See *Axis Step*.

Root Node

A well-formed XML document must have a single top-level element, the so-called root element or root node. See also *Document Node*.

Schema Collection

An XML schema collection is a SQL Server collection of XML schema documents. Associating `xml` data type variables or columns with an XML schema collection creates a typed `xml` instance.

Sequences

XPath 2.0 and XQuery 1.0 define sequences as ordered collections of zero or more items. The term *ordered* is important here, as it differentiates a sequence from a *set*, which, as most T-SQL programmers know (or quickly come to realize), is unordered. XPath 1.0 defined its results in terms of *node sets*, which are unordered and cannot contain duplicates. XQuery changes this terminology to *node sequences*, which recognize the importance of node order in XML and can contain duplicates.

SGML

Standard Generalized Markup Language is a metalanguage used to design markup languages for documents. XML is based on SGML, and the XML working group and SGML committee worked closely to maintain compatibility between both markup languages. SGML is defined by the ISO 8879 standard.

Shredding

Shredding is the process of converting XML data to relational style rows and columns. SQL Server performs a shredding process implicitly when querying XML data and explicitly when using the `nodes()` method of the `xml` data type. You can also use XML indexes to persist a preshredded version of your XML data for better performance.

Simple Content

Simple content is defined by XML Schema as content that does not contain child XML elements.

Simple Type

Simple types are XML Schema data types that contain simple content. See also *Simple Content*.

SOAP

Simple Object Access Protocol is an XML-based protocol designed for exchanging structured information in distributed, decentralized environments. Because it is XML-based, SOAP works well with, and is often used in conjunction with, HTTP as a transmission protocol.

SQL:2003

See *ISO SQL:2003 Standard*.

SQL/XML

See *ISO SQL/XML*.

SQLCLR

SQL Server provides Common Language Runtime support via SQLCLR integration. SQLCLR provides additional support for XML through the standard .NET Framework XML classes.

Step

A step in XQuery is composed of an axis, a node test, and zero or more predicates. Each step is a part of a path expression that generates a sequence of items and then filters the sequence.

Template

XSLT stylesheets are based on templates, which are roughly analogous to functions or procedures in other languages. A template in XSLT is a functional block of code that allows you to break up your code into manageable sections.

Text Nodes

Text nodes are the XQuery representation of XML character data.

Unicode

Unicode provides standardization for the representation of a wide range of international character data. XML provides default support for the Unicode character encoding system.

Universal Table Format

Universal Table Format is a standard format used to return data by the `FOR XML EXPLICIT` clause.

URI

Universal Resource Identifier is a string of characters used to identify or name a resource, such as a web page or local system resource. XML namespaces associate a namespace prefix with a URI.

UTF-16

UTF-16 (Unicode Transformation Format 16) is a variable-length character encoding for Unicode. UTF-16 is one of the standard character encodings that XML supports by default.

UTF-8

UTF-8 is one of the standard character encodings that XML supports by default.

Valid

A valid XML document is one that conforms to semantic rules defined by an XML DTD or XML schema.

Value Comparison

This is a comparison of singleton atomic values in XQuery.

Value Comparison Operators

XQuery provides several value comparison operators, such as `eq` for equality and `gt` for greater than.

Well-Formed

A well-formed XML document is one that conforms to the basic XML rules for well-formedness. To be well-formed, an XML document must have one and only one root node, have all special characters entitized or properly encapsulated in a CDATA section, and have all elements properly nested.

W3C

The World Wide Web Consortium is a standards body with the stated mission of “developing interoperable technologies. . .to lead the Web to its full potential.”

XDM

The XQuery 1.0 and XPath 2.0 Data Model is defined by the W3C at www.w3.org/TR/2006/PR-xpath-datamodel-20061121. The XDM extends the XML Schema type system with additional data types. XML Schema PSVI instances map directly to, and can be converted to, XDM instances. The additional type information and validation of XDM instances provide for highly optimized XML querying and manipulation. See *Infoset*, *PSVI*.

XML

Extensible Markup Language is a restricted form of SGML designed to be easily served, received, and processed on the Web.

XML DML

XML Data Manipulation Language is a set of Microsoft SQL Server-specific extensions to XQuery that are used to modify XML data. XML DML can be used with the SQL `UPDATE` or `SET` statements and the `xml` data type `modify()` method.

XML Index

SQL Server provides a mechanism for indexing XML content stored in `xml` data type columns, known as an XML index. The XML index is a mechanism for preshredding, or preconverting, XML data to relational format to increase XQuery query efficiency.

XML Schema

Part 2 of the W3C XML Schema 1.1 Recommendation defines XML Schema data types, which are the basic data types utilized by XQuery.

XPath

XML Path Language is an expression language designed to allow processing of values that conform to the XDM.

XQuery

XML Query Language is a declarative language based on XPath. XQuery is designed to query, retrieve, and interpret data from diverse XML sources.

XSL

Extensible Stylesheet Language is a language for expressing stylesheets, consisting of a language for transforming XML documents and an XML vocabulary for specifying formatting semantics.

XSLT

Extensible Stylesheet Language Transformations (or XSL Transformations) is a language for transforming XML documents into other XML documents. For instance, XSLT can be used to transform an XML document into an XHTML document.



Selected T-SQL and .NET Code Listings

With this book, I've introduced a wide variety of sample code listings. My intent with this appendix is to reproduce some of the code listings that introduce key concepts, with additional documentation.

Chapter 1

Chapter 1 included sample listings of data in various formats, including XML, for comparison purposes. Listing 1-3 introduced a simple hierarchical XML structure for the high-grossing movie listing. This listing introduced several XML node types including elements, attributes, processing instructions, comments, and text nodes.

Listing 1-3. *Modified Sample XML*

```
<?xml version = "1.0" encoding = "UTF-16"?>
<!-- High-grossing movie listing -->
<movies>
  <film>
    <?style superhero?>
    <name>Spider-Man</name>
    <releaseDate>2002-05-03-05:00</releaseDate>
    <gross area="world-wide">821706375.00</gross>
    <gross area="domestic">403706375.00</gross>
    <director>Sam Raimi</director>
    <cast>
      <actor actor_id="MA011">Maguire, Tobey
        <movie>Spider-Man 3</movie>
        <movie>The Good German</movie>
        <movie>Spider-Man 2</movie>
        <movie>Seabiscuit</movie>
      </actor>
      <actor actor_id="DA982">Dafoe, Willem
        <movie>Clear and Present Danger</movie>
```

```

        <movie>Mississippi Burning</movie>
        <movie>Platoon</movie>
    </actor>
    <actor actor_id="DU208">Dunst, Kirsten
        <movie>Spider-Man 3</movie>
        <movie>Interview with the Vampire</movie>
    </actor>
</cast>
</film>
</movies>

```

Chapter 2

In Chapter 2, I introduced legacy SQL Server XML functionality, including the FOR XML clause of SELECT queries and the OPENXML and OPENROWSET rowset providers. Listing 2-1 introduced a simple FOR XML PATH query to return identifying information for a couple of AdventureWorks employees in XML format. This listing introduced the basic format of the FOR XML query, the ELEMENTS XSINIL option for handling nulls, and the basic format for aliasing column names with XPath location paths.

Listing 2-1. Sample FOR XML PATH Query

```

USE AdventureWorks;
GO

-- This query is a simple FOR XML PATH query to return employee
-- information for a couple of AdventureWorks employees. Notice
-- that the columns are aliased with XPath-style names that indicate
-- the hierarchical structure of the result.

SELECT emp.NationalIDNumber AS "Employee/@ID",
       emp.HireDate AS "Employee/Hire-Date",
       per.LastName AS "Employee/Name/Last",
       per.FirstName AS "Employee/Name/First",
       per.MiddleName AS "Employee/Name/Middle"
FROM HumanResources.Employee emp
INNER JOIN Person.Person per
    ON emp.BusinessEntityID = per.BusinessEntityID
WHERE emp.BusinessEntityID IN ( 2, 3 )
FOR XML PATH,
       ELEMENTS XSINIL;
GO

```

Listing 2-6 demonstrated FOR XML RAW mode with some additional options. In this example, the row node name is specified in parentheses following the FOR XML RAW clause, the ROOT option is used to indicate the creation of a root node, the ELEMENTS option is used to create the result in element-centric format, and the XMLSCHEMA option generates an inline XML schema.

The default name for the XML node representing each row of the result is `row`. Specifying the row node name in the `FOR XML RAW` clause overrides this default behavior. By default, `FOR XML` does not create a root node, behavior which the `ROOT` option overrides. Also by default, `FOR XML` creates its output in attribute-centric format, which is overridden by the `ELEMENTS` option in the example. Also by default, no XML schema is included in the result.

Listing 2-6. *Sample FOR XML RAW Query*

```
USE AdventureWorks;
GO

-- This FOR XML RAW query introduces several of the FOR XML
-- formatting options that override the default behavior of
-- FOR XML, including the ROOT option, row naming option,
-- ELEMENTS option, and XMLSCHEMA option.

SELECT d.DepartmentID,
       d.Name,
       d.GroupName
FROM HumanResources.Department d
WHERE d.DepartmentID IN ( 7, 8 )
      FOR XML RAW ('MyNode'),
      ROOT('TheRootNode'),
      ELEMENTS,
      XMLSCHEMA;
GO
```

Listing 2-10 provided a demonstration of a multi-table `FOR XML AUTO` query. The result of `FOR XML AUTO` is notable for its automatic node naming algorithm, based on the table names used in the join.

Listing 2-10. *Joined Tables FOR XML AUTO Query*

```
USE AdventureWorks;
GO

-- This query demonstrates FOR XML AUTO when used on a
-- multi-table joined result set. The element names are
-- created based on the names/aliases of the tables when
-- FOR XML AUTO is used.

SELECT dep.DepartmentID,
       dep.Name,
       emp.BusinessEntityID
FROM HumanResources.Department dep
INNER JOIN HumanResources.EmployeeDepartmentHistory emp
      ON dep.DepartmentID = emp.DepartmentID
WHERE emp.BusinessEntityID BETWEEN 20 AND 22
```

```
FOR XML AUTO;
GO
```

Listing 2-12 gave an example of the `FOR XML EXPLICIT` clause in action. `FOR XML EXPLICIT` is a complex format that requires two queries unioned together, and “bang” notation column name aliases. `FOR XML EXPLICIT` performs a similar function as the simpler and more powerful `FOR XML PATH` syntax.

Listing 2-12. *FOR XML EXPLICIT Sample Query*

```
SELECT 1 AS Tag,
      NULL AS Parent,
      p.ProductID AS [Product!1!ID],
      p.Name AS [Product!1!Name],
      p.ProductNumber AS [Product!1!Number],
      NULL AS [Quantity!2]
FROM Production.Product p
WHERE p.ProductID IN ( 770, 772 )
UNION ALL
SELECT 2 AS Tag,
      1 AS Parent,
      p.ProductID,
      p.Name,
      p.ProductNumber,
      pi.Quantity
FROM Production.ProductInventory pi
INNER JOIN Production.Product p
      ON p.ProductID = pi.ProductID
WHERE p.ProductID IN ( 770, 772 )
ORDER BY [Product!1!ID], [Product!1!Number], [Quantity!2]
FOR XML EXPLICIT;
GO
```

`FOR XML PATH` supports XPath node tests in column names that provide additional control over the generated XML result. Listing 2-13 demonstrates the use of XPath node tests in `FOR XML PATH`.

Listing 2-13. *FOR XML PATH XPath Node Tests*

```
USE AdventureWorks;
GO

-- The FOR XML PATH node tests used in this sample offer
-- greater control over the format of the resulting XML.

SELECT d.DepartmentID AS "Department/@ID",
      d.Name AS "Department/Name",
      ',' AS "Department/Name/text()",
      d.GroupName AS "Department/Name/text()"
```

```
FROM HumanResources.Department d
WHERE d.DepartmentID IN (1, 2)
    FOR XML PATH;
GO
```

Namespaces are an integral part of XML. Listing 2-18 demonstrated the use of the WITH XMLNAMESPACES clause in conjunction with the FOR XML PATH clause.

Listing 2-18. *FOR XML Clause and WITH XMLNAMESPACES*

```
USE AdventureWorks;
GO

-- This query demonstrates the WITH XMLNAMESPACES clause to
-- specify the default XML namespace to be used by the FOR
-- XML clause

WITH XMLNAMESPACES(DEFAULT 'http://www.microsoft.com/AdventureWorksDB/Product')
SELECT p.ProductID AS "Product/@ID",
    p.Name AS "Product/Name",
    p.ProductNumber AS "Product/Number",
    p.Size AS "Product/Size/data()",
    p.SizeUnitMeasureCode AS "Product/Size/data()"
FROM Production.Product p
WHERE p.ProductID = 775
    FOR XML PATH;
GO
```

Unlike SQL Server 2000, SQL Server 2008 allows you to nest FOR XML queries. This method was used in Listing 2-23 to generate a hierarchical XML result.

Listing 2-23. *Nested FOR XML Queries*

```
USE AdventureWorks;
GO

-- SQL Server 2008 allows nested FOR XML queries, which can be
-- used to produce hierarchical XML results. This sample query
-- generates a nested hierarchical bill of materials for a
-- given AdventureWorks product. Notice that each nested subquery
-- is correlated by ComponentID and ProductAssemblyID.

DECLARE @ProductID int;
SET @ProductID = 749;

SELECT a.ComponentID AS "@id",
    p.ProductNumber AS "@number",
    p.Name AS "name",
    p.Color AS "color",
```

```

p.ListPrice AS "list-price",
a.PerAssemblyQty AS "quantity",
p.Size AS "size",
p.SizeUnitMeasureCode AS "unit-of-measure",
(
    SELECT b.ComponentID AS "@id",
           p.ProductNumber AS "@number",
           p.Name AS "name",
           p.Color AS "color",
           p.ListPrice AS "list-price",
           b.PerAssemblyQty AS "quantity",
           p.Size AS "size",
           p.SizeUnitMeasureCode AS "unit-of-measure",
    (
        SELECT c.ComponentID AS "@id",
               p.ProductNumber AS "@number",
               p.Name AS "name",
               p.Color AS "color",
               p.ListPrice AS "list-price",
               c.PerAssemblyQty AS "quantity",
               p.Size AS "size",
               p.SizeUnitMeasureCode AS "unit-of-measure",
        (
            SELECT d.ComponentID AS "@id",
                   p.ProductNumber AS "@number",
                   p.Name AS "name",
                   p.Color AS "color",
                   p.ListPrice AS "list-price",
                   d.PerAssemblyQty AS "quantity",
                   p.Size AS "size",
                   p.SizeUnitMeasureCode AS "unit-of-measure",
            (
                SELECT e.ComponentID AS "@id",
                       p.ProductNumber AS "@number",
                       p.Name AS "name",
                       p.Color AS "color",
                       p.ListPrice AS "list-price",
                       e.PerAssemblyQty AS "quantity",
                       p.Size AS "size",
                       p.SizeUnitMeasureCode AS "unit-of-measure"
            FROM Production.BillofMaterials e
            INNER JOIN Production.Product p
                ON e.ComponentID = p.ProductID
            WHERE e.ProductAssemblyID = d.ComponentID
            AND e.EndDate IS NULL
            FOR XML PATH (N'item'), TYPE
        )
    )
)

```

```

        FROM Production.BillofMaterials d
        INNER JOIN Production.Product p
            ON d.ComponentID = p.ProductID
        WHERE d.ProductAssemblyID = c.ComponentID
            AND d.EndDate IS NULL
        FOR XML PATH (N'item'), TYPE
    )
    FROM Production.BillofMaterials c
    INNER JOIN Production.Product p
        ON c.ComponentID = p.ProductID
    WHERE c.ProductAssemblyID = b.ComponentID
        AND c.EndDate IS NULL
    FOR XML PATH (N'item'), TYPE
)
FROM Production.BillofMaterials b
INNER JOIN Production.Product p
    ON b.ComponentID = p.ProductID
WHERE b.ProductAssemblyID = a.ComponentID
    AND b.EndDate IS NULL
FOR XML PATH(N'item'), TYPE
)
FROM Production.BillofMaterials a
INNER JOIN Production.Product p
    ON a.ComponentID = p.ProductID
WHERE p.ProductID = @ProductID
    FOR XML PATH(N'item'),
        ROOT(N'items'),
        TYPE;

```

In Chapter 2, I also discussed the legacy OPENXML functionality, which is provided for backward-compatibility with existing SQL Server 2000 and 2005 applications that rely on it. The `xml` data type and its methods should be used instead of the legacy OPENXML rowset provider for new development. Listing 2-24 demonstrates the use of OPENXML.

Listing 2-24. *Legacy OPENXML Demonstration*

```

USE AdventureWorks;
GO

-- Legacy OPENXML support is provided for backward-compatibility
-- with older applications. OPENXML was updated in SQL Server 2005
-- to support the xml data type, and this support continues in
-- SQL Server 2008. The xml data type and its methods should be
-- used instead of OPENXML in new development. OPENXML requires
-- use of the sp_xml_preparedocument and sp_xml_removedocument
-- system stored procedures.

```

```
-- First, retrieve an XML document from the Production.ProductModel table
DECLARE @x xml;

SELECT @x = pm.Instructions
FROM Production.ProductModel pm
WHERE pm.ProductModelID = 10;

-- Now create a document handle to the XML document
DECLARE @idoc int;

EXECUTE sp_xml_preparedocument @idoc OUTPUT, @x,
    '<ns xmlns:a =
      "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
      ProductModelManuInstructions" />';

-- The next step is to query the XML document using OPENXML
SELECT [Station], [Text]
FROM OPENXML(@idoc, '//a:Location/a:step', 3)
WITH (
    [Station] int '@LocationID',
    [Text] nvarchar(max) '.'
);

-- Finally, remove the XML document from memory
EXECUTE sp_xml_removedocument @idoc;
GO
```

The OPENROWSET provider can be used to populate an XML variable or column from an external file without any additional tools or programming. Listing 2-25 shows how to use OPENROWSET to load an XML file into an xml data type variable instance.

Listing 2-25. *Loading an xml Variable via OPENROWSET*

```
USE AdventureWorks;
GO

-- To use this example, copy the state-list.xml file to the c:
-- root directory, or change the code to point to a directory
-- of your choice.

DECLARE @x xml;

SELECT @x = xCol.BulkColumn
FROM OPENROWSET (BULK 'c:\state-list.xml', SINGLE_CLOB) AS xCol;

SELECT @x;
GO
```

Chapter 3

In Chapter 3, I began the discussion of the SQL Server xml data type. The first sample code, Listing 3-1, demonstrated how to populate an xml instance through both explicit and implicit casting of XML data from nvarchar format.

Listing 3-1. *Explicitly Casting nvarchar Data to xml*

```
-- The xml data type allows implicit and explicit casting of
-- character and binary data to populate xml instances.

DECLARE @imp_x xml,
        @exp_x xml,
        @source nvarchar(200);

SET @source = N'<?xml version = "1.0"?>
<message>
  <to>SQL Server Team</to>
  <from>Michael Coles</from>
  <subject>Thanks</subject>
  <content>Thanks for the new version of SQL Server</content>
</message>';

/* Implicit conversion to xml */
SET @imp_x = @source;

/* Explicit conversion to xml */
SET @exp_x = CAST(@source AS xml);
GO
```

SQL Server 2008 provides limited support for inline DTDs, including entity expansion. Listing 3-7 demonstrates how to use the CONVERT function to convert XML with a DTD into an xml data type instance.

Listing 3-7. *Sample xml Instance with DTD*

```
-- This example populates an xml data type instance from an
-- XML document with an inline DTD.

DECLARE @x xml;

SET @x = CONVERT(xml, N'<?xml version="1.0"?>
<!DOCTYPE meal [
  <!ATTLIST bread type CDATA "quick bread">
  <!ENTITY copyright "&#xA9; 2007 by Apress Chefs, Inc.">
]>
<meal>
  <food>
```

```

    <meat type="beef">Steak</meat>
    <vegetable type="legume">Green Beans</vegetable>
    <bread>Cornbread</bread>
    &copyright;
  </food>
</meal>', 2);

SELECT @x;
GO

```

In Chapter 3, I gave a quick introduction to the `xml` data type methods, `query()`, `value()`, `modify()`, `nodes()`, and `exist()`. Listing 3-8 is a simple example of the `xml` data type `query()` method that retrieves the street nodes from `xml` data type instances.

Listing 3-8. *query() Method Example*

```
-- This query() method example retrieves the street address nodes
-- from a sample XML document containing addresses of a chain of
-- bookstores.
```

```

DECLARE @x xml;
SET @x = N'<?xml version = "1.0"?>
<bookstores company = "Borders Group">

  <store name = "Borders">
    <address>
      <street>2 PENN PLAZA</street>
      <city>NEW YORK</city>
      <state>NY</state>
      <postal-code>10121-0101</postal-code>
      <country>US</country>
      <geo>
        <lat>40.749278</lat>
        <long>-73.992078</long>
      </geo>
    </address>
  </store>

  <store name = "Waldenbooks">
    <address>
      <street>318 E FAIRMOUNT AVENUE</street>
      <city>LAKEWOOD</city>
      <state>NY</state>
      <postal-code>14750-2007</postal-code>
      <country>US</country>
      <geo>
        <lat>42.098387</lat>
        <long>-79.305532</long>
      </geo>
    </address>
  </store>
</bookstores>

```



```

        </geo>
    </address>
</store>

<store name = "Borders Express">
    <address>
        <street>1401 ROUTE 300</street>
        <city>NEWBURGH</city>
        <state>NY</state>
        <postal-code>12550-2990</postal-code>
        <country>US</country>
        <geo>
            <lat>41.518067</lat>
            <long>-74.068843</long>
        </geo>
    </address>
</store>

</bookstores>';

SELECT @x.query(N'//street');
GO

```

Listing 3-9 demonstrates using the `value()` method to retrieve scalar atomic values from an `xml` data type instance.

Listing 3-9. *value() Method Example*

```

-- This example uses the value() method to retrieve scalar values
-- from an xml instance. The values retrieved are the price and
-- release date for a particular book. The value() method converts
-- the scalar values returned to the specified SQL Server data
-- types automatically.

DECLARE @x xml;

SET @x = N'<?xml version = "1.0"?>
<book>
    <title>Harry Potter and the Deathly Hallows</title>
    <author>Rowling, J.K.</author>
    <isbn>0545010225</isbn>
    <release-date>2007-07-21Z</release-date>
    <price>34.99</price>
</book>';

SELECT @x.value(N'(/book/price)[1]', 'decimal(5, 2)') AS Price,
       @x.value(N'(/book/release-date)[1]', 'date') AS Release_Date;
GO

```

The `exist()` method checks for the existence of a node within an `xml` data type instance. The code in Listing 3-10 uses the `exist()` method to check a simple XML-formatted menu for the item "pie".

Listing 3-10. *exist() Method Example*

-- This example creates a simple dessert menu in XML format
 -- and then checks for the existence of "pie" on the menu.

```
DECLARE @x xml;

SET @x = N'<?xml version = "1.0"?>
<dessert-menu>
  <item type = "pie">
    <name>Cherry Pie</name>
    <serving-info>
      <size>1 slice</size>
      <calories>277</calories>
      <price>2.99</price>
    </serving-info>
  </item>
  <item type = "cookie">
    <name>Peanut Butter Blossom</name>
    <serving-info>
      <size>3 cookies</size>
      <calories>348</calories>
      <price>1.29</price>
    </serving-info>
  </item>
  <item type = "cake">
    <name>German Chocolate Cake</name>
    <serving-info>
      <size>1 slice</size>
      <calories>793</calories>
      <price>3.99</price>
    </serving-info>
  </item>
</dessert-menu>';

SELECT CASE @x.exist(N'/dessert-menu/item[@type eq "pie"]')
          WHEN 1 THEN N'Pie is on the menu'
          WHEN 0 THEN N'There is no pie'
          ELSE 'XML instance is NULL'
END;

GO
```

Back in SQL Server 2000, the only way to shred XML data into relational format was through the OPENXML rowset provider. The `xml` data type provides the `nodes()` method to shred

XML data. Listing 3-11 uses the `nodes()` method in conjunction with the `value()` method to shred a simple XML bill of materials into relational format.

Listing 3-11. *nodes() Method Example*

```
-- The nodes() method used in this example shreds the XML data into
-- rows of xml instances, from which the value() method can retrieve
-- scalar values and return them in relational format.
```

```
DECLARE @x xml;

SET @x = N'<?xml version = "1.0"?>
<bill-of-materials>
  <finished-good name = "kiddie picnic table">
    <material name = "pine lumber">
      <item qty = "2">
        <dimensions uom = "mm">50 x 50 x 1100</dimensions>
      </item>
      <item qty = "4">
        <dimensions uom = "mm">50 x 25 x 800</dimensions>
      </item>
      <item qty = "8">
        <dimensions uom = "mm">50 x 25 x 400</dimensions>
      </item>
      <item qty = "2">
        <dimensions uom = "mm">50 x 50 x 475</dimensions>
      </item>
      <item qty = "4">
        <dimensions uom = "mm">50 x 50 x 180</dimensions>
      </item>
      <item qty = "6">
        <dimensions uom = "mm">50 x 50 x 75</dimensions>
      </item>
      <item qty = "5">
        <dimensions uom = "mm">100 x 25 x 800</dimensions>
      </item>
      <item qty = "8">
        <dimensions uom = "mm">50 x 25 x 800</dimensions>
      </item>
    </material>
    <material name="bolts">
      <item qty = "6">
        <dimensions uom = "mm">100 x 7</dimensions>
      </item>
      <item qty = "24">
        <dimensions uom = "mm">75 x 7</dimensions>
      </item>
    </material>
```

```

<material name="washers">
  <item qty = "30">
    <dimensions uom = "mm">7</dimensions>
  </item>
</material>
<material name="nuts">
  <item qty = "30">
    <dimensions uom = "mm">7</dimensions>
  </item>
</material>
<material name = "8-gauge treated screws">
  <item qty = "36">
    <dimensions uom = "mm">45</dimensions>
  </item>
</material>
</finished-good>
</bill-of-materials>';

```

```

SELECT my_table. my_col.value(N'../@name', N'nvarchar(100)') AS Material,
       my_table. my_col.value(N'./dimensions)[1]', N'nvarchar(50)') AS Dimensions,
       my_table. my_col.value(N'./dimensions/@uom')[1]', N'nvarchar(10)') AS UOM,
       my_table. my_col.value(N'./@qty', N'int') AS Quantity
FROM @x.nodes(N'//item') AS my_table ( my_col );
GO

```

The `modify()` method allows you to execute XML DML statements on an `xml` data type instance. XML DML statements allow you to insert, delete, and update nodes within an `xml` instance. Listing 3-12 uses the `modify()` method to delete a node from an `xml` instance.

Listing 3-12. *modify() Method Example*

```

-- The modify() method must be used in a SET statement to
-- update an xml type instance. The sample statement in this
-- listing deletes a product node from an XML formatted
-- inventory listing.

```

```

DECLARE @x xml;

SET @x = N'<?xml version = "1.0"?>
<inventory store-number = "9834">
  <product ean = "051500241776">
    <name>Jif Creamy Peanut Butter</name>
    <size>28 oz</size>
  </product>
  <product ean = "0024600010030">
    <name>Morton Iodized Salt</name>
    <size>26 oz</size>
  </product>

```

```

<product ean = "0086600000138">
  <name>Bumble Bee Chunked White Albacore in Water</name>
  <size>6 oz</size>
</product>
<product ean = "0013130006125">
  <name>Cream of Wheat Enriched Farina</name>
  <company>Nabisco</company>
  <size>28 oz</size>
</product>
</inventory>';

```

```
SET @x.modify (N'delete (/inventory/product[@ean = "0086600000138"]/name)');
```

```

SELECT @x;
GO

```

Chapter 4

XML schemas are defined by the W3C to constrain the structure and content of XML documents. In Chapter 4, I discussed SQL Server 2008 support for the XML Schema recommendation, and I provided a wide range of examples of XML schemas, from the very simple to more complex examples. Listing 4-10 demonstrated a complex XML schema that allowed recursive nesting of item elements in a bill of materials.

Listing 4-10. Complex Recursive BOM XML Schema

```

-- This XML schema is complex because it allows recursive nesting
-- of item elements. This is useful for constraining recursive
-- structures like the bill of materials used in the example.

CREATE XML SCHEMA COLLECTION dbo.ComplexBOMSchema
AS
N'<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="items">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="item">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" minOccurs="1" maxOccurs="1" />

```

```

        <xsd:element name="color" minOccurs="0" maxOccurs="1" />
        <xsd:group ref="price-group" minOccurs="1" maxOccurs="1" />
        <xsd:element name="quantity" minOccurs="1" maxOccurs="1" />
        <xsd:group ref="size-group" minOccurs="0" maxOccurs="1" />
        <xsd:element ref="item" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:attributeGroup ref="id-attr-group" />
</xsd:complexType>
</xsd:element>

<xsd:attributeGroup name="id-attr-group">
    <xsd:attribute name="id" />
    <xsd:attribute name="number" />
</xsd:attributeGroup>

<xsd:group name="price-group">
    <xsd:choice>
        <xsd:element name="list-price" minOccurs="1" maxOccurs="1" />
        <xsd:element name="standard-cost" minOccurs="1" maxOccurs="1" />
    </xsd:choice>
</xsd:group>

<xsd:group name="size-group">
    <xsd:sequence>
        <xsd:element name="size" minOccurs="1" maxOccurs="1" />
        <xsd:element name="unit-of-measure" minOccurs="1" maxOccurs="1" />
    </xsd:sequence>
</xsd:group>

</xsd:schema>';
GO

-- This portion of the example creates a typed xml instance
-- using the previously created XML schema collection to constrain
-- the structure and content. It then populates the xml instance
-- with recursive BOM data.

DECLARE @x XML (dbo.ComplexBOMSchema);

SET @x = N'<?xml version="1.0"?>
<items>
    <item id="749" number="BK-R93R-62">
        <name>Road-150 Red, 62</name>
        <color>Red</color>
        <list-price>3578.2700</list-price>
        <quantity>1.00</quantity>
        <size>62</size>
    </item>
</items>
';
```

```
<unit-of-measure>CM </unit-of-measure>
<item id="519" number="SA-R522">
  <name>HL Road Seat Assembly</name>
  <list-price>196.9200</list-price>
  <quantity>1.00</quantity>
</item>
<item id="717" number="FR-R92R-62">
  <name>HL Road Frame - Red, 62</name>
  <color>Red</color>
  <list-price>1431.5000</list-price>
  <quantity>1.00</quantity>
  <size>62</size>
  <unit-of-measure>CM </unit-of-measure>
</item>
<item id="807" number="HS-3479">
  <name>HL Headset</name>
  <list-price>124.7300</list-price>
  <quantity>1.00</quantity>
</item>
<item id="813" number="HB-R956">
  <name>HL Road Handlebars</name>
  <list-price>120.2700</list-price>
  <quantity>1.00</quantity>
</item>
<item id="820" number="FW-R820">
  <name>HL Road Front Wheel</name>
  <color>Black</color>
  <list-price>330.0600</list-price>
  <quantity>1.00</quantity>
</item>
<item id="828" number="RW-R820">
  <name>HL Road Rear Wheel</name>
  <color>Black</color>
  <list-price>357.0600</list-price>
  <quantity>1.00</quantity>
</item>
<item id="894" number="RD-2308">
  <name>Rear Derailleur</name>
  <color>Silver</color>
  <list-price>121.4600</list-price>
  <quantity>1.00</quantity>
</item>
<item id="907" number="RB-9231">
  <name>Rear Brakes</name>
  <color>Silver</color>
  <list-price>106.5000</list-price>
  <quantity>1.00</quantity>
```

```

    </item>
    <item id="940" number="PD-R853">
      <name>HL Road Pedal</name>
      <color>Silver/Black</color>
      <list-price>80.9900</list-price>
      <quantity>1.00</quantity>
    </item>
    <item id="945" number="FD-2342">
      <name>Front Deraillleur</name>
      <color>Silver</color>
      <list-price>91.4900</list-price>
      <quantity>1.00</quantity>
    </item>
    <item id="948" number="FB-9873">
      <name>Front Brakes</name>
      <color>Silver</color>
      <list-price>106.5000</list-price>
      <quantity>1.00</quantity>
    </item>
    <item id="951" number="CS-9183">
      <name>HL Crankset</name>
      <color>Black</color>
      <list-price>404.9900</list-price>
      <quantity>1.00</quantity>
    </item>
    <item id="952" number="CH-0234">
      <name>Chain</name>
      <color>Silver</color>
      <list-price>20.2400</list-price>
      <quantity>1.00</quantity>
    </item>
    <item id="996" number="BB-9108">
      <name>HL Bottom Bracket</name>
      <list-price>121.4900</list-price>
      <quantity>1.00</quantity>
    </item>
  </items>';

SELECT @x;
GO

```

During the discussion of XML schema, I covered advanced concepts like union data types, which are created from the union of value spaces of two given data types. Listing 4-15 creates a union data type that combines the value spaces of the `xsd:negativeInteger` and `xsd:positiveInteger` data types. This new union data type can accept any valid integer value except zero.

Listing 4-15. *Simple Union Data Type*

```

CREATE XML SCHEMA COLLECTION UnionSchemaCollection
AS
N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="nonZeroInteger">
    <xsd:simpleType>
      <xsd:union>
        <xsd:simpleType>
          <xsd:restriction base = "xsd:negativeInteger"/>
        </xsd:simpleType>
        <xsd:simpleType>
          <xsd:restriction base = "xsd:positiveInteger"/>
        </xsd:simpleType>
      </xsd:union>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>';
GO

DECLARE @x xml (UnionSchemaCollection);
SET @x = N'<nonZeroInteger>1</nonZeroInteger>';
SELECT @x;
GO

```

I also covered list data types, which define space-separated lists of items. The example in Listing 4-16 demonstrates the creation of a simple list data type that can hold lists of AdventureWorks product numbers. The `xsd:pattern` restriction and a regular expression are used to limit the lexical space of your data.

Listing 4-16. *Simple List Data Type*

```

CREATE XML SCHEMA COLLECTION ListSchemaCollection
AS
N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ProductNumberList">
    <xsd:simpleType>
      <xsd:list>
        <xsd:simpleType>
          <xsd:restriction base = "xsd:string">
            <xsd:pattern
              value = "[A-Z]{2}-[0-9A-Z][0-9]{2}[0-9A-Z](-([0-9]{2}|[BLMRSX]))?" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:list>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>';

```

```
GO

DECLARE @x xml (ListSchemaCollection);

SET @x = N'<ProductNumberList>RM-M823 FR-R92B-58
  CB-2903 FR-M94B-42</ProductNumberList>';

SELECT @x;

GO
```

Chapter 5

In Chapter 5, I discussed the W3C XQuery recommendation and SQL Server's implementation of XQuery in great detail. The basic building blocks of XQuery are expressions and sequences. Listing 5-3 demonstrated how to create sequences using parentheses and the comma operator. Some important properties of sequences are demonstrated in this example. For instance, sequences can contain duplicate values, their values are stored in creation order (or document order), and sequences contained within sequences are flattened out and contained empty sequences are removed.

Listing 5-3. *XQuery Sequence Creation*

```
DECLARE @x xml;
SELECT @x = N'';
SELECT @x.query('(1, 1, 2, (), 3, 4, 5, 10, (9, 8), 6, 7)');
```

An important aspect of XQuery expressions, of which I gave several examples in Chapter 5, is their predicates. Predicates are used to limit the results of an XQuery expression. XQuery provides many types of predicates and several comparison and logical operators, adding a lot of flexibility to XQuery predicates. Listing 5-7 demonstrated a simple predicate that limits the results to a Result node with a Name attribute equal to the string value "Apple Inc."

Listing 5-7. *Expression Using Value Comparison Operator*

```
DECLARE @x xml;

SELECT @x = N'
<Geocode-Results>
  <Result Name = "Microsoft Corp.">
    <Address>Microsoft Way</Address>
    <City>Redmond</City>
    <State>WA</State>
    <Zip>98052</Zip>
    <Latitude>47.643727</Latitude>
    <Longitude>-122.130474</Longitude>
  </Result>
  <Result Name = "Apple Inc.">
    <Address>1 Infinite Loop</Address>
```

```

    <City>Cupertino</City>
    <State>CA</State>
    <Zip>95014</Zip>
    <Latitude>37.332315</Latitude>
    <Longitude>-122.030749</Longitude>
  </Result>
</Geocode-Results>';

```

```
SELECT @x.query('/Geocode-Results/Result[@Name eq "Apple Inc."]);
```

Some of the new expressions provided by XQuery are the quantified expressions, *some* and *every*. These expressions return true if some or all of the values in a sequence, respectively, satisfy a given condition. Listing 5-10 demonstrated XQuery quantified expressions.

Listing 5-10. *XQuery Quantified Expressions in Action*

```

DECLARE @x xml;

SET @x = '';

SELECT @x.query ('some $x in (1, 2, 3)
  satisfies $x * $x = 9');

SELECT @x.query ('every $x in (1, 2, 3)
  satisfies $x * $x = 9');

```

One of the most powerful and highly touted features of XQuery is its support for FLWOR expressions. SQL Server 2008 has enhanced FLWOR expression support, including newly added support for the *let* clause. Listings 5-13 and 5-14 demonstrated usage of the FLWOR expression, including the newly added *let* clause support.

Listing 5-13. *FLWOR Expression with where Clause*

```

SELECT Resume.query('declare default element namespace
  "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
  for $i in (/Resume/Employment)
  where ($i/Emp.JobTitle ne "Machinist")
  return ($i/Emp.JobTitle)')
FROM HumanResources.JobCandidate
WHERE JobCandidateId = 1;

```

Listing 5-14. *FLWOR Expression let Clause Support*

```

SELECT Resume.query('declare default element namespace
  "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
  for $i in (/Resume/Employment)
  let $j := ($i/Emp.JobTitle)
  order by fn:string($j) descending
  return ($j)')

```

```
FROM HumanResources.JobCandidate
WHERE JobCandidateId = 1;
```

XQuery also supports direct XML construction, which allows you to create XML documents with a specific structure on the fly, quickly and easily. Listing 5-16 demonstrated direct construction of XML.

Listing 5-16. *XML Direct Construction*

```
SELECT Resume.query ('declare namespace
    ns = "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
<Education.History>
{
    for $i in (/ns:Resume/ns:Education)
    return
    (
        <Level>
        <Degree>
            { fn:data($i/ns:Edu.Level) }
        </Degree>
        <Date>
            { fn:data($i/ns:Edu.EndDate) }
        </Date>
        </Level>
    )
}
</Education.History>')
FROM HumanResources.JobCandidate
WHERE JobCandidateId = 2;
```

While direct XML construction is handy, experience has shown that sometimes XML construction requires fine-grained control. To support a higher degree of control over XML construction, XQuery also supports computed construction of XML, as demonstrated in Listing 5-17.

Listing 5-17. *XML Computed Construction*

```
SELECT Resume.query ('declare namespace
    ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
for $i in (/ns:Resume/ns:Education)
return (
    element Education.History
    {
        element Level
        {
            attribute School { fn:data($i/ns:Edu.School) },
            element Degree { fn:data($i/ns:Edu.Level) },
            element Date { fn:data($i/ns:Edu.EndDate) }
        }
    }
)
```

```

    }
  )')
FROM HumanResources.JobCandidate
WHERE JobCandidateId = 2;

```

I also discussed the xml data type methods in greater detail in Chapter 5. One of the significant advantages of the xml data type is its ability to shred XML data into relational format without the need for the developer to create, manage, and dispose of external COM object instances. Listing 5-20 demonstrated using the nodes() and value() methods to turn SQL Server's cached XML query plans into relational data.

Listing 5-20. *Querying the Cached XML Query Plans*

```

WITH Plans(nodeid, physicalop, estimated_cost, plan_handle,
    text, query_plan, cacheobjtype, objtype)
AS
(
    SELECT RelOp.op.value('declare default element namespace
        "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
        @NodeId', 'int'),
        RelOp.op.value('declare default element namespace
        "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
        @PhysicalOp', 'varchar(50)'),
        RelOp.op.value('declare default element namespace
        "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
        @EstimatedTotalSubtreeCost ', 'float'),
        cp.plan_handle,
        st.text,
        qp.query_plan,
        cp.cacheobjtype,
        cp.objtype
    FROM sys.dm_exec_cached_plans cp
    CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
    CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle) qp
    CROSS APPLY qp.query_plan.nodes('declare default element namespace
        "http://schemas.microsoft.com/sqlserver/2004/07/showplan";
        //RelOp') RelOp (op)
)
SELECT ROW_NUMBER() OVER (PARTITION BY p.plan_handle ORDER BY p.NodeId)
    AS Operation_Num,
    p.physicalop,
    p.text,
    p.cacheobjtype,
    p.objtype,
    p.estimated_cost
FROM Plans p
WHERE p.cacheobjtype = 'Compiled Plan';

```

Chapter 6

XQuery provides a complete set of functions and operators. There are so many in fact that they fill up an entire W3C recommendation by themselves. SQL Server supports a large subset of these functions and operators, as well as some extensions to XQuery for manipulating XML, programmatically known as XML DML. In Chapter 6, I discussed the SQL Server–supported functions and operators, as well as XML DML.

XQuery supports casting data types via the `cast` expression, which was demonstrated in Listing 6-1. SQL Server requires you to use the optional occurrence indicator (?) after the expression to indicate that an empty sequence could be returned.

Listing 6-1. *cast Operator Example*

```
DECLARE @x xml;
SET @x = '';
SELECT @x.query(' "123.4567" cast as xs:decimal? ');
```

SQL Server also supports a variety of standard numeric and math functions. Listing 6-6 demonstrated some of these numeric functions in action.

Listing 6-6. *Numeric Function Example*

```
DECLARE @x xml;
SET @x = '';
SELECT 'fn:round', @x.query('fn:round(-10.4)'),
    @x.query('fn:round(10.4)');
SELECT 'fn:ceiling', @x.query('fn:ceiling(-10.4)'),
    @x.query('fn:ceiling(10.4)');
SELECT 'fn:floor', @x.query('fn:floor(-10.4)'),
    @x.query('fn:floor(10.4)');
```

To make XQuery even more useful to SQL Server developers, SQL Server 2008 provides the `sql:column` and `sql:variable` functions. These functions allow you to access relational data and T-SQL variables from within XQuery queries, providing powerful XML-to-relational data integration. Listing 6-16 demonstrates the `sql:column` function in action.

Listing 6-16. *Constructing XML with Relational Data*

```
SELECT jc.BusinessEntityID, Resume.query ('declare namespace ns =
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
    <contact>
        <name> {
            sql:column("p.LastName"),
            sql:column("p.FirstName"),
            sql:column("p.MiddleName")
        } </name>
    </contact>')
FROM HumanResources.JobCandidate jc
INNER JOIN HumanResources.Employee e
```

```

ON jc.BusinessEntityID = e.BusinessEntityID
INNER JOIN Person.Person p
ON e.BusinessEntityID = p.BusinessEntityID
WHERE jc.BusinessEntityID = 274;

```

SQL Server 2008 XML DML has been improved to include support for using xml data type variables in XML DML insert statements. Listing 6-18 demonstrated using the sql:variable function with a T-SQL xml data type variable in an XML DML statement.

Listing 6-18. *Using sql:variable() with XML DML insert Statement*

```

DECLARE @x xml,
        @y xml;

SET @y = N'<ns:Employment xmlns:ns =
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume">
    <ns:Emp.StartDate>2001-01-01Z</ns:Emp.StartDate>
    <ns:Emp.EndDate>2007-10-31Z</ns:Emp.EndDate>
    <ns:Emp.OrgName>AdventureWorks</ns:Emp.OrgName>
    <ns:Emp.JobTitle>Vice President of Sales</ns:Emp.JobTitle>
</ns:Employment>';

SELECT @x = Resume
FROM HumanResources.JobCandidate
WHERE BusinessEntityID = 274;

SET @x.modify('declare namespace ns =
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume";
insert
    sql:variable("@y")
before
    (/ns:Resume/ns:Employment)[1]');

SELECT @x;

```

Chapter 7

In Chapter 7, I discussed how to increase xml data type query performance through the use of XML indexes. The primary XML index is a prerequisite to performing any kind of indexing on xml data type columns, and Listing 7-1 demonstrated how to create one. One important take-away from this example is the SQL Server connection settings. Creating primary XML indexes requires that certain options be set, or the index will not be created. The settings required are basically the same as for creation of an indexed view. The listing demonstrates the proper settings.

Listing 7-1. *Primary XML Index Creation*

```

SET ARITHABORT ON;
SET CONCAT_NULL_YIELDS_NULL ON;
SET QUOTED_IDENTIFIER ON;
SET ANSI_NULLS ON;
SET ANSI_PADDING ON;
SET ANSI_WARNINGS ON;
SET NUMERIC_ROUNDABORT OFF;
GO

IF EXISTS ( SELECT 1
            FROM sys.indexes i
            WHERE i.name = 'PXML_ProductModel_CatalogDescription'
          )
    DROP INDEX PXML_ProductModel_CatalogDescription
    ON Production.ProductModel;
GO

CREATE PRIMARY XML INDEX PXML_ProductModel_CatalogDescription
ON Production.ProductModel
(
    CatalogDescription
);
GO

```

Secondary XML indexes are built on top of the primary XML index to further increase XQuery efficiency. Listing 7-3 demonstrated the syntax for creating a PATH type secondary XML index.

Listing 7-3. *PATH Secondary XML Index*

```

CREATE XML INDEX SXML_ProductModel_Instructions_PATH
ON Production.ProductModel
(
    Instructions
)
USING XML INDEX PXML_ProductModel_Instructions
FOR PATH;

```

Full-text search provides another tool for increasing XML query efficiency. Listings 7-9 and 7-10 demonstrated full-text catalog and full-text index creation on an xml data type column.

Listing 7-9. *Full-Text Catalog Creation*

```

CREATE FULLTEXT CATALOG FTC_Instructions_XML;

```


Listing 7-10. *Full-Text Index on an xml Column*

```
CREATE FULLTEXT INDEX
ON Production.ProductModel
(
    Instructions
)
KEY INDEX PK_ProductModel_ProductModelID
ON FTC_Instructions_XML;
```

Listing 7-12 demonstrated a compound SQL predicate that combined both a full-text search FREETEXT predicate and the xml data type exist() method to increase query efficiency.

Listing 7-12. *Full-Text Search Compound Predicate Example*

```
SELECT *
FROM Production.ProductModel
WHERE FREETEXT(Instructions, 'applies')
AND Instructions.exist('declare namespace ns =
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/➡
    ProductModelManuInstructions";
    (/ns:step/ns:material[contains(., "aluminum")])') = 1;
```

Chapter 8

Chapter 8 introduced methods for taking advantage of .NET-based XML functionality via SQL-CLR integration. The focus of Chapter 8 was on performing XSL transformations from T-SQL. The heart of the SQLCLR XSLT functionality is found in the support functions in Listing 8-4. This functionality takes advantage of the .NET XslCompiledTransform class and the SqlXml SQL Server nullable data type.

Listing 8-4. *XSLT Support Functions*

```
using System.Data.SqlTypes;
using System.Text;
using System.Xml;
using System.Xml.Xsl;
using System.IO;

namespace Apress.Samples
{
    public partial class XsltAssembly
    {
        private static XmlReader CreateXmlReader(SqlXml source_xml)
        {
            byte[] xmldata = Encoding.UTF8.GetBytes(source_xml.Value);
            MemoryStream stream = new MemoryStream(xmldata);
```

```

        return XmlReader.Create(stream);
    }

    private static XslCompiledTransform CreateXsltStylesheet (SqlXml source_xslt)
    {
        XslCompiledTransform xslt_stylesheet = new XslCompiledTransform();
        xslt_stylesheet.Load(CreateXmlReader(source_xslt));
        return xslt_stylesheet;
    }

    private static MemoryStream ApplyStylesheet(XmlReader source_xml_reader,
        XslCompiledTransform xslt_stylesheet)
    {
        MemoryStream buffer = new MemoryStream();
        StreamWriter stream = new StreamWriter(buffer);
        xslt_stylesheet.Transform(source_xml_reader, XmlWriter.Create(stream));
        return buffer;
    }

    private static SqlXml Transform(SqlXml source_xml,
        SqlXml source_xslt)
    {
        XmlReader source_xml_reader = CreateXmlReader(source_xml);
        XslCompiledTransform xslt_stylesheet = CreateXsltStylesheet(source_xslt);
        return new SqlXml(ApplyStylesheet(source_xml_reader, xslt_stylesheet));
    }
}

```

The C# XSL transformation function, which performs transformations in memory, was given in Listing 8-5. The result of this XSL transformation is returned as a SQL Server xml data type instance.

Listing 8-5. In-Memory XSL Transformation Function

```

using System.Data.SqlTypes;

namespace Apress.Samples
{
    public partial class XsltAssembly
    {
        [Microsoft.SqlServer.Server.SqlFunction]
        public static SqlXml fn_XsltTransform(SqlXml source_xml,
            SqlXml source_xslt)
        {
            SqlXml result = new SqlXml();
            if (source_xml.IsNull ||
                source_xslt.IsNull)

```

```

        result = SqlXml.Null;
    else
        result = Transform(source_xml, source_xslt);
    return result;
}
}
}

```

Listing 8-6 provided another XSL transformation routine. Instead of returning an xml data type instance in memory, this procedure writes its result to a specified file in the file system.

Listing 8-6. *Procedure to Transform and Write to File*

```

using System.Data.SqlTypes;
using System.Xml;
using System.IO;

namespace Apress.Samples
{
    public partial class XsltAssembly
    {
        [Microsoft.SqlServer.Server.SqlProcedure]
        public static void p_XsltTransformToFile(SqlXml source_xml,
            SqlXml source_xslt,
            SqlString output_file)
        {
            SqlXml result = new SqlXml();
            if (!source_xml.IsNull &&
                !source_xslt.IsNull &&
                !output_file.IsNull)
            {
                result = Transform(source_xml, source_xslt);
                StreamWriter sw = new StreamWriter(output_file.Value);
                sw.Write(result.Value);
                sw.Dispose();
            }
        };
    }
}

```

Once I'd introduced these routines, of course, I had to show them in action. Listing 8-7 used FOR XML PATH to generate an XML report of customers by state in the United States. The generated xml data type instance was then transformed to HTML and saved to the file system where it could be opened and viewed in any standard web browser.

Listing 8-7. *AdventureWorks XSLT Customer Summary Report*

```

DECLARE @source_xslt xml,
        @source_xml xml;

SELECT @source_xml =
(
    SELECT ic1.CountryRegionName AS "Country",
        (
            SELECT COUNT(*) AS "Count",
                ic2.StateProvinceName AS "State-Name"
            FROM Sales.vIndividualCustomer ic2
            WHERE ic2.CountryRegionName = ic1.CountryRegionName
            GROUP BY ic2.CountryRegionName, ic2.StateProvinceName
            FOR XML PATH ('State'), TYPE
        ) AS "Customers"
    FROM Sales.vIndividualCustomer ic1
    WHERE ic1.CountryRegionName = 'United States'
    GROUP BY CountryRegionName
    ORDER BY CountryRegionName
    FOR XML PATH ('Individual-Customer-Summary'), TYPE
);

SET @source_xslt = N'<xsl:stylesheet version = "1.0"
    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
    xmlns = "http://www.w3.org/1999/xhtml">

    <xsl:template match = "/Individual-Customer-Summary">
        <html>

            <head>
                <style type = "text/css">

                    table {
                        border-width: 1px;
                        border-style: solid;
                        border-color: black;
                        border-collapse: collapse;
                        background-color: white;
                        width: 50%;
                    }

                    table th {
                        border-width: 1px;
                        padding: 3px;
                        border-style: dotted;
                        border-color: gray;
                        background-color: #6666dd;

```

```

        font-family: arial;
        font-size: 12px;
        color: white;
    }

    table td.light {
        border-width: 1px;
        padding: 3px;
        border-style: dotted;
        border-color: gray;
        background-color: white;
        font-family: arial;
        font-size: 12px;
    }

    table td.dark {
        border-width: 1px;
        padding: 3px;
        border-style: dotted;
        border-color: gray;
        background-color: #66ffff;
        font-family: arial;
        font-size: 12px;
    }
}

</style>
</head>

<body>
    <h2>
        AdventureWorks Customer Breakdown: <xsl:value-of select = "Country"/>
    </h2>
    <table>
        <tr>
            <th>
                State
            </th>
            <th>
                Customer Count
            </th>
        </tr>
        <xsl:for-each select="Customers/State">
            <xsl:sort select="Count" data-type="number" order="descending"/>
            <xsl:sort select="State-Name" order="ascending"/>
            <tr>
                <xsl:choose>
                    <xsl:when test = "position() mod 2 = 0">

```

```

        <td class = "dark">
            <xsl:value-of select = "State-Name" />
        </td>
        <td class="dark">
            <xsl:value-of select = "Count"/>
        </td>
    </xsl:when>
    <xsl:otherwise>
        <td class="light">
            <xsl:value-of select = "State-Name" />
        </td>
        <td class="light">
            <xsl:value-of select = "Count"/>
        </td>
    </xsl:otherwise>
</xsl:choose>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>';

EXEC p_XsltTransformToFile @source_xml,
    @source_xslt,
    N'C:\customer_report.html';

GO

```

Chapter 9

In Chapter 9, I started looking at taking advantage of SQL Server's XML functionality from the client's perspective. HTTP SOAP endpoints make it easier than ever to expose SQL Server procedures and user-defined functions to remote clients safely and securely. Note that the sample AdventureWorks Product Browser application presented in Chapter 9 has a dependency on the XSL transformation functions presented in Chapter 8, so they should be installed before trying to run these samples.

After installing the stored procedures and user-defined functions you want to expose as web methods, Listing 9-4 creates an HTTP SOAP endpoint that lets client applications access them.

Listing 9-4. *Create an HTTP SOAP Endpoint*

```

CREATE ENDPOINT AdvWorksProductBrowser
STATE = STARTED
AS HTTP
(

```

```

    PATH = '/Browser',
    AUTHENTICATION = ( INTEGRATED ),
    PORTS = ( CLEAR ),
    SITE = '*',
    CLEAR_PORT = 888
)
FOR SOAP
(
    WEBMETHOD 'GetProductHierarchy'
    (
        NAME = 'AdventureWorks.dbo.p_GetProductHierarchy',
        SCHEMA = STANDARD,
        FORMAT = ROWSETS_ONLY
    ),
    WEBMETHOD 'GetProductPhoto'
    (
        NAME = 'AdventureWorks.dbo.fn_GetProductPhoto',
        SCHEMA = STANDARD,
        FORMAT = ALL_RESULTS
    ),
    WEBMETHOD 'GetHtmlCatalogDescription'
    (
        NAME = 'AdventureWorks.dbo.fn_GetHtmlCatalogDescription',
        SCHEMA = STANDARD,
        FORMAT = ALL_RESULTS
    ),
    BATCHES = DISABLED,
    WSDL = DEFAULT,
    LOGIN_TYPE = WINDOWS,
    DATABASE = 'AdventureWorks'
);

```

Listing 9-5 demonstrates C# code to consume the HTTP SOAP web methods from the client, including methods to populate a Windows Forms TreeView control, a WebBrowser control, and an Image control.

Listing 9-5. *Web Service Method Call Private Functions*

```

private void PopulateSearchTree()
{
    Sql2008.AdvWorksProductBrowser ws = new Sql2008.AdvWorksProductBrowser();
    ws.Credentials = CredentialCache.DefaultCredentials;
    DataSet ds = ws.GetProductHierarchy();

    string CategoryName = "";
    string SubcategoryName = "";
    string ProductName = "";
    TreeNode CategoryNode = new TreeNode();

```

```

TreeNode SubcategoryNode = new TreeNode();
foreach (DataRow dr in ds.Tables[0].Rows)
{
    CategoryName = dr["ProductCategoryName"].ToString();
    SubcategoryName = dr["ProductSubcategoryName"].ToString();
    ProductName = dr["ProductName"].ToString();
    if (CategoryName != CategoryNode.Text)
    {
        CategoryNode = new TreeNode();
        CategoryNode.Text = CategoryName;
        treeView1.Nodes.Add(CategoryNode);
    }
    if (SubcategoryName != SubcategoryNode.Text)
    {
        SubcategoryNode = new TreeNode();
        SubcategoryNode.Text = SubcategoryName;
        CategoryNode.Nodes.Add(SubcategoryNode);
    }
    TreeNode ProductNode = new TreeNode();
    ProductNode.Text = ProductName;
    ProductNode.Tag = (int)dr["ProductID"];
    SubcategoryNode.Nodes.Add(ProductNode);
}
}

private Image GetProductImage(int ProductID)
{
    Sql2008.AdvWorksProductBrowser ws = new Sql2008.AdvWorksProductBrowser();
    ws.Credentials = CredentialCache.DefaultCredentials;
    SqlBinary photo = ws.GetProductPhoto(new SqlInt32(ProductID), SqlBoolean.False);
    MemoryStream ms = new MemoryStream(photo.Value);
    return Image.FromStream(ms);
}

private string GetHtmlProductDescription(int ProductID)
{
    Sql2008.AdvWorksProductBrowser ws = new Sql2008.AdvWorksProductBrowser();
    ws.Credentials = CredentialCache.DefaultCredentials;
    Sql2008.xml html =
        ws.GetHtmlCatalogDescription(new SqlInt32(ProductID));
    string html_string = "<html><body><b>No Description</b></body></html>";
    if (html != null)
        html_string = html.Any[0].OuterXml;
    return html_string;
}

```


Chapter 10

Chapter 10 continued the exploration of .NET XML support with examples like the function in Listing 10-3 that validates an XML document's structure and content against a DTD, functionality that is missing from the SQL Server `xml` data type. An important thing to note about this function is that it requires the XML be passed to it as a `SqlString`. This is required because SQL Server automatically strips the DTD from `xml` data type instances, which would defeat the purpose of this function.

Listing 10-3. *SQLCLR UDF to Validate XML Against a DTD*

```
using System;
using System.Data.SqlTypes;
using System.Xml;
using System.IO;
using System.Text;
using System.Xml.Schema;

namespace Apress.Samples
{
    public partial class ValidationDTD
    {
        [Microsoft.SqlServer.Server.SqlFunction]
        public static SqlBoolean fn_ValidateDTD(SqlString xml)
        {
            SqlBoolean result = new SqlBoolean(true);
            try
            {
                UnicodeEncoding encoder = new UnicodeEncoding();
                byte[] buffer = encoder.GetBytes(xml.Value);
                MemoryStream stream = new MemoryStream(buffer);

                XmlReaderSettings settings = new XmlReaderSettings();
                settings.CloseInput = true;
                settings.ValidationFlags = settings.ValidationFlags |
                    XmlSchemaValidationFlags.ReportValidationWarnings;
                settings.ConformanceLevel = ConformanceLevel.Document;
                settings.ValidationType = ValidationType.DTD;
                settings.ProhibitDtd = false;

                XmlReader reader = XmlReader.Create(stream, settings);

                settings.ValidationEventHandler +=
                    new ValidationEventHandler(MyHandler);

                while (reader.Read()) { ; }
            }
            catch (Exception ex)
```

```

        {
            result = SqlBoolean.False;
            throw (new Exception(ex.Message));
        }
        return result;
    }

    private static void MyHandler(object sender, ValidationEventArgs e)
    {
        throw (e.Exception);
    }
}
};

```

Listing 10-5 demonstrated using this function to validate an invalid document against a DTD. Note that this code returns an error because the XML is invalid per the DTD.

Listing 10-5. *Invalid XML Document Validation Against a DTD*

```

DECLARE @eContract nvarchar(max);

SET @eContract = N'<?xml version="1.0" encoding="UTF-16"?>

<!DOCTYPE contract
    SYSTEM "c:\eContracts-Reference.dtd">

<sales-invoice>
</sales-invoice>';

SELECT dbo.fn_ValidateDTD(@eContract);

```

I also considered situations in which it might be beneficial to access XML data remotely on a network or on the Web. It's not an uncommon requirement to access data on mainframe computers over a network, for instance. Listing 10-6 demonstrated the simplicity of a SQLCLR function that retrieves an XML file remotely over a network and returns it as an `xml` data type instance.

Listing 10-6. *Function to Retrieve an XML Document Remotely*

```

using System.Data.SqlTypes;
using System.Net;
using System.IO;
using System.Text;

namespace Apress.Sample
{
    public partial class UserDefinedFunctions
    {
        [Microsoft.SqlServer.Server.SqlFunction]

```

```

    public static SqlXml fn_GetRemoteXML(SqlString uri)
    {
        WebClient wc = new WebClient();
        byte[] reqXML = wc.DownloadData(uri.Value);
        MemoryStream ms = new MemoryStream(reqXML);
        return new SqlXml(ms);
    }
};
}

```

Listing 10-7 gave a simple demonstration of retrieving an RSS feed remotely and shredding it into relational format.

Listing 10-7. Retrieving and Shredding an RSS Feed

```

DECLARE @rss xml;

SET @rss = dbo.fn_GetRemoteXML('http://www.sqlservercentral.com/Xml/Rss/➡
Articles/SQL+Server+2008');

SELECT Col.value('title[1]', 'varchar(200)') AS Title,
       Col.value('link[1]', 'varchar(200)') AS Link,
       Col.value('description[1]', 'varchar(800)') AS [Description],
       Col.value('language[1]', 'varchar(20)') AS Lang,
       Col.value('ttl[1]', 'int') AS TTL,
       Col.value('managingEditor[1]', 'varchar(100)') AS ManagingEditor
FROM @rss.nodes('/rss/channel') AS Feed(Col)

SELECT Col.value('title[1]', 'varchar(200)') AS Title,
       Col.value('description[1]', 'varchar(800)') AS [Description],
       Col.value('guid[1]', 'varchar(200)') AS [GUID],
       Col.value('pubDate[1]', 'datetime') AS [PubDate],
       Col.value('link[1]', 'varchar(200)') AS [Link]
FROM @rss.nodes('/rss/channel/item') FeedItems (Col);

```

In Chapter 10 I also demonstrated REST-style service calls from SQL Server. Listing 10-8 demonstrates a simple Yahoo! API geocoding service call.

Listing 10-8. C# Function to Call Yahoo! API

```

using System.Data.SqlTypes;
using System.Net;
using System.IO;
using System.Text;

namespace Apress.Sample
{
    public partial class UserDefinedFunctions
    {

```

```

[Microsoft.SqlServer.Server.SqlFunction]
public static SqlXml fn_YahooGeocodeRest(SqlString location)
{
    WebClient wc = new WebClient();
    string uri = string.Format(http://local.yahooapis.com/" +
        "MapsService/V1/geocode?appid=YahooDemo&location={0}",
        location.Value);
    byte[] reqXML = wc.DownloadData(uri);
    MemoryStream ms = new MemoryStream(reqXML);
    return new SqlXml(ms);
}
};
}

```

Chapter 11

Chapter 11 focused on the Geography Markup Language (GML), which is the XML-based standard for transmitting and sharing geographic data. GML support is provided with the new SQL Server 2008 geometry and geography data types. Listing 11-3 demonstrated creation of a geometric shape representing the state of Delaware and points inside and outside the state.

Listing 11-3. *GML for Borders of the State of Delaware*

```
DECLARE @Delaware geometry;
```

```

SET @Delaware = geometry::GeomFromGml(N'<gml:Polygon
xmlns:gml="http://www.opengis.net/gml">
  <gml:exterior>
    <gml:LinearRing>
      <gml:posList>
        -75.70742 38.557476 -75.71106 38.649551 -75.724937 38.83017
        -75.752922 39.141548 -75.761658 39.247753 -75.764664 39.295849
        -75.772697 39.383007 -75.791435 39.723755 -75.775269 39.724442
        -75.745934 39.774818 -75.695114 39.820347 -75.644341 39.838196
        -75.583794 39.840008 -75.470345 39.826435 -75.42083 39.79887
        -75.412117 39.789658 -75.428009 39.77813 -75.460754 39.763248
        -75.475128 39.741718 -75.476334 39.719971 -75.489639 39.714745
        -75.610725 39.612793 -75.562996 39.566723 -75.590187 39.463768
        -75.515572 39.36694 -75.402481 39.257637 -75.397728 39.073036
        -75.324852 39.012386 -75.307899 38.945911 -75.190941 38.80867
        -75.083138 38.799812 -75.045998 38.44949 -75.068298 38.449963
        -75.093094 38.450451 -75.350204 38.455208 -75.69915 38.463066
        -75.70742 38.557476
      </gml:posList>
    </gml:LinearRing>
  </gml:exterior>
</gml:Polygon>', 0);

```

```

DECLARE @DelawareStateCapitol geometry,
        @LibertyBell geometry;

SET @DelawareStateCapitol = Geometry::GeomFromGml(N'<gml:Point
    xmlns:gml = "http://www.opengis.net/gml">
    <gml:pos>-75.522864 39.156473</gml:pos>
</gml:Point>', 0);

SET @LibertyBell = Geometry::GeomFromGml(N'<gml:Point
    xmlns:gml = "http://www.opengis.net/gml">
    <gml:pos>-75.15028 39.95028</gml:pos>
</gml:Point>', 0);

SELECT 'DE Contains DE State Capitol' AS [Desc],
        @Delaware.STContains(@DelawareStateCapitol) AS Flag
UNION
SELECT 'DE Contains Liberty Bell', @Delaware.STContains(@LibertyBell);

```

Chapter 12

In Chapter 12, I discussed SQLXML support in .NET and SQL Server 2008. SQLXML is an OLEDB/COM-based tool for querying and manipulating relational data as XML. The first listing of the chapter, Listing 12-1, showed how to query relational data using the FOR XML NESTED clause to format your result as XML on the client side. To prove that the formatting is occurring on the client side, the example adds the FOR XML NESTED clause to a stored procedure call, a construction that would be invalid on the server.

Listing 12-1. Sample SQLXML Query Code

```

string SqlXmlConnString = "Provider=SQLOLEDB;" +
    "Server=(local);" +
    "Database=AdventureWorks;" +
    "Integrated Security=SSPI";

private void btnQuery_Click(object sender, EventArgs ev)
{
    SqlXmlCommand cmd = new SqlXmlCommand(SqlXmlConnString);
    cmd.CommandText = "EXEC dbo.uspGetEmployeeManagers ? " +
        "FOR XML NESTED";
    cmd.ClientSideXml = true;

    SqlXmlParameter p;
    p = cmd.CreateParameter();
    p.Value = txtContactID.Text;

    try
    {
        Stream strm = cmd.ExecuteStream();
    }
}

```

```

        strm.Position = 0;
        using (StreamReader sr = new StreamReader(strm))
        {
            txtResult.Text = sr.ReadToEnd();
        }
    }
    catch (SqlXmlException e)
    {
        e.ErrorStream.Position = 0;
        string s = new StreamReader(e.ErrorStream).ReadToEnd();
        System.Console.WriteLine(s);
    }
}

```

I also discussed mapping schemas that provide an XML view over your relational data. Listing 12-2 showed a simple mapping schema that mapped the columns of one table to XML nodes.

Listing 12-2. *XML Mapping Schema*

```

<?xml version = "1.0"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
  xmlns:sql = "urn:schemas-microsoft-com:mapping-schema"
  elementFormDefault = "qualified"
  attributeFormDefault = "unqualified"
  xmlns:msdata = "urn:schemas-microsoft-com:mapping-schema">

  <xs:annotation>
    <xs:appinfo>
      <msdata:relationship name = "Contact-Person"
        parent = "Person.BusinessEntity"
        parent-key = "BusinessEntityID"
        child = "Person.Person"
        child-key = "BusinessEntityID"/>
    </xs:appinfo>
  </xs:annotation>

  <xs:element name = "Contact"
    sql:relation = "Person.BusinessEntity">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "Id"
          type = "xs:integer"
          sql:field = "BusinessEntityID"
          sql:identity = "useValue"/>
        <xs:element name = "Person"
          sql:relation = "Person.Person"
          sql:relationship = "Contact-Person">

```

```

<xs:complexType>
  <xs:attribute name = "id"
    type = "xs:string"
    use = "optional"
    sql:field = "BusinessEntityID" />
  <xs:attribute name = "person-type"
    type = "xs:string"
    use = "required"
    sql:field = "PersonType"/>
  <xs:attribute name = "name-style"
    type = "xs:integer"
    use = "required"
    sql:field = "NameStyle"/>
  <xs:attribute name = "last-name"
    type = "xs:string"
    use = "required"
    sql:field = "LastName"/>
  <xs:attribute name = "first-name"
    type = "xs:string"
    use = "required"
    sql:field = "FirstName"/>
  <xs:attribute name = "promotion"
    type = "xs:integer"
    use = "optional"
    sql:field = "EmailPromotion"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

```

Updategrams provide the classic tools for updating relational data through an XML view. Listing 12-3 was a sample insert updategram.

Listing 12-3. *Sample insert Updategram*

```

<?xml version = "1.0"?>
<ROOT xmlns:sql = "urn:schemas-microsoft-com:xml-sql"
  xmlns:updg = "urn:schemas-microsoft-com:xml-updategram">
  <updg:sync mapping-schema = "c:\contactupdg.xsd">
    <updg:before>
      </updg:before>
    <updg:after>
      <Contact>
        <Id>21000</Id>
        <Person person-type = "IN"
          name-style = "0"

```

```

        last-name = "Braff"
        first-name = "Zach"
        promotion = "0" />
    </Contact>
</updg:after>
</updg:sync>
</ROOT>

```

The .NET Framework provides access to SQLXML functionality through a set of COM interop wrapper classes. The `SqlXmlCommand` class was demonstrated in Listing 12-6.

Listing 12-6. *SqlXmlCommand Updategram Sample Code*

```

// The SqlXmlConnString is the SQLXML connection string
string SqlXmlConnString = "Provider=SQLOLEDB;" +
    "Server=(local);" +
    "database=AdventureWorks;" +
    "Integrated Security=SSPI";

// The Updategrams hash table contains all three sample updategrams,
// the sample Insert updategram, sample Update updategram, and sample
// Delete updategram.
Hashtable Updategrams = new Hashtable(3);

. . .

// The sample updategrams are added to the hash table below
Updategrams.Add("Insert", "<?xml version='1.0'?'> " +
    "<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql' " +
    "xmlns:updg='urn:schemas-microsoft-com:xml-updategram'> " +
    . . .
    "</ROOT>");

Updategrams.Add("Update", "<?xml version='1.0'?'> " +
    "<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql' " +
    "xmlns:updg='urn:schemas-microsoft-com:xml-updategram'> " +
    . . .
    "</ROOT>");

Updategrams.Add("Delete", "<?xml version='1.0'?'> " +
    "<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql' " +
    "xmlns:updg='urn:schemas-microsoft-com:xml-updategram'> " +
    . . .
    "</ROOT>");

. . .

```



```

SqlXmlCommand cmd = new SqlXmlCommand(SqlXmlConnectionString);
cmd.CommandType = SqlXmlCommandType.UpdateGram;
cmd.RootTag = "ROOT";

cmd.CommandText = Updategrams[btnInsertUpdateDelete.Text].ToString();
txtResult.Text = Updategrams[btnInsertUpdateDelete.Text].ToString();
cmd.ExecuteNonQuery();

```

I also discussed SQLXML bulk loading, which allows you to bulk load XML-formatted data into a relational database via a mapping schema, as shown in Listing 12-7.

Listing 12-7. *SQLXML Bulk Load Mapping Schema*

```

<?xml version = "1.0"?>
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
  xmlns:sql = "urn:schemas-microsoft-com:mapping-schema">

  <xsd:element name = "CustomerCoordinates"
    sql:relation = "Sales.CustomerGeography">

    <xsd:complexType>

      <xsd:sequence>

        <xsd:element name = "Resolution"
          sql:field = "Resolution"
          type = "xsd:string" />

        <xsd:element name = "Latitude"
          sql:field = "Latitude"
          type = "xsd:decimal" />

        <xsd:element name = "Longitude"
          sql:field = "Longitude"
          type = "xsd:decimal" />

      </xsd:sequence>

      <xsd:attribute name = "CustomerID"
        sql:field = "CustomerID"
        type = "xsd:integer" />

    </xsd:complexType>

  </xsd:element>

</xsd:schema>

```

The destination table for the bulk load and the source data were shown in Listings 12-8 and 12-9.

Listing 12-8. *Sales.CustomerGeography Table*

```
CREATE TABLE Sales.CustomerGeography (
    CustomerID INT NOT NULL PRIMARY KEY,
    Latitude NUMERIC (15, 6),
    Longitude NUMERIC (15, 6),
    Resolution VARCHAR(20)
);
```

Listing 12-9. *SQLXML Bulk Load Source Data*

```
<?xml version = "1.0"?>
<ROOT xmlns:sql = "urn:schemas-microsoft-com:xml-sql">
    <CustomerCoordinates CustomerID = "1">
        <Resolution>best</Resolution>
        <Latitude>47.611744</Latitude>
        <Longitude>-122.347698</Longitude>
    </CustomerCoordinates>
    <CustomerCoordinates CustomerID = "2">
        <Resolution>best</Resolution>
        <Latitude>47.475590</Latitude>
        <Longitude>-122.204569</Longitude>
    </CustomerCoordinates>
    <CustomerCoordinates CustomerID = "3">
        <Resolution>best</Resolution>
        <Latitude>32.935160</Latitude>
        <Longitude>-97.017039</Longitude>
    </CustomerCoordinates>
    <CustomerCoordinates CustomerID = "4">
        <Resolution>good</Resolution>
        <Latitude>30.302679</Latitude>
        <Longitude>-97.761980</Longitude>
    </CustomerCoordinates>
    . . .
</ROOT>
```

The SQLXML Bulk Load function is provided by another COM interop wrapper class, which was demonstrated in Listing 12-10.

Listing 12-10. *Bulk Load Procedure*

```
[STAThread()]
private void BulkLoadData()
{
    SQLXMLBULKLOADLib.SQLXMLBulkLoad4Class bulkloader =
        new SQLXMLBULKLOADLib.SQLXMLBulkLoad4Class();
    bulkloader.ConnectionString = SqlXmlConnString;
    bulkloader.ErrorLogFile = "c:\\bulkloadererrors.xml";
    bulkloader.KeepIdentity = false;
    bulkloader.Transaction = false;
    bulkloader.Execute("c:\\AWCustGeo.xsd", "c:\\AWCustGeo.xml");
}
```


Index

Symbols

- // (double slash) axis step, 126, 148
- () empty sequence notation, 120
- / (forward slash) character, 375
- [] (hard brackets), 126
- ? (occurrence indicator), 156
- / (single slash) axis step, 126
- x command, 341–342

A

- absolute location paths, 394
- accessing
 - REST services, 256–261
 - XML documents from external sources, 253–256
 - XSLT through .NET Framework, 194–199
- Add Web Reference window, 241
- advanced transformation, performing with XSLT
 - multitemplate stylesheet, 221–226
 - overview of, 219–221
 - recursion in stylesheet, 226–228
- AdventureWorks 2008 database (Microsoft)
 - Bill of Materials (BOM), 47
 - customer summary report, 429–432
 - Dim Scenario dimension, 343–344
 - LINQ to SQL, using to populate LINQ to XML XElement, 325
 - overview of, 1
 - Product Browser client application, 232, 244
 - SQLXML sample query result, 296
- aggregate functions (XQuery), 162–163
- aliases for column names, 20
- all element (XML Schema), 366
- ALTER INDEX statement, 178, 187
- Altova software, XMLSpy, 350–351
- annotated mapping schema
 - SQLXML Bulk Load, 311
 - updategram, 298–303
- annotation element (XML Schema), 366
- annotations
 - definition of, 389
 - XSD and, 85
- anonymous delegates, 336–337
- anonymous types, 331
- <any> schema component, 102–107, 367

- anyAttribute element (XML Schema), 367
- anySimpleType data type, 108
- appInfo element (XML Schema), 367
- ApplyStylesheet function (SQLCLR), 197
- AS HTTP clause (CREATE ENDPOINT statement), 237
- AsGml() method, 276
- asynchronous processing, 272
- atomic data type, 389
- <attribute> element, 93–95, 367
- attribute nodes, 117, 389
- attribute-centric XML, 25
- <attributeGroup> element, 94–95, 368
- attributes
 - definition of, 389
 - HTML formatting and, 205
 - XSD and, 93–95
- author blog, 355
- AUTO mode, FOR XML clause
 - logging DML actions, 32–34
 - overview of, 29–31
- Average method, 336
- AWCustGeo.xml document, 312
- AWCustGeo.xsd document, 311
- axis specifiers
 - definition of, 389
 - XPath 1.0, 385–386
 - XQuery, 377
- axis steps
 - definition of, 125, 390
 - double (/), 126, 148
 - reverse, avoiding, 148
 - single (/), 126

B

- back-end transformation, performing with XSLT
 - Acme Retailers XSLT stylesheet listing, 212–215
 - fn_XsltTransform listing, 219
 - overview of, 208–209
 - XML orders from different sources listing, 209–211
 - XYZ Bike Repair XSLT stylesheet listing, 216–218
- “bang notation,” 34
- Base64, 390

- BATCHES option (CREATE ENDPOINT statement), 238
- BCP (Bulk Copy Program), 341
- BeginExecuteXmlReader method (SqlCommand class), 270–272
- BIDS (Business Intelligence Development Studio), 346–347
- Bill of Materials (BOM)
 - database and, 47–48
 - example with user-defined simple types, 110–112
 - hierarchical, listing for, 219–221
 - nested, with occurrence constraints, 96–99
 - recursive, query to generate, 99–101
 - result of recursive XSD applied to, 102
 - typed BOM schema collection, 108–109
- BINARY BASE64 option, FOR XML RAW
 - clause and, 28
- binary data type, 63
- binary formats, 10–11
- blog by author, 355
- BOM (Bill of Materials)
 - database and, 47–48
 - example with user-defined simple types, 110–112
 - hierarchical, listing for, 219–221
 - nested, occurrence constraints, 96–99
 - recursive, query to generate, 99–101
 - result of recursive XSD applied to, 102
 - typed BOM schema collection, 108–109
- Boolean function (XQuery), 160
- branching, 150–151
- built-in constructors, types with, 168
- Bulk Copy Program (BCP), 341
- BULK INSERT statement (T-SQL), 342
- Bulk Loading, 310–314, 443–444
- BulkLoadData procedure, 313
- Business Intelligence Development Studio (BIDS), 346–347
- BusinessEntityID 21000, 304
- byte order mark, 64

C

- cached XML query plans, querying, 143, 423–424
- calling xml data type methods, 73
- Cascading Style Sheets, 357
- CASE expression, 208
- case-sensitivity of method names, 279
- cast expression (XQuery), 156
- CAST function (T-SQL), 61, 167
- cast operator, 424
- casting
 - data types to xml data type, 63–65
 - SqlXml to XmlDocument
 - with CreateReader(), 267
 - with MemoryStream, 267
 - with XmlTextReader, 268

- XmlDocument to SqlXml with
 - MemoryStream, 268
- char data type, 63
- character entity references, 6
- character references, 6–8, 390
- checking for node existence, 142
- child sequences, 120
- <choice> component, 89–90, 368
- classes
 - model group definition compared to, 92
 - .NET Framework
 - System.Data.SqlClient.SqlCommand, 269–272, 307–309
 - System.Data.SqlTypes namespace, 266–269
 - System.Xml namespace, 261–266
 - XmlNodeReader, 264–266
 - System.Xml.XmlReader, 245–249
 - System.Xml.XmlReaderSettings, 249–251
 - XAttribute, 320
 - XDocument, 326–330
 - XElement, 319, 321, 325–326
 - XmlDocument, 261–264, 267–268
 - XmlNode, 261
 - XmlNodeList, 262
 - XslCompiledTransform, 381
- closed language, 390
- column names
 - aliases for, 20
 - FOR XML PATH clause and, 21–22
- column order, FOR XML PATH queries and, 22
- comma operator, 120, 155, 390
- Comma Separated Values (CSV) format, 8–10, 391
- command-line tools, Bulk Copy Program, 341–342
- comment() node test, 39
- comment nodes, 117, 390
- comments
 - XML, 6
 - XQuery, 133, 390
- Common Table Expression (CTE), FOR XML
 - clause and, 50
- comparison operators (XQuery), 129, 154
- complex elements, XSD and, 87–90
- complex queries, creating with FOR XML
 - clause, 47–54
- complex types, 108, 390
- complexContent element (XML Schema), 368
- complexType element (XML Schema), 368
- compound predicates, 391
- computed construction (XQuery), 422
- computed constructors (XQuery), 380
- computed element constructors, 136–139, 391

- conditional evaluation, 146–147
- conditional expressions (XQuery), 380
- constraining facets (XML Schema), 372
- constructor functions (XQuery), 168–169
- constructors
 - built-in, types with, 168
 - computed, 380
 - computed element, 136–139, 391
 - direct element, 136–139, 391
- consuming endpoints, 240–244
- CONTAINS predicate (T-SQL), 190–191
- contains predicate (XQuery), 191
- CONTENT facet (xml data type), 65
- context functions (XQuery), 166–167
- context item expression (XQuery), 391
- context nodes
 - definition of, 124, 391
 - XQuery, 79
- conversion modes, FOR XML clause, 17
- CONVERT function, 62, 409
- converting
 - data types to xml data type, 64
 - SqlXml to XmlDocument with NULL handling, 268
- CREATE ENDPOINT statement, 236–239
- CREATE FULLTEXT CATALOG statement, 189
- CREATE FULLTEXT INDEX statement, 189
- CREATE PRIMARY XML INDEX statement, 178
- CREATE TABLE statement, 66–68
- CREATE XML INDEX statement, 181–187
- CreateXmlReader function (SQLCLR), 197
- CreateXsltStyleSheet function (SQLCLR), 197
- cross-products, producing in XQuery, 136
- CSV (Comma Separated Values) format, 8–10, 391
- CTE (Common Table Expression), FOR XML clause and, 50
- customer summary report, 199–202, 429–432

D

- data accessor functions (XQuery), 158–159
- data connection, adding to project, 323
- data() node test, 41
- data type system (XDM), 117–119
- data types
 - See also specific data types*
 - atomic, list, and union, 389
 - XML Schema
 - constraining facets, 372
 - definition of, 399
 - XPath 1.0, 387
 - XQuery Data Model (XDM), 361–364
- database
 - See also* AdventureWorks 2008 database (Microsoft)
 - storing XML data in, 64
 - TRUSTWORTHY mode of, turning on, 194

- Database Tuning Advisor (DTA), 349–350
- date context functions, 167
- dbo.uspGetBillOfMaterials stored procedure call, 47
- decimal data type, 109
- declaration components, XSD and, 86–87
- declarations, 6, 391
- declaring multiple namespaces, 45
- default element namespace declaration, 122
- default function namespace declaration, 122
- defining inline DTDs, 68
- Delaware state borders, 438–439
- delete statement (XML DML), 174–175
- delete updategram, 306–307
- deploying SQLCLR assemblies, 195
- derived built-in types, 108
- descendant-or-self axis step, avoiding, 148–149
- designing XML schema in XMLSpy, 351
- destination table, Bulk Load, 444
- details template, 226
- diffgrams (SQLXML), 310
- Dim Scenario dimension, 343–344
- direct construction (XQuery), 422
- direct element constructors, 136–139, 391
- Directive options (FOR XML EXPLICIT clause), 35
- DOCTYPE declaration, xml data type instance and, 249
- DOCUMENT facet (xml data type), 65
- document nodes, 117, 392
- document orders, 392
- Document Type Definitions (DTDs), 6, 392
 - external, 393
 - inline, 68, 394, 409
 - LegalXML eContract, 248–249
 - loading external, 251
 - overview of, 245–246
 - validating XML documents against, 246–253, 435–436
- documentation element (XML Schema), 369
- documents
 - See also* XSLT (Extensible Stylesheet Language Transformations)
 - accessing from external source, 253–256
 - AWCustGeo.xml, 312
 - AWCustGeo.xsd, 311
 - definition of, 391
 - invalid, 436
 - valid, 398
 - well-formed
 - definition of, 8, 399
 - DOCUMENT facet and, 65
- XML
 - accessing from external source, 253–256
 - converted to node table format, 178
 - retrieving remotely, 436

- sample, 5–6
- valid type XML, 86, 88
- validating against DTDs, 246–253, 435–436
 - as XDM instance, 115
- double slash (//) axis step, 126, 148
- DROP INDEX statement, 178–179
- dropping table onto LINQ to SQL Classes designer surface, 324
- DTA (Database Tuning Advisor), 349–350
- DTDs (Document Type Definitions), 6, 392
 - external, 393
 - inline, 68, 394, 409
 - LegalXML eContract, 248–249
 - loading external, 251
 - overview of, 245
 - validating XML document against, 246–253, 435–436
 - xml data type and, 68–70
- DTSX file listings, 345–347
- dynamic context (XQuery), 129

E

- eBay Shopping Services Application
 - Programming Interface (API), 258–260
- EBCDIC (Extended Binary Coded Decimal Interchange Code), 392
- editing tools, XMLSpy, 350–351
- effective Boolean value, 160
- element element (XML Schema), 369
- element information items (XML Schema)
 - all element, 366
 - annotation element, 366
 - any element, 102–107, 367
 - anyAttribute element, 367
 - appInfo element, 367
 - attribute element, 93–95, 367
 - attributeGroup element, 94–95, 368
 - choice element, 89–90, 368
 - complexContent element, 368
 - complexType element, 368
 - documentation element, 369
 - element element, 369
 - extension element, 369
 - formatting conventions, 366
 - group element, 90–92, 370
 - import element, 370
 - list element, 370
 - notation element, 370
 - overview of, 365
 - restriction element, 110, 370
 - schema element, 371
 - sequence element, 89–90, 371
 - simpleContent element, 371
 - simpleType element, 110, 372
 - union element, 372
- element nodes, 117, 392
- elements
 - definition of, 392
 - GML and, 286–292
 - recursive, defining, 95
 - xsl:for-each, 206
 - xsl:if, 226
 - xsl:output, 215
 - xsl:preserve-space, 215
 - xsl:sort, 206
 - xsl:strip-space, 215
 - xsl:stylesheet, 203
 - XSLT 1.0, 381–383
- ELEMENTS option
 - FOR XML AUTO clause, 30
 - FOR XML RAW clause, 25
- ELEMENTS XSINIL option (FOR XML PATH clause), 22
- emoticon remark symbols for inline
 - documentation, 133
- empty sequences, 120, 392
- enabling SQLCLR, 194
- enclosed expressions, 136, 392
- EndExecuteXmlReader method
 - (SqlCommand class), 270–272
- endpoints, HTTP SOAP
 - consuming, 240–244
 - creating, 232–239, 432–433
 - security issues, 232, 238
 - support for, 15, 231
- entities, 392
- entitizing, 6
- eq (value comparison operator), 76
- error messages
 - attempting to validate invalid XML document against DTD, 253
 - HTTP SOAP endpoints and, 231
- ETL (Extract-Transform-Load) services, 345–347
- every (quantified expression), 132–133
- Excel spreadsheet binary data, 10
- ExecuteXmlReader method (SqlCommand class), 269, 321
- executing updategrams with
 - SqlXmlCommand class, 307–308
- exist() method (xml data type), 75–76, 142, 412
- existential quantifier, 132
- EXPLICIT mode (FOR XML clause), 34–37
- expressions
 - CASE, 208
 - definition of, 119, 393
 - enclosed, 136, 392
 - every, 132–133
 - filter, 393

- lambda, 336–337
 - primary, 119, 396
 - some, 132–133
 - XQuery
 - cast, 156
 - conditional, 380
 - content item, 391
 - FLWOR, 133–135, 379, 393, 421
 - instance of, 156–157
 - quantified, 132–133, 421
 - range, 126
 - type, 156–158
 - unimplemented, 158
 - value comparison operator, using, 420
 - expressive node construction functionality of XQuery, 136–139
 - Extended Binary Coded Decimal Interchange Code (EBCDIC), 392
 - extending XML schemas with wildcards, 102–107
 - Extensible HTML (XHTML), 8
 - Extensible HTML (XHTML) 1.0, 358
 - Extensible HTML (XHTML) standard, 205
 - Extensible Markup Language (XML)
 - attribute-centric, 25
 - character references, 6–8
 - choosing to use, 11–12
 - declarations, 6
 - description of, 2, 399
 - HTML compared to, 8
 - in SQL Server, history of, 1–2
 - terminology of. *See* Appendix F, Glossary
 - Extensible Markup Language (XML) 1.0, 3–4, 358
 - Extensible Markup Language (XML) 1.1, 358
 - Extensible Stylesheet Language (XSL), 400, 428–429
 - Extensible Stylesheet Language Transformations (XSLT). *See* XSLT
 - extension element (XML Schema), 369
 - extension functions (XQuery), 170–171
 - exterior bounding rings, 285–286
 - EXTERNAL ACCESS ASSEMBLY permissions, granting, 195
 - external DTDs, 393
 - Extract-Transform-Load services, 345–347
- F**
- F&O, 393
 - facets, 66, 393
 - Fielding, Roy, 256
 - file system, loading XML from, 326–327
 - files
 - command line
 - to bulk load, 342
 - to generate XML format, 341
 - DTA XML input, 349–350
 - DTSX, 345–347
 - loading
 - XDocument class from, 326
 - XML with OPENROWSET, 58
 - transforming and writing, 429
 - Unicode, 64
 - filter expressions, 393
 - filtering using numeric predicates, 127–128
 - Firefox (Mozilla), 352
 - FLWOR expressions
 - definition of, 393
 - XQuery, 133–135, 379
 - with where clause, 421
 - fn_GetRemoteXML user-defined function (SQLCLR), 254
 - fn_ValidateDTD function, 250–251
 - fn_XsltTransform (SQLCLR), 197, 219
 - for clause (XQuery), 133
 - FOR SOAP clause (CREATE ENDPOINT statement), 238–239
 - FOR XML AUTO query, code sample, 403
 - FOR XML clause
 - adding namespaces to, 45–47
 - AUTO mode
 - logging DML actions, 32–34
 - overview of, 29–31
 - complex queries, creating, 47–54
 - description of, 17, 393
 - enhancements to, 15–17
 - EXPLICIT mode, 34–37
 - OPENROWSET XML loading, 58
 - OPENXML rowset provider, 54–58
 - options, 18–19
 - PATH mode, 19–23
 - RAW mode, 23–29
 - ROOT option, 18
 - SELECT statement, 1
 - SQLXML, 298
 - TYPE option, 18, 53
 - usage scenarios, 18
 - WITH XMLNAMESPACES clause, 405
 - XMLDATA option, 18
 - XMLSCHEMA option, 18
 - FOR XML EXPLICIT clause
 - Directive options, 35
 - FOR XML PATH clause and, 18–19
 - support for, 17
 - Universal Table Format and, 398
 - FOR XML EXPLICIT query, code sample, 404
 - FOR XML PATH clause
 - customer summary report, 429, 432
 - note tests, 375
 - path expressions, 375
 - pipe-delimited list, creating, 44

- query example, 376
 - XPath node tests, 37–44, 404
 - FOR XML PATH query, code sample, 402
 - FOR XML query, nested, 405–407
 - FOR XML RAW query, code sample, 402–403
 - format command, 341–342
 - formats
 - See also* node table format
 - BCP XML, 342
 - binary, 10–11
 - Comma Separated Values (CSV), 8–10, 391
 - command line to generate XML, 341
 - nvarchar, 409
 - relational, 11
 - Unicode Transformation Format (UTF), 398
 - universal table, 34–35, 398
 - xml_obj.query(), 73
 - formatting conventions for XML Schema
 - element information items, 366
 - forward slash (/) character, 375
 - fragments
 - CONTENT facet and, 65
 - definition of, 393
 - FREETEXT predicate, 190, 427
 - full-text catalog creation, 426
 - full-text index on xml data type column, 426
 - full-text indexing, 189–191
 - full-text search compound predicate, 427
 - functional construction (LINQ to XML)
 - creating XML using, 319–320
 - transforming XML data via, 337–339
 - functions
 - ApplyStylesheet (SQLCLR), 197
 - CAST (T-SQL), 61, 167
 - CONVERT, 62, 409
 - date context, 167
 - fn:count(), 162
 - fn:empty() and fn:distinct-values(), 164
 - fn_GetRemoteXML, 254
 - fn:id(), 164–165
 - fn:last() and fn:position(), 166
 - fn:min() and fn:max(), 162
 - fn:sum() and fn:avg(), 163
 - fn_ValidateDTD, 250–251
 - fn_XsltTransform, 197, 219
 - GETDATE (T-SQL), 167
 - GetHtmlCatalogDescription, 234–236
 - GetHtmlProductDescription, 243
 - GetProductImage, 243
 - numeric, 424
 - OPENXML, 17
 - PopulateSearchTree, 242
 - sql:column, 170, 424
 - sql:variable, 167, 425
 - STUFF, 44
 - time context, 167
 - Transform, 195–197
 - web service method call private, 241–242, 433–435
 - XMLCAST, 65
 - XPath 1.0, 383, 385
 - XPath 2.0, 383
 - XQuery
 - aggregate, 162–163
 - Boolean, 160
 - constructor, 168–169
 - context, 166–167
 - data accessor, 158–159
 - extension, 170–171
 - node, 165
 - numeric, 161
 - overview of, 158
 - QName, 169–170
 - sequence, 163–165
 - sql:column, 424
 - sql:variable, 425
 - string, 159–160
 - user-defined, 162
 - XSL transformation (XSLT), 428–429
 - XSLT 1.0, 383–385
 - XSLT support, 427–428
 - Functions and Operators Recommendation
 - Web site, 393
- ## G
- general comparison operators, 130, 393
 - general comparisons, 393
 - geography data type
 - coordinates, 277
 - description of, 275–276
 - instance methods, 276
 - three-dimensional points and, 287
 - Geography Markup Language (GML)
 - borders for state of Delaware, 438–439
 - description of, 276–280
 - elements of, 286–292
 - geometric objects, 280–286
 - geometric objects, GML and, 280–286
 - geometry data type
 - coordinates, 277
 - description of, 275–276
 - instance methods, 276
 - three-dimensional points and, 287
 - GeometryCollection object (GML), 282
 - GeomFromGml() method, 278–279
 - GETDATE function (T-SQL), 167
 - GetHtmlCatalogDescription function, 234–236
 - GetHtmlProductDescription function, 243
 - GetProductHierarchy stored procedure, 233
 - GetProductImage function, 243
 - GetProductPhoto function, 233

GML (Geography Markup Language)
 borders for state of Delaware, 438–439
 description of, 276–280
 elements of, 286–292
 geometric objects, 280–286
<group> element, 90–92, 370

H

hard brackets ([]), 126
heterogeneous sequences, 155, 394
HTML, XML compared to, 8
HTML 4.01 specification, 205
HTTP, loading XML from web via, 329–330
HTTP SOAP endpoints
 consuming, 240–244
 creating, 232–239, 434–433
 security issues, 232, 238
 support for, 15, 231

I

IANA Language Subtag Registry, 360
idoc parameter (OPENXML rowset provider), 54
if . . . then . . . else construct, 146–147
import element (XML Schema), 370
indexing
 full-text, 189–191
 primary XML index, creating, 177–181
 secondary XML index, creating
 overview of, 181
 PATH type, 182–183
 PROPERTY type, 185–186
 types of, 187
 VALUE type, 183–185
 setting options, 187–188
InfoSet, 394
inline documentation (XQuery), 133
inline DTDs
 code sample, 409
 defining, 68
 definition of, 394
insert statement (XML DML), 171–173
insert updategram, 303–304, 441
instance methods, 71, 276
instance of expression (XQuery), 156–157
instances of xml data type, creating, 61–62, 68
integer data type, 109
interior bounding rings, 285–286
internal xml representation, 64
International Organization for Standardization
 (ISO) 8879 standard, 2
Internet Assigned Numbers Authority
 Language Subtag Registry, 360
Internet Explorer (IE, Microsoft), 353
invalid documents, 436
ISO (International Organization for
 Standardization) 8879 standard, 2

ISO SQL
 standards, 360
 2003 standard, 394
ISO SQL/XML standard, 394

J

joined tables FOR XML AUTO query, 29, 403

K

keywords
 var, 331
 XML DML, 80

L

lambda expressions, 336–337
Language-Integrated Query (LINQ), 319. *See also* LINQ to XML (XLinQ)
Large Object (LOB), 394
Large Object (LOB) data type. *See* xml data type
legacy OPENXML demonstration, 407–408
LegalXML eContracts version 1.0
 specification, 246–248
let clause (XQuery), 134
LineString element, 288
LineString object (GML), 282
LINQ (Language-Integrated Query), 319. *See also* LINQ to XML (XLinQ)
LINQ to SQL Classes item, querying with,
 323–326
LINQ to XML (XLinQ)
 description of, 319
 functional construction, 319–320
 loading XML
 from file system, 326–327
 from string, 327–328
 from web via HTTP, 329–330
 overview of, 321
 with XmlReader, 321–322
 namespaces, 332
 querying XML, 330–336
 standard query operators, 334
 transforming XML, 337–339
list data type, 113, 389, 419–420
list element (XML Schema), 370
listings
 AdventureWorks, XSLT customer
 summary report, 199–202
 altering XML index, 188
 base query of FOR XML hierarchical BOM,
 50
 BCP XML format file, 342
 BOM example with user-defined simple
 types, 110–112
 branched path expression comparison,
 150–151

- BULK INSERT statement (T-SQL), 342
- BulkLoadData procedure, 313
- cast operator, 424
- casting from XmlDocument to SqlXml
 - with MemoryStream, 268
- casting nvarchar data to xml
 - CAST function, using, 61
 - CONVERT function, using, 62
- casting SqlXml to XmlDocument
 - with CreateReader(), 267
 - with MemoryStream, 267
 - with XmlTextReader, 268
- command line
 - to bulk load file, 342
 - to generate XML format file, 341
- comment() node test, 39
- CREATE TABLE statement
 - with untyped xml, 66
 - XML schema collections and, 66–68
- C# source for fn_GetRemoteXML SQLCLR function, 254
- data() node test, 41
- dbo.uspGetBillOfMaterials stored procedure call, 47
- Delaware borders defined, 276
- delete updategram, 306–307
- descendant-or-self axis and wildcard in
 - the middle comparison, 148
- DTA XML input file, 349–350
- DTSX file, 345
- eBay REST service item search function, 259
- ExecuteXmlReader query, 269–270
- ExecuteXmlReader, populating XElement with, 321
- executing XmlReader asynchronously, 270–272
- exist() method, 75, 142
- explicit namespace URI, 46
- explicitly casting nvarchar data to xml, 409
- EXTERNAL ACCESS ASSEMBLY
 - permissions, granting, 195
- finished goods hierarchical BOM, 219–221
- fn_GetHtmlCatalogDescription function, 234–236
- fn_GetProductPhoto function, 233
- fn_ValidateDTD SQLCLR UDF, 250–251
- fn_XsltTransform transformations, 219
- fn_YahooGeocodeREST geocoding function, 257
- FOR XML AUTO query
 - code sample, 403
 - joined tables, 29
 - single table, 29
 - with ELEMENTS option, 30
- FOR XML clause
 - generating XML BOM using, 48–50
 - WITH XMLNAMESPACES clause, 405
- FOR XML EXPLICIT query, 35, 404
- FOR XML PATH query
 - code sample, 19, 402
 - customer summary report, 429–432
 - XPath node tests, 404
 - without ELEMENTS XSINIL option, 22
- FOR XML query, nested, 405–407
- FOR XML RAW query
 - code sample, 23, 402–403
 - with BINARY BASE64 option, 28
 - with ELEMENTS option, 25
 - with ROOT option, 24
 - with XMLDATA option, 28
 - with XMLSCHEMA option, 26
- full-text catalog creation, 189, 426
- full-text index on xml column, creating, 189, 426
- full-text search
 - combined with xml data type predicate, 191
 - compound predicate, 427
 - with FREETEXT predicate, 190
- geocoding Madison Square Garden, 258
- geometric objects, creating, 282–284
- GML
 - borders of state of Delaware, 438–439
 - Point definition for Mount McKinley, 286
- hierarchical annotated schema, 314
- HTML finished goods list, creating, 227
- HTTP SOAP endpoint, creating, 236–237, 432–433
- if . . then . . else construct, 146
- inline DTDs, 409
- insert updategram, 303, 441
- instances of xml data type, creating using DTD, 68
- invalid XML document validation against DTD, 252, 436
- legacy OPENXML demonstration, 407–408
- LegalXML eContract
 - example, 246–248
 - partial DTD, 248–249
 - validating against DTD, 252
- LineString element, 288
- LINQ to SQL, using to populate LINQ to XML XElement, 325
- LINQ to XML, transformation through functional construction, 337–338
- list data type, 113
- modify() method, xml data type, 79
- movie data
 - CSV format sample, 8
 - modified sample XML, 9–10
 - sample XML document, 5–6

- MultiLineString element, 289
- MultiPoint example with tallest US mountains, 287
- nested BOM with occurrence constraints, 96–99
- nested FOR XML hierarchical BOM, 51–53
- .NET XmlNodeReader class, 264–266
- .NET XPath example, 262–263
- node types in high-grossing movie sample, 401–402
- node() node test, 40
- nodes() method, 76–78, 143
- numeric function, 424
- OPENROWSET, loading xml variable via, 58, 408
- OPENXML query, 55
- PATH secondary XML index creation, 182
- Polygon element, 292
- populating geometry instance from GML, 278–279
- primary XML index creation, 179
- processing-instruction(name) node test, 42
- PROPERTY secondary XML index
 - creating, 185
 - efficiency sample, 186
- p_GetProductHierarchy stored procedure, 233
- queries
 - LINQ to XML with grouping and aggregation, 334–336
 - simple LINQ, 330
 - simple LINQ to XML, 331
- query() method, xml data type, 71–72
- querying cached XML query plans, 423–424
- recursive BOM, query to generate, 99–101
- retrieving
 - RSS feed, 255, 437
 - XML document remotely, 436
- reverse axis step comparison, 148
- sales order header, 261
- Sales.CustomerGeography table, 312
- searching eBay for *Star Wars*, 260
- shredding RSS feed, 255
- simple XQuery efficiency example, 179
- singleton atomic value query comparison, 147
- SQLCLR
 - enabling, 194
 - fn_XsltTransform, 197
 - p_XsltTransformToFile, 198–199
 - Transform function, 195–197
- SQLCLR UDF to validate XML against DTD, 435–436
- sql:column function, 424
- sql:variable function, 425
- SQLXML
 - query sample, 297, 439
 - XPath query sample, 315
- SQLXML Bulk Load
 - destination table, 444
 - mapping schema, 311, 443
 - procedure, 444
 - source data, 312, 444
- SqlXml to XmlDocument conversion with NULL handling, 268
- SqlXmlCommand updategram, 307–308, 442
- text() node test, 38
- transforming and writing file, 429
- TRUSTWORTHY mode of database, turning on, 194
- typed BOM schema collection, 108–109
- union data type, 112–113
- update updategram, 305
- updategram mapping schema, 298–300
- validating geometry instance data with STContains() method, 279–280
- VALUE secondary XML index
 - creating, 183
 - efficiency sample, 184
- value() method, 73, 141
- web service method call private functions, 241–242, 433–435
- WITH XMLNAMESPACES query, 45
- XDocument class, loading
 - from file, 326
 - from string, 327–328
 - from web, 329–330
- xml data type
 - exist() method, 412
 - modify() method, 414–415
 - nodes() method, 412–414
 - query() method, 410–411
 - value() method, 411–412
- XML DML
 - delete statement, 174
 - insert statement, 171
 - insert statement, using sql:variable() with, 173
 - modify() method, 145
 - replace value of statement, 175
- XML index
 - creating, 425
 - Path type secondary, 426
- xml interval representation vs. string representation, 64
- XML mapping schema, 440–441
- XML orders from different sources, 209–211
- XML query plan, 347–348
- XML schema namespace declaration, 84

- XML schemas
 - complex recursive bill of materials, 415–418
 - list data types, 419–420
 - root element with single element declaration, 86
 - single element declaration, 86
 - union data types, 418–419
 - with <any> component, 102–106
 - with attribute group definition component, 94–95
 - with attributes, 93–94
 - with complex type definition, 87–88
 - with complex type using <sequence> and <choice>, 89–90
 - with model group definitions, 90–92
 - XMLA script to create dimension, 343–344
 - XPath
 - path expressions sample, 375
 - query example, 376
 - XQuery
 - axis steps, 125
 - Boolean function, 160
 - cast expression, 156
 - compound predicate, 131
 - computed construction, 422
 - computed element constructor, 138
 - constructing XML with relational data, 170
 - constructor function, 168
 - data accessor function, 158
 - direct construction, 422
 - direct element constructor, 136
 - empty sequences, 120
 - expression using value comparison operator, 420
 - filtering using numeric predicates, 127–128
 - FLWOR expression code for, 133, 421
 - FLWOR expression direct element construction, 137
 - FLWOR with let clause, 134
 - FLWOR with order by clause, 135
 - FLWOR with where clause, 135
 - fn:count() aggregate function, 162
 - fn:empty() and fn:distinct-values() sequence functions, 164
 - fn:id() sequence function, 164–165
 - fn:last() and fn:position() context functions, 166
 - fn:min() and fn:max() aggregate functions, 162
 - fn:sum() and fn:avg() aggregate functions, 163
 - instance of expression, 157
 - node comparison operator, 130
 - numeric functions, 161
 - path expression, 123, 377
 - QName functions, 169
 - quantified expression, 132, 421
 - sequence construction, 379
 - sequence creation, 420
 - simple queries, 119
 - simple sequences, 120
 - string functions, 159
 - in subquery comparison, 149–150
 - value comparison operator, 129
 - XSD element, adding annotation element to, 85
 - XSL transformation function, 428–429
 - XSLT stylesheet
 - Acme Retailers, 212–215
 - recursive multitemplate, 221–226
 - XYZ Bike Repair, 216–218
 - XSLT support functions, 427–428
 - Yahoo! API geocoding service call, 437
 - Load method, XDocument class, 326–327
 - loading
 - Bulk Loading, 310–314, 443–444
 - external DTDs, 251
 - XML
 - from file system, 326–327
 - from string, 327–328
 - from web via HTTP, 329–330
 - overview of, 321
 - with XmlReader instance, 321–322
 - xml variable via OPENROWSET, 58, 408
 - LoadXml method (XmlDocument class), 263
 - LOB (Large Object), 394
 - LOB (Large Object) data type. *See* xml data type
 - location paths, 394
 - logging DML actions with FOR XML AUTO mode, 32–34
- ## M
- mapping schema
 - SQLXML Bulk Load, 311, 443
 - updategram, 298–303
 - XML, 440–441
 - math operators (XQuery), 154
 - maximizing XQuery performance
 - predicates in the middle, avoiding, 150–151
 - reverse axis steps, avoiding, 148
 - subqueries, using, 149–150
 - value() method, 147
 - wildcards in the middle and //, avoiding, 148–149
 - maxOccurs attribute, 95–102
 - methods
 - AsGml(), 276
 - Average, 336

- BeginExecuteXmlReader, 270–272
 - case-sensitivity of names of, 279
 - EndExecuteXmlReader, 270–272
 - ExecuteXmlReader, 269, 321
 - geometric objects, defining, 284
 - GeomFromGml(), 278–279
 - instance, 71, 276
 - Load, 326–327
 - LoadXml, 263
 - SelectNodes, 264
 - STContains(), 279–280
 - STGeomFromText(), 276
 - ToString(), 278
 - xml data type
 - exist(), 75–76, 142, 412
 - modify(), 79–80, 145–145, 414–415
 - nodes(), 54–57, 76–79, 142–145, 412–414
 - overview of, 71
 - query(), 71–73, 139–141, 410–411
 - value(), 73–75, 141, 147, 411–412
 - Microsoft
 - See also* AdventureWorks database; .NET Framework
 - Business Intelligence Development Studio, 346–347
 - CodePlex web site, 1
 - Internet Explorer (IE), 353
 - MSDN web site, 1
 - SQL Server 2008 Books Online, 360
 - SQLXML BulkLoad 4.0 Type Library COM object, adding reference to, 310
 - XDR schemas, 245
 - XML Core Services Library and OPENXML rowset provider, 57
 - Microsoft.Data.SqlXml namespace, adding reference to, 309
 - minOccurs attribute, 95–102
 - model group schema components, 87–88
 - model groups, XSD and, 90–92
 - modify() method (ml data type), 79–80, 145–146, 414–415
 - modifying XML with XML DML
 - delete statement, 174–175
 - insert statement, 171–173
 - replace value of statement, 175–176
 - Mozilla Firefox, 352
 - MultiCurve element, 289
 - MultiGeometry element, 292
 - MultiLineString element, 289
 - MultiLineString object (GML), 282
 - MultiPoint element, 287
 - MultiPoint object (GML), 282
 - MultiPolygon object (GML), 282, 285–286
 - multipurpose moves, 44
 - MultiSurface element, 292
 - multitemplate stylesheet, 221, 224, 226
- ## N
- namespace declaration, 122
 - namespace nodes, 117, 395
 - namespace prefixes, 395
 - namespace Uniform Resource Identifier (URI), 45
 - namespaces
 - adding to FOR XML clause, 45–47
 - declaring multiple, 45
 - definition of, 85, 394
 - LINQ to XML and, 332
 - Microsoft.Data.SqlXml, adding reference to, 309
 - .NET Framework XML support, 272–273
 - predeclared, 153
 - predefined, in XQuery, 122
 - System.Data.SqlTypes classes, 266–269
 - System.Xml, 245, 261–266
 - System.Xml.Linq, 320
 - System.Xml.Xsl, 194
 - XML schema, 84
 - in XML, W3C recommendations for, 358
 - nchar data type, 63
 - nesting
 - Bill of Materials, 96–99
 - FOR XML queries, 405–407
 - xml data type and, 99
 - .NET Framework
 - See also* LINQ to XML
 - accessing XSLT through, 194–199
 - XML classes
 - System.Data.SqlClient.SqlCommand class, 269–272
 - System.Data.SqlTypes namespace, 266–269
 - System.Xml namespace, 261–266
 - XML functionality and, 245
 - XML support namespace, 272–273
 - XML validation and, 245–253
 - .NET Framework Library, 266
 - node comparison operators, 130, 395
 - node comparisons, 395
 - node existence, checking for, 142
 - node functions (XQuery), 165
 - node() node test, 40
 - node order in sequences, 120
 - node table format
 - PATH secondary XML index on, 182
 - PK column, 178
 - PROPERTY secondary XML index on, 185
 - single XML document converted to, 178
 - VALUE secondary XML index on, 184

- node tests
 - comment(), 39
 - data(), 41
 - definition of, 124, 395
 - FOR XML PATH query, 404
 - node(), 40
 - processing-instruction(name), 42
 - text(), 38
 - XPath, 375
 - XPath 1.0, 37–41, 386
 - XQuery supported, 378
 - node types
 - in high-grossing movie code sample, 401–402
 - for instance of expression, 157
 - XPath 1.0, 387
 - XQuery/XDM, 377
 - nodes
 - attribute, 117, 389
 - checking for existence of, 142
 - comment, 117, 390
 - context, 79, 124, 391
 - definition of, 395
 - document, 117, 392
 - element, 117, 392
 - namespace, 117, 395
 - processing instruction, 117, 396
 - root, 396
 - text, 117, 398
 - nodes() method (xml data type), 54–57, 76–79, 142–145, 412–414
 - nonliteral storage, 119
 - normalizedString data type, 109
 - notation element (XML Schema), 370
 - NULL handling, SqlXml to XmlDocument conversion with, 268
 - numeric character references, 7–8
 - numeric function, 161, 424
 - numeric predicates, 127–128
 - nvarchar data type, 12, 63
 - nvarchar format, explicitly casting data to xml, 409
- O**
- OASIS (Organization for the Advancement of Structured Information Standards), 246
 - object.method() notation, 71
 - occurrence indicator (?), 156
 - occurrences, constraining, XSD and, 95–102
 - OGC (Open Geospatial Consortium), 276
 - OleDb/COM interface, SqlXmlCommand class and, 309
 - Open Geospatial Consortium (OGC), 276
 - OPENROWSET, loading xml variable via, 58, 408
 - OPENXML function, support for, 17
 - OPENXML legacy functionality
 - demonstration, 407–408
 - OPENXML rowset provider, 54–58, 76
 - operators
 - cast, 424
 - comma, 120, 155, 390
 - general comparison, 130, 393
 - node comparison, 395
 - standard query, and LINQ, 331–334
 - value comparison
 - definition of, 129, 399
 - eq, 76
 - expression using, 420
 - XPath 1.0, 386–387
 - XQuery
 - comma, 155
 - comparison, 129, 154
 - math, 154
 - overview of, 154, 378–379
 - optional occurrence indicator, 396
 - options, FOR XML clause, 18–19
 - order and sequences, 119
 - order by clause (XQuery), 135
 - Organization for the Advancement of Structured Information Standards (OASIS), 246
- P**
- parameters
 - OPENXML rowset provider, 54
 - xml, declaring, 65
 - path expressions
 - FOR XML PATH, 375
 - XQuery, 123–126, 377
 - PATH mode, FOR XML clause, 19–23
 - PATH secondary XML index, 182–183, 426
 - <pattern> schema component, 112
 - pessimistic validation, 128
 - pipe-delimited list, creating, 44
 - PK column (node table format), 178
 - Point element (GML), 286
 - Point object (GML), 282
 - Polygon element, 290–292
 - Polygon object (GML), 282–285
 - PopulateSearchTree function, 242
 - populating spatial data, 276–280
 - Post-Schema-Validation Infoset (PSVI), 84, 396
 - predeclared namespaces, 153
 - predefined namespaces, in XQuery, 122
 - predicates
 - definition of, 396
 - FREETEXT, 427
 - in the middle, avoiding to maximize XQuery performance, 150–151

- XQuery
 - comparison operators and, 129–131
 - compound, 131–132
 - overview of, 126–128
- preshrdding, 14
- primary expressions, 119, 396
- primary XML index, creating, 177–181
- processing instruction nodes, 117, 396
- processing instructions, 396
- processing-instruction(name) node test, 42
- product-number-type user-defined type, 112
- Production.ManuInstructionsSchemaCollection XML schema collection, 67–68
- prolog, defining, 121–122, 396
- property bag configuration, 185
- PROPERTY secondary XML index, 185–186
- protocols, XML for Analysis, 343–344
- PSVI (Post-Schema-Validation Infoset), 84, 396
- p_XsltTransformToFile (SQLCLR), 198–199, 227

Q

- QName functions (XQuery), 169–170
- quantified expression (XQuery), 132–133, 421
- queries
 - base, of FOR XML hierarchical BOM, 50
 - complex, creating with FOR XML clause, 47–54
 - ExecuteXmlReader, 269–270
 - FOR XML AUTO
 - code sample, 403
 - joined tables, 29
 - single table, 29
 - with ELEMENTS option, 30
 - FOR XML EXPLICIT, 35, 404
 - FOR XML PATH
 - code sample, 19, 376, 402
 - column order, 22
 - customer summary report, 429–432
 - XPath node tests, 404
 - without ELEMENTS XSINIL option, 22
 - FOR XML RAW
 - code sample, 24, 402–403
 - with BINARY BASE64 option, 28
 - with ELEMENTS option, 25
 - with ROOT option, 24
 - with XMLDATA option, 28
 - with XMLSCHEMA option, 26
 - to generate recursive BOM, 99–101
 - LINQ to XML, 334–336
 - nesting FOR XML, 405–407
 - OPENXML, 55
 - simple LINQ, 330
 - simple LINQ to XML, 331
 - WITH XMLNAMESPACES, 45
 - XQuery

- creating, 119–121
- prolog, creating, 121–122
- query plans, 143, 347–349, 423–424
- query() method (xml data type), 71–73, 139–141, 410–411
- querying
 - cached XML query plans, 143, 423–424
 - SQLXML with XPath, 314–317
 - xml data type and, 15
 - XML, LINQ to XML and, 330–336
 - with LINQ to SQL Classes item, 323–326
 - with SQLXML, 295–298

R

- range expressions, 126
- RAW mode, FOR XML clause, 23–29
- Really Simple Syndication (RSS), 253–256, 437
- recommendations, standards compared to, 3
- recursion in stylesheet, 226–228
- recursive Bill of Materials, query to generate, 99–101
- recursive element, defining, 95
- recursive multitemplate stylesheet, 221–226
- recursive nesting and XML schemas, 415–418
- regular expression syntax, 112
- relational format, 11
- relative location paths, 394
- replace value of statement (XML DML), 175–176
- Resource Description Framework (RDF) Site Summary, 253
- resources, regular expression syntax, 112
- REST (Representational State Transfer) services, 256–261
- <restriction> schema component, 110, 370
- retrieving
 - RSS feed, 255, 437
 - scalar values, 141
 - XML document remotely, 436
- return clause (XQuery), 133
- return types, xml, declaring, 65
- return values, exist() method, 75
- reverse axis steps, avoiding, 148
- RFC 2045 (Base64 system), 390
- root nodes, 396
- ROOT option
 - FOR XML clause, 18
 - FOR XML RAW clause, 24
- rowpattern parameter (OPENXML rowset provider), 54
- RSS (Really Simple Syndication), 253–256
- RSS feed, retrieving and shredding, 255, 437

S

- Sales.CustomerGeography table, 312
- scalar values, retrieving, 141

- schema collections, 397
- schema element (XML Schema), 371
- secondary XML index, creating
 - overview of, 181
 - PATH type, 182–183
 - PROPERTY type, 185–186
 - types of, 187
 - VALUE type, 183–185
- security issues with HTTP SOAP endpoints, 231–232, 238
- SELECT statement
 - FOR XML clause, 1
 - STUFF function, 44
- SelectNodes method (XmlDocument class), 264
- sequence construction (XQuery), 379
- <sequence> component, 89–90, 371
- sequence functions (XQuery), 163–165
- sequences
 - child, 120
 - comma operator and, 155
 - definition of, 119, 397
 - empty, 120, 392
 - simple, 120
 - singleton atomic values and, 121
 - XQuery, creating, 420
- Server Explorer window, viewing data
 - connections in, 323
- SGML (Standard Generalized Markup Language), 2, 397
- shredding
 - definition of, 14, 397
 - RSS feed, 255, 437
 - XML, 174
 - XML with nodes() method, 142–145
- simple content, 397
- Simple Object Access Protocol (SOAP), 397.
 - See also* SOAP endpoints over HTTP
- simple types, 108, 397
- simpleContent element (XML Schema), 371
- <simpleType> schema component, 110, 372
- single slash (/) axis step, 126
- single table FOR XML AUTO query, 29
- singleton atomic values, 117, 121
- SOAP (Simple Object Access Protocol), 397
- SOAP endpoints over HTTP
 - consuming endpoints, 240–244
 - creating endpoints, 232–239
 - security issues, 232
 - support for, 231
- some (quantified expression), 132–133
- source data, SQLXML Bulk Load, 312, 444
- spatial data
 - geometry and geography data types, 275–276
 - populating, 276–280
- sp_xml_preparedocument stored procedure, 56
- sp_xml_removedocument stored procedure, 57
- SQL Common Language Runtime. *See* SQLCLR (SQL Common Language Runtime)
- SQL Server, history of XML in, 1–2
- SQL Server 2008 XML, features of, 13–14
- SQL Server Analysis Service (SSAS), 343
- SQL Server Central, RSS feed, 253
- SQL Server Integration Services (SSIS), 345–347
- SQLCLR (SQL Common Language Runtime)
 - assemblies, deploying, 195
 - definition of, 398
 - enabling, 194
 - fn_GetRemoteXML user-defined function, 254
 - fn_XsltTransform, 197, 219
 - p_XsltTransformToFile, 198–199, 227
 - Transform function, 195–197
- SQLCLR UDF to validate XML against DTD, 435–436
- sql:column() function, 170, 424
- SqlCommand object, ExecuteXmlReader
 - method, 321
- .sqlplan XML query plan, 348
- sql:variable() function, 167, 425
- SQLXML
 - Bulk Load
 - destination table, 444
 - mapping schema, 443
 - overview of, 310–314
 - procedure, 444
 - source data, 444
 - diffgrams, 310
 - functionality available through, 295
 - query code, 439
 - querying with, 295–298
 - querying, with XPath, 314–317
 - updategrams
 - delete, 306–307
 - executing with SqlXmlCommand class, 307–308
 - insert, 303–304
 - overview of, 298–302
 - update, 305–306
- SqlXml data type, 197, 267–269
- SQLXMLBulkLoad4Class, 310–311
- SqlXmlCommand class
 - executing updategrams with, 307–308
 - OLEDB/COM interface and, 309
- SqlXmlCommand object, 295–298
- SqlXmlCommand updategram, 307–308, 442
- SSAS (SQL Server Analysis Service), 343
- SSIS (SQL Server Integration Services), 345–347

Standard Generalized Markup Language (SGML), 2, 397

standard query operators, and LINQ, 331–334

standards

- ISO 8879, 2
- ISO SQL, 360
- ISO SQL/XML, 394
- recommendations compared to, 3
- Unicode, 360
- XHTML, 205

statements

- ALTER INDEX, 178, 187
- BULK INSERT (T-SQL), 342
- CREATE ENDPOINT, 236–239
- CREATE FULLTEXT CATALOG, 189
- CREATE FULLTEXT INDEX, 189
- CREATE PRIMARY XML INDEX, 178
- CREATE TABLE, 66–68
- CREATE XML INDEX, 181–187
- delete (XML DML), 174–175
- DROP INDEX, 178–179
- insert (XML DML), 171–173
- replace value of (XML DML), 175–176
- SELECT
 - FOR XML clause, 1
 - STUFF function, 44
- XML DML, 380

static analysis (XQuery), 128–129

STContains() method, 279–280

steps, 398

STGeomFromText() method, 276

storage, nonliteral, 119

storing XML data in databases, 64

string, loading XML from, 327–328

string data type, 109

string functions (XQuery), 159–160

string xml representation, 64

STUFF function, 44

subqueries, using to maximize XQuery performance, 149–150

syntax of DTDs, 69–70

System.Data.SqlClient.SqlCommand class, 269–272

System.Data.SqlTypes namespace classes, 266–269

System.Data.SqlTypes.SqlXml data type, xml data type and, 65

System.Xml namespace, 245, 261–266

System.Xml.Linq namespace, 320

System.Xml.XmlReader class, 245–249

System.Xml.XmlReaderSettings class, 249–251

System.Xml.Xsl namespace, 194

T

T-SQL (Transact-SQL)

- BULK INSERT statement, 342
- CAST function, 61

- CONTAINS and FREETEXT predicates, 190–191
- CONVERT function, 62
- GETDATE and CAST functions, 167
- Table Valued Function [XML Reader with XPath filter] operation, 140
- table, dropping onto LINQ to SQL Classes designer surface, 324
- TCP (Transfer Control Protocol) endpoints, 232
- templates, 226, 398
- testing
 - See also* node tests
 - FOR XML PATH clause and XPath node tests, 37–44
 - XPath expression in XMLSpy, 351
- text() node test, 38
- text nodes, 117, 398
- Thomson, Jamie, 346
- three-dimensional points, 287
- time context functions, 167
- time zone and XQuery processor, 169
- tools
 - Bulk Copy Program, 341–342
 - Database Tuning Advisor, 349–350
 - SQL Server Integration Services, 345–347
 - Visual Studio, 354
 - web browsers, 352–353
 - XML for Analysis, 343–344
 - XML query plans, 347–349
 - XMLSpy, 350–351
- top-level schema components, 92
- ToString() method, 278
- Transact-SQL (T-SQL)
 - BULK INSERT statement, 342
 - CAST function, 61
 - CONTAINS and FREETEXT predicates, 190–191
 - CONVERT function, 62
 - GETDATE and CAST functions, 167
- Transfer Control Protocol (TCP) endpoints, 232
- Transform function (SQLCLR), 195–197
- transformation, performing. *See* advanced transformation, performing with XSLT; back-end transformation, performing with XSLT
- transforming
 - files, 429
 - XML, LINQ to XML and, 337–339
- TRUSTWORTHY mode of database, turning on, 194
- tuple stream, 134
- two-dimensional, spatial data as, 275–276
- type expressions (XQuery)
 - cast, 156
 - instance of, 156–157
 - unimplemented, 158
- TYPE option, FOR XML clause, 18, 53

- typed XML data, 66
- typed xml instance, 117
- types with built-in constructors, 168
- typing function of XML schemas, 108–113

U

- Unicode
 - definition of, 398
 - files, byte order mark, 64
 - standards, 360
- Unicode Transformation Format 8 (UTF-8), 398
- Unicode Transformation Format 16 (UTF-16), 398
- union data type, 112–113, 389, 418–419
- union element (XML Schema), 372
- unique particle attribution constraints, 107
- universal quantifier, 132
- Universal Resource Identifier (URI), 398
- universal table format, 34–35, 398
- untyped XML data, 66
- untyped xml instance, 117
- update updategram, 305–306
- updategrams (SQLXML)
 - delete, 306–307
 - insert, 303–304, 441
 - overview of, 298–302
 - SqlXmlCommand, 307–308, 442
 - update, 305–306
- URI (Universal Resource Identifier), 398
- USE PLAN query hint, 348
- user-defined functions (XQuery), 162
- user-defined simple data types, 110
- UTF-8 (Unicode Transformation Format 8), 398
- UTF-16 (Unicode Transformation Format 16), 398

V

- valid documents, 398
- valid typed XML document
 - applying complex type XML schema to, 88
 - simple XML schema and, 86
- validating XML document against DTD, 435–436
- validation of XML data, .NET Framework and, 245–253
- value comparison operators
 - definition of, 129, 399
 - eq, 76
 - expression using, 420
- value comparisons, 398
- VALUE secondary XML index, 183–185
- value() method (xml data type), 73–75, 141, 147, 411–412
- var keyword, 331
- varbinary data type, 63
- varchar data type, 12, 63

- verifying
 - delete updategram, 307
 - insert updategram, 305
 - SQLXML Bulk Load process, 313
 - update updategram, 306
- version declaration, 122
- Visual Studio, 240–241, 354

W

- W3C (World Wide Web Consortium)
 - description of, 2, 399
 - HTML4 and, 205
 - processes of, 357
 - Recommendations for XML Schema, 365–372
 - specifications of, 357–359
 - web site, 83
- web, loading XML from, via HTTP, 329–330
- web browsers, 352–353
- web reference, adding in Visual Studio, 240–241
- web service method call private functions, 241–242, 433–435
- web services, REST services and, 256
- Web Services Description Language (WSDL), HTTP SOAP endpoints and, 239
- web sites
 - author blog, 355
 - eBay Developers Program, 259
 - eBay Developers Program REST Developer Center, 260
 - Functions and Operators Recommendation, 393
 - IANA Language Subtag Registry, 360
 - ISO SQL standards, 360
 - LegalXML, 249
 - Microsoft
 - CodePlex, 1
 - MSDN, 1
 - SQL Server 2008 Books Online, 360
 - .NET Framework Library, 266
 - nodes, 395
 - RFC 2045 (Base64 system), 390
 - sample code, 1
 - SQL Server Central, 254
 - Unicode standards, 360
 - W3C, 83, 357–359
 - XMLSpy, 352
 - XQuery 1.0 and XPath 2.0 Data Model, 399
 - Yahoo! Geocoding API, 256
- well-formed documents
 - definition of, 8, 399
 - DOCUMENT facet and, 65
- well-known text (WKT) representation of
 - populating spatial data, 276–277
- where clause (XQuery), 135

- wildcards
 - extending XML schemas with, 102–107
 - in the middle, avoiding, 148–149
- WITH clause, OPENXML function and, 55
- WITH XMLNAMESPACES clause, 45–47, 405
- WKT (well-known text) representation of
 - populating spatial data, 276–277
- word-breakers, 189
- World Wide Web Consortium (W3C)
 - description of, 2, 399
 - HTML4 and, 205
 - processes of, 357
 - Recommendations for XML Schema, 365–372
 - specifications of, 357–359
 - web site, 83
- writing files, 429
- WSDL (Web Services Description Language),
 - HTTP SOAP endpoints and, 239

X

- x command, 341–342
- XAttribute class, 320
- XDM (XQuery/XPath Data Model)
 - data type system, 117–119, 123
 - data types, 361–364
 - description of, 359, 399
 - node types, 377
 - nodes mapped by, 117
 - nonliteral storage, 119
 - W3C specification for, 115–117
 - XML schema collections and, 83
- XDM instance, XML document as, 115
- XDocument class
 - Load method, 326–327
 - loading from string, 327–328
 - loading from web via HTTP, 329–330
- XDR (XML Data-Reduced) schemas, 245, 298
- XElement class, populating
 - functional construction and, 319
 - with ExecuteXmlReader method, 321
 - with LINQ to SQL, 325–326
- XHTML (Extensible HTML), 8
- XHTML (Extensible HTML) 1.0, 358
- XHTML (Extensible HTML) standard, 205
- XLinq (LINQ to XML)
 - description of, 319
 - functional construction, 319–320
 - loading XML
 - from file system, 326–327
 - from string, 327–328
 - from web via HTTP, 329–330
 - overview of, 321
 - with XmlReader, 321–322
 - namespaces, 332
 - querying XML, 330–336
 - standard query operators, 334
 - transforming XML, 337–339
- XML (Extensible Markup Language)
 - attribute-centric, 25
 - character references, 6–8
 - choosing to use, 11–12
 - declarations, 6
 - description of, 2, 399
 - HTML compared to, 8
 - in SQL Server, history of, 1–2
 - terminology of. *See* Appendix F, Glossary
- XML 1.0 Recommendation, 3–4, 358
- XML 1.1, 358
- XML data
 - forms of, 8
 - item types, 4
 - sample movies document, 5–6
- XML Data Manipulation Language (XML DML)
 - description of, 79–80, 399
 - modify() method and, 145
 - modifying XML with
 - delete statement, 174–175
 - insert statement, 171–173
 - replace value of statement, 175–176
 - performance and, 174
 - statements, 380
- xml data type
 - casting and converting, 63–65
 - CONVERT function and, 409
 - DOCTYPE declaration and, 249
 - DTDs and, 68–70
 - enhancements to, 14
 - explicitly casting nvarchar data to, 409
 - facets for, 65–66
 - full-text index on column of, 426
 - instances, creating, 61–62
 - methods
 - See also* indexing
 - exist(), 75–76, 142, 412
 - modify(), 79–80, 145–146, 414–415
 - nodes(), 54–57, 76–79, 142–145, 412–414
 - overview of, 71
 - query(), 71–73, 139–141, 410–411
 - value(), 73–75, 141, 411–412
 - nesting and, 99
 - new methods for querying and modifying
 - data, 15
 - OPENXML function compared to, 58
 - overview of, 61
 - parameters and return types, 65
 - XML Information Set and, 13
 - XML schema collections and, 66, 68
- XML Data-Reduced (XDR) schemas, 245, 298
- XML DML (XML Data Manipulation Language)
 - description of, 79–80, 399
 - modify() method and, 145

- modifying XML with
 - delete statement, 174–175
 - insert statement, 171–173
 - replace value of statement, 175–176
- performance and, 174
- statements, 380
- XML document
 - accessing from external source, 253–256
 - converted to node table format, 178
 - invalid, 436
 - retrieving remotely, 436
 - sample, 5–6
 - valid type, 86, 88
 - validating against DTDs, 246–253, 435–436
 - as XDM instance, 115
- XML for Analysis (XMLA), 343–344
- XML Fragment Interchange, 358
- XML indexes
 - creating, 425
 - definition of, 399
 - enhancements to, 14
 - PATH type secondary, 426
- XML indexing. *See* indexing
- XML Information Set, 358
- XML mapping schema, 440–441
- XML Path Language. *See* XPath (XML Path Language)
- XML Path Language Version 1.0, 358
- XML Query 1.0 Use Cases, 358
- XML Query language. *See* XDM (XQuery/XPath Data Model); XQuery language
- XML query plans, 347–349, 423–424
- XML Schema
 - components, 365
 - data type facets, 372
 - description of, 365
 - element information items
 - all element, 366
 - annotation element, 366
 - any element, 102–107, 367
 - anyAttribute element, 367
 - appInfo element, 367
 - attribute element, 93–95, 367
 - attributeGroup element, 94–95, 368
 - choice element, 89–90, 368
 - complexContent element, 368
 - complexType element, 368
 - documentation element, 369
 - element element, 369
 - extension element, 369
 - formatting conventions, 366
 - group element, 90–92, 370
 - import element, 370
 - list element, 370
 - notation element, 370
 - overview of, 365
 - restriction element, 110, 370
 - schema element, 89–90, 371
 - sequence element, 371
 - simpleContent element, 371
 - simpleType element, 110, 372
 - union element, 372
 - GML and, 276
- XML schema collections
 - enhancements to, 14
 - overview of, 8, 83
 - XDM and, 83
 - xml data type and, 66–68
- XML Schema data types, 399
- XML Schema Definition (XSD) language
 - annotations and, 85
 - attributes and, 93–95
 - complex elements and, 87–90
 - constraining occurrences and, 95–102
 - declaration components and, 86–87
 - DTDs, XDR schemas, and, 246
 - mapping schemas, 298–303
 - model groups and, 90–92
 - wildcards and, 102–107
 - XML schema namespace and, 84
 - XML schemas and, 84
- XML schema namespace, XSD and, 84
- XML Schema Part 0, Primer, Second Edition, 359
- XML Schema Part 1, Structures, Second Edition, 359
- XML Schema Part 2, Datatypes, Second Edition, 359
- XML Schema recommendation, 83
- XML schemas
 - See also* XML schema collections; XML Schema Definition (XSD) language
 - complex recursive bill of materials, 415–418
 - designing, in XMLSpy, 351
 - list data types, 419–420
 - typing function of, 108–113
 - union data types, 418–419
 - unique particle attribution constraints for, 107
 - XSD and, 84
- XMLA (XML for Analysis), 343–344
- XMLCAST function, 65
- XMLDATA option
 - FOR XML clause, 18
 - FOR XML RAW clause, 28
- XmlDocument class
 - casting SqlXml to
 - with CreateReader(), 267
 - with MemoryStream, 267
 - to SqlXml with MemoryStream, 268
 - with XmlTextReader, 268
 - LoadXml method, 263

- .NET Framework and, 261–262
- SelectNodes method, 264
- XmlNode class, 261
- XmlNodeList class, 262
- XmlNodeReader class, 264, 266
- XmlReader instance, loading XML with, 321–322
- XMLSCHEMA option
 - FOR XML clause, 18
 - FOR XML RAW clause, 26
- XMLSpy (Altova software), 350–351
- xml_obj.query() format, 73
- XNamespace object, 332
- XPath (XML Path Language)
 - See also* XDM (XQuery/XPath Data Model)
 - description of, 399
 - FOR XML PATH query example, 376
 - path expressions, 375
 - querying SQLXML with, 314–317
 - supported node tests, 375
 - XSLT and, 194
- XPath 1.0
 - axis specifiers, 385–386
 - data and node types, 387
 - expression, testing, in XMLSpy, 351
 - functions, 383, 385
 - node tests, 37–44, 386
 - operators, 386–387
 - syntax, XQuery syntax compared to, 123
- XPath 2.0 functions, and XSLT 2.0, 383
- XQuery (XML Query Language)
 - See also* XDM (XQuery/XPath Data Model); XQuery performance, maximizing
 - axis specifiers, 377
 - computed construction, 422
 - computed constructors, 380
 - conditional evaluation, 146–147
 - conditional expressions, 380
 - context node, 79, 124
 - cross-products, 136
 - description of, 115, 400
 - direct construction, 422
 - DML operations and, 171
 - exist() method and, 75–76
 - expressions
 - cast, 156
 - conditional, 380
 - content item, 391
 - FLWOR, 133–135, 379, 393, 421
 - instance of, 156–157
 - quantified, 132–133, 421
 - range, 126
 - type, 156–158
 - unimplemented, 158
 - value comparison operator, using, 420
 - expressive node construction
 - functionality of, 136–139
 - functions
 - aggregate, 162–163
 - Boolean, 160
 - constructor, 168–169
 - context, 166–167
 - data accessor, 158–159
 - extension, 170–171
 - node, 165
 - numeric, 161, 424
 - overview of, 158
 - QName, 169–170
 - sequence, 163–165
 - sql:column, 424
 - sql:variable, 425
 - string, 159–160
 - user-defined, 162
 - implementation of, 115
 - inline documentation, 133
 - node tests, 124, 378
 - node types, 377
 - nodes() method and, 78
 - operators
 - cast, 424
 - comma, 129, 155
 - comparison, 154
 - math, 154, 378–379
 - overview of, 154
 - path expressions, 123–126, 377
 - predefined namespaces, 122
 - predicates
 - comparison operators and, 129–131
 - compound, 131–132
 - overview of, 126–128
 - queries
 - creating, 119–121
 - prolog, creating, 121–122
 - query() method and, 73
 - sequence construction, 379
 - sequence creation, 420
 - static analysis, 128–129
 - type expressions
 - cast, 156
 - instance of, 156–157
 - unimplemented, 158
 - value() method and, 74
 - xml data type methods and, 71–73
 - XML DML and, 79–80
- XQuery 1.0, 359
- XQuery performance, maximizing
 - predicates in the middle, avoiding, 150–151
 - reverse axis steps, avoiding, 148
 - subqueries, using, 149–150
 - value() method, 147

- wildcards in the middle and //, avoiding, 148–149
 - XQuery plan
 - with no XML index, 180
 - with PATH secondary XML index, 182
 - with primary XML index, 180
 - with PROPERTY secondary XML index, 186
 - with VALUE secondary XML index, 184
 - XQuery processor and time zone, 169
 - XQuery Update Facility, W3C working draft for, 359
 - XQuery/XPath Data Model (XDM)
 - data type system, 117–119, 123
 - data types, 361–364
 - description of, 359, 399
 - node types, 377
 - nodes mapped by, 117
 - nonliteral storage, 119
 - W3C specification for, 115–117
 - XML schema collections and, 83
 - XSD (XML Schema Definition) language
 - annotations and, 85
 - attributes and, 93–95
 - complex elements and, 87–90
 - constraining occurrences and, 95–102
 - declaration components and, 86–87
 - DTDs, XDR schemas, and, 246
 - mapping schemas, 298–303
 - model groups and, 90–92
 - wildcards and, 102–107
 - XML schema namespace and, 84
 - XML schemas and, 84
 - xs:dateTime instance, 169
 - XSL (Extensible Stylesheet Language), 400, 428–429
 - xsl:attribute element, 215
 - xsl:call-template element, 226
 - xsl:choose element, 207–208
 - xsl:comment element, 215
 - XslCompiledTransform class, XSLT 1.0 and, 381
 - XslCompiledTransform Net 2.0 class, 197
 - xsl:for-each element, 206
 - xsl:if element, 226
 - xsl:output element, 215
 - xsl:preserve-space element, 215
 - xsl:sort element, 206
 - xsl:strip-space element, 215
 - xsl:stylesheet element, 203
 - XSLT (Extensible Stylesheet Language Transformations)
 - accessing through .NET, 194–199
 - advanced transformation, performing
 - multitemplate stylesheet, 221–226
 - overview of, 219–221
 - recursion in stylesheet, 226–228
 - back-end transformation, performing, 208–219
 - description of, 400
 - processing overview, 193–194
 - simple transformation, performing, 199–202
 - stylesheet elements, 203–208
 - support functions, 427–428
 - W3C recommendations for, 359
 - XSLT 1.0
 - elements, 381–383
 - functions, 383–385
 - XslCompiledTransform class and, 381
 - XSLT 2.0, XPath 2.0 functions and, 383
 - xsl:template element, 204
 - xsl:value-of element, 205
 - xsl:with-param element, 226
- ## Y
- Yahoo! API geocoding service call, 437
 - Yahoo! Geocoding Application Programming Interface (API), 256–258