Microsoft®

Programming Microsoft® SQL Server® 2012

Leonard G. Lobel
Andrew J. Brust

TALLAN

Programming Microsoft® SQL Server® 2012

Your essential guide to key programming features in Microsoft SQL Server 2012

Take your database programming skills to a new level—and build customized applications using the developer tools introduced with SQL Server 2012. This hands-on reference shows you how to design, test, and deploy SQL Server databases through tutorials, practical examples, and code samples. If you're an experienced SQL Server developer, this book is a must-read for learning how to design and build effective SQL Server 2012 applications.

Discover how to:

- Build and deploy databases using the SQL Server Data Tools IDE
- Query and manipulate complex data with powerful Transact-SQL enhancements
- Integrate non-relational features, including native file streaming and geospatial data types
- Consume data with Microsoft ADO.NET, LINQ, and Entity Framework
- Deliver data using Windows® Communication Foundation (WCF) Data Services and WCF RIA Services
- Move your database to the cloud with Windows Azure™ SQL Database
- Develop Windows Phone cloud applications using SQL Data Sync
- Use SQL Server BI components, including xVelocity in-memory technologies

Get code samples on the web
Ready to download at
http://go.microsoft.com/fwlink/?Linkid=252994

*For **system requirements**, see the Introduction.*

About the Authors

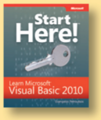
Leonard Lobel, MVP for SQL Server and a .NET specialist, is a principal consultant at Tallan, Inc., as well as the cofounder and CTO of Sleek Technologies, Inc.

Andrew Brust is the founder and CEO of Blue Badge Insights, an analyst, strategy, and advisory firm that helps customers navigate the Microsoft technology stack.

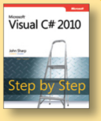
Microsoft® Most Valuable Professional

DEVELOPER ROADMAP

Start Here!
- Beginner-level instruction
- Easy to follow explanations and examples
- Exercises to build your first projects

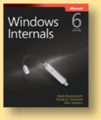


Step by Step
- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook



Developer Reference
- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples

Focused Topics
- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples



microsoft.com/mspress

ISBN: 978-0-7356-5822-6
9 780735 658226 90000

U.S.A. $59.99
Canada $62.99
[*Recommended*]

Programming/
Microsoft SQL Server

Microsoft®

Microsoft®

Programming Microsoft® SQL Server® 2012

Leonard Lobel
Andrew Brust

ISBN: 978-0-7356-5822-6

1 2 3 4 5 6 7 8 9 M 7 6 5 4 3 2

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://www.microsoft.com/learning/booksurvey.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation,
nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Russell Jones

Developmental Editor: Russell Jones

Production Editor: Melanie Yarbrough

Editorial Production: Christian Holdener, S4Carlisle Publishing Services

Technical Reviewer: John Paul Meuller

Copyeditor: Andrew Jones

Indexer: WordCo Indexing Services

Cover Design: Twist Creative • Seattle

Cover Composition: ContentWorks, Inc.

Illustrator: Rebecca Demarest

To my partner, Mark, and our children, Adam, Jacqueline, Joshua, and Sonny. With all my love, I thank you guys, for all of yours.

— Leonard Lobel

For my three boys: Miles, Sean, and Aidan. And for my sister, Valerie Hope.

— Andrew Brust

Contents at a Glance

Contents

PART I CORE SQL SERVER DEVELOPMENT

Chapter 2 T-SQL Enhancements 45

PART III APPLIED SQL

Chapter 12 Moving to the Cloud with SQL Azure 579

Chapter 14 Pervasive Insight 675

Chapter 15 xVelocity In-Memory Technologies 701

Introduction

—Leonard Lobel

Welcome! This is a book about Microsoft SQL Server 2012 written just for you, the developer. Whether you are programming against SQL Server directly at the database level or further up the stack using Microsoft .NET, this book shows you the way.

The latest release of Microsoft's flagship database product delivers an unprecedented, highly scalable data platform capable of handling the most demanding tasks and workloads. As with every release, SQL Server 2012 adds many new features and enhancements for developers, administrators, and (increasingly) end users alike. Collectively, these product enhancements reinforce—and advance—SQL Server's position as a prominent contender in the industry. As the product continues to evolve, its stack of offerings continues to expand. And as the complete SQL Server stack is too large for any one book to cover effectively, our emphasis in *this* book is on *programmability*. Specifically, we explore the plethora of ways in which SQL Server (and its cloud cousin, Microsoft SQL Azure) can be programmed for building custom applications and services.

How Significant Is the SQL Server 2012 Release?

SQL Server, particularly its relational database engine, matured quite some time ago. So the "significance" of every new release over recent years can be viewed—in some ways—as relatively nominal. The last watershed release of the product was actually SQL Server 2005, which was when the relational engine (that, for years, *defined* SQL Server) stopped occupying "center stage," and instead took its position alongside a set of services that today, collectively, define the product. These include the Business Intelligence (BI) components Reporting Services, Analysis Services, and Integration Services—features that began appearing as early as 1999 but, prior to SQL Server 2005, were integrated sporadically as a patchwork of loosely coupled add-ons, wizards, and management consoles. SQL Server 2005 changed all that with a complete overhaul. For the first time, the overall SQL Server product delivered a broader, richer, and more consolidated set of features and services which are built into—rather than bolted onto—the platform. None of the product versions that have been released since that time—SQL Server 2008, 2008 R2, and now 2012—have changed underlying architecture this radically.

That said, each SQL Server release continues to advance itself in vitally significant ways. SQL Server 2008 (released August 6, 2008) added a host of new features to the relational engine—T-SQL enhancements, Change Data Capture (CDC), Transparent Data Encryption (TDE), SQL Audit, FILESTREAM—plus powerful BI capabilities with Excel PivotTables, charts, and *CUBE* formulas. SQL Server 2008 R2 (released April 21, 2010), internally dubbed the "BI Refresh" while in development, added a revamped version of Reporting Services as well as PowerPivot for Excel and SharePoint, Master Data Services, and StreamInsight, but offered little more than minor tweaks and fixes to the relational engine.

The newest release—SQL Server 2012—officially launched on March 7, 2012. Like every new release, this version improves on all of the key "abilities" (availability, scalability, manageability, programmability, and so on). Among the chief reliability improvements is the new High Availability Disaster Recovery (HADR) alternative to database mirroring. HADR (also commonly known as "Always On") utilizes multiple secondary servers in an "availability group" for scale-out read-only operations (rather than forcing them to sit idle, just waiting for a failover to occur). Multisubnet failover clustering is another notable new manageability feature.

SQL Server 2012 adds many new features to the relational engine, most of which are covered in this book. There are powerful T-SQL extensions, most notably the windowing enhancements, plus 22 new T-SQL functions, improved error handling, server-side paging, sequence generators, rich metadata discovery techniques, and contained databases. There are also remarkable improvements for unstructured data, such as the FileTable abstraction over FILESTREAM and the Windows file system API, full-text property searching, and Statistical Semantic Search. Spatial support gets a big boost as well, with support for circular data, full-globe support, increased performance, and greater parity between the *geometry* and *geography* data types. And new "columnstore" technology drastically increases performance of extremely large cubes (xVelocity for PowerPivot and Analysis Services) and data warehouses (using an xVelocity-like implementation in the relational engine).

The aforementioned relational engine features are impressive, but still amount to little more than "additives" over an already established database platform. A new release needs more than just extra icing on the cake for customers to perceive an upgrade as compelling. To that end, Microsoft has invested heavily in BI with SQL Server 2012, and the effort shows. The BI portion of the stack has been expanded greatly, delivering key advances in "pervasive insight." This includes major updates to the product's analytics, data visualization (such as self-service reporting with Power View), and master data management capabilities, as well Data Quality Services (DQS), a brand new data quality engine. There is also a new Business Intelligence edition of the product that includes all of these capabilities without requiring a full Enterprise edition license. Finally, SQL Server Data Tools (SSDT) brings brand new database tooling inside Visual Studio. SSDT provides a declarative, model-based design-time experience for developing databases while connected, offline, on-premise, or in the cloud.

Who Should Read This Book

This book is intended for developers who have a basic knowledge of relational database terms and principles.

Assumptions

In tailoring the content of this book, there are a few assumptions that we make about you. First, we expect that you are a developer who is already knowledgeable about relational database concepts—whether that experience is with SQL Server or non-Microsoft platforms. As such, you already know about tables, views, primary and foreign keys (relationships), stored procedures, user-defined functions, and triggers. These essentials are assumed knowledge and are not covered in this book. Similarly, we don't explain proper relational design, rules of data normalization, strategic indexing practices, how to express basic queries, and other relational fundamentals. We also assume that you have at least basic familiarity with SQL statement syntax—again, either T-SQL in SQL Server or SQL dialects in other platforms—and have a basic working knowledge of .NET programming in C# on the client.

Having said all that, we have a fairly liberal policy regarding these prerequisites. For example, if you've only dabbled with T-SQL or you're more comfortable with Microsoft Visual Basic .NET than C#, that's okay, as long as you're willing to try and pick up on things as you read along. Most of our code samples are not that complex. However, our explanations assume some basic knowledge on your part, and you might need to do a little research if you lack the experience.

Note For the sake of consistency, all the .NET code in this book is written in C#. However, this book is in no way C#-oriented, and there is certainly nothing C#-specific in the .NET code provided. As we just stated, the code samples are not very complex, and if you are more experienced with Visual Basic .NET than you are with C#, you should have no trouble translating the C# code to Visual Basic .NET on the fly as you read it.

With that baseline established, our approach has been to add value to the SQL Server documentation by providing a developer-oriented investigation of its features, especially the new and improved features in SQL Server 2012. We start with the brand new database tooling, and the many rich extensions made to T-SQL and the relational database engine. Then we move on to wider spaces, such as native file streaming, geospatial data, and other types of unstructured data. We also have chapters on security, transactions, client data access, security, mobile/cloud development, and more.

Within these chapters, you will find detailed coverage of the latest and most important SQL Server programming features. You will attain practical knowledge and technical understanding across the product's numerous programmability points, empowering you to develop the most sophisticated database solutions for your end users. Conversely, this is not intended as a resource for system administrators, database administrators, project managers, or end users. Our general rule of thumb is that we don't discuss features that are not particularly programmable.

Who Should Not Read This Book

This book is not intended for SQL Server administrators; it is aimed squarely at developers—and only developers who have mastery of basic database concepts.

Organization of This Book

The chapters of this book are organized in three sections:

- Core SQL Server features
- Beyond relational features
- Applied SQL for building applications and services

By no means does this book need to be read in any particular order. Read it from start to finish if you want, or jump right in to just those chapters that suit your needs or pique your interests. Either way, you'll find the practical guidance you need to get your job done.

The following overview provides a summary of these sections and their chapters. After the overview, you will find information about the book's companion website, from which you can download code samples and work hands-on with all the examples in the book.

Core SQL Server Development

In Part I, we focus on core SQL Server features. These include brand new tooling (SSDT), enhancements to T-SQL, extended programmability with SQL CLR code in .NET languages such as Microsoft Visual Basic .NET and C#, transactions, and security.

- Chapter 1 Introducing SQL Server Data Tools

  Our opening chapter is all about SQL Server Data Tools (SSDT). With the release of SQL Server 2012, SSDT now serves as your primary development environment for building SQL Server applications. While SQL Server Management Studio (SSMS) continues to serve as the primary tool for database administrators, SSDT represents a brand new developer experience. SSDT plugs in to Microsoft Visual Studio for connected development of on-premise databases or SQL Azure databases running in the cloud, as well as a new database project type for offline development and deployment. Using practical, real-world scenarios, you will also learn how to leverage SSDT features such as code navigation, IntelliSense, refactoring, schema compare, and more.

- Chapter 2 T-SQL Enhancements

  In Chapter 2, we explore the significant enhancements made to Transact-SQL (T-SQL)—which still remains the best programming tool for custom SQL Server development. We cover several powerful extensions to T-SQL added in SQL Server 2008, beginning with table-valued parameters (TVPs). You learn how to pass entire sets of rows around between stored procedures and functions on the server, as well as between client and server using Microsoft ADO.NET. Date and time features are explored next, including separate date and time data types, time zone awareness, and improvements in date and time range, storage, and precision. We then show many ways to use *MERGE*, a flexible data manipulation language (DML) statement that encapsulates all the individual operations typically involved in any merge scenario. From there, you learn about INSERT OVER DML for enhanced change data capture from the *OUTPUT* clause of any DML statement. We also examine *GROUPING SETS*, an extension to the traditional *GROUP BY* clause that increases your options for slicing and dicing data in aggregation queries.

  We then dive in to the new T-SQL enhancements introduced in SQL Server 2012, starting with *windowing* features. The first windowing functions to appear in T-SQL date back to SQL Server 2005, with the introduction of several ranking functions. Windowing capabilities have been quite limited ever since, but SQL Server 2012 finally delivers some major improvements to change all that. First you will grasp windowing concepts and the principles behind the *OVER* clause, and then leverage that knowledge to calculate running and sliding aggregates and perform other analytic calculations. You will learn about every one of the 22 new T-SQL functions, including 8 analytic windowing functions, 3 conversion functions, 7 date and time related functions, 2 logical functions, and 2 string functions. We also examine improved error handling with *THROW*, server-side paging with *OFFSET/FETCH NEXT*, sequence generators, and rich metadata discovery techniques. We explain all of these new functions and features, and provide clear code samples demonstrating their use.

- Chapter 3 Exploring SQL CLR

  Chapter 3 provides thorough coverage of SQL CLR programming—which lets you run compiled .NET code on SQL Server—as well as guidance on when and where you should put it to use. We go beyond mere stored procedures, triggers, and functions to explain and demonstrate the creation of CLR types and aggregates—entities that cannot be created *at all* in T-SQL. We also cover the different methods of creating SQL CLR objects in SQL Server Database Projects in Visual Studio and how to manage their deployment, both from SSDT/Visual Studio and from T-SQL scripts in SQL Server Management Studio and elsewhere.

- Chapter 4 Working with Transactions

  No matter how you write and package your code, you must keep your data consistent to ensure its integrity. The key to consistency is transactions, which we cover in Chapter 4. Transactions can be managed from a variety of places, like many SQL Server programmability features. If you are writing T-SQL code or client code using the ADO.NET *SqlClient* provider or *System.Transactions*, you need to be aware of the various transaction isolation levels supported by SQL Server, the appropriate scope of your transactions, and best practices for writing transactional code. This chapter gets you there.

- Chapter 5 SQL Server Security

  Chapter 5 discusses SQL Server security at length and examines your choices for keeping data safe and secure from prying eyes and malicious intent. We begin with the basic security concepts concerning logins, users, roles, authentication, and authorization. You then go on to learn about key-based encryption support, which protects your data both while in transit and at rest. We then examine other powerful security features, including Transparent Data Encryption (TDE) and SQL Server Audit. TDE allows you to encrypt entire databases in the background without special coding requirements. With SQL

Server Audit, virtually any action taken by any user can be recorded for auditing in either the file system or the Windows event log. We also show how to create *contained databases*, a new feature in SQL Server 2012 that eliminates host instance dependencies by storing login credentials directly in the database. The chapter concludes by providing crucial guidance for adhering to best practices and avoiding common security pitfalls.

Going Beyond Relational

With the release of SQL Server 2012, Microsoft broadens support for semi-structured and unstructured data in the relational database. In Part II, we show how to leverage the "beyond relational" capabilities in SQL Server 2012—features that are becoming increasingly critical in today's world of binary proliferation, and the emergence of high-performance so-called "No SQL" database platforms.

- Chapter 6 XML and the Relational Database

  SQL Server 2005 introduced the *xml* data type, and a lot of rich XML support to go along with it. That innovation was an immeasurable improvement over the use of plain *varchar* or *text* columns to hold strings of XML (which was common in earlier versions of SQL Server), and thus revolutionized the storage of XML in the relational database. It empowers the development of database applications that work with hierarchical data *natively*—within the environment of the relational database system—something not possible using ordinary string columns. In Chapter 6, we take a deep dive into the *xml* data type, XQuery extensions to T-SQL, server-side XML Schema Definition (XSD) collections, XML column indexing, and many more XML features.

- Chapter 7 Hierarchical Data and the Relational Database

  But XML is not your only option for working with hierarchical data in the database. In Chapter 7, we explore the *hierarchyid* data type that enables you to cast a hierarchical structure over any relational table. This data type is implemented as a "system CLR" type, which is nothing more really than a SQL CLR user-defined type (UDT), just like the ones we show how to create on your own in Chapter 3. The value stored in a *hierarchyid* data type encodes the complete path of any given node in the tree structure, from the root down to the specific ordinal position among other sibling nodes sharing the same parent. Using methods provided by this new type, you can now efficiently build, query, and manipulate tree-structured data in your relational tables. This data type also plays an important role in SQL Server's new FileTable feature, as we explain in the next chapter on native file streaming.

- Chapter 8 Native File Streaming

  In Chapter 8, you learn all about the FILESTREAM, an innovative feature that integrates the relational database engine with the NTFS file system to provide highly efficient storage and management of large binary objects (BLOBs)—images, videos, documents, you name it. Before FILESTREAM, you had to choose between storing BLOB data in the database using *varbinary(max)* (or the now-deprecated *image*) columns, or outside the database as unstructured binary streams (typically, as files in the file system). FILESTREAM provides a powerful abstraction layer that lets you treat BLOB data logically as an integral part of the database, while SQL Server stores the BLOB data physically separate from the database in the NTFS file system behind the scenes. You will learn everything you need to program against FILESTREAM, using both T-SQL and the high-performance *SqlFileStream* .NET class. The walkthroughs in this chapter build Windows, web, and Windows Presentation Foundation (WPF) applications that use FILESTREAM for BLOB data storage.

  You will also see how FileTable, a new feature in SQL Server 2012, builds on FILESTREAM. FileTable combines FILESTREAM with the *hierarchyid* (covered in Chapter 7) and the Windows file system API, taking database BLOB management to new levels. As implied by the two words joined together in its name, one FileTable functions as two distinct things simultaneously: a table and a file system—and you will learn how to exploit this new capability from both angles.

- Chapter 9 Geospatial Support

  Chapter 9 explores the world of geospatial concepts and the rich spatial support provided by the *geometry* and *geography* data types. With these system CLR types, it is very easy to integrate location-awareness into your applications—at the database level. Respectively, *geometry* and *geography* enable spatial development against the two basic geospatial surface models: planar (flat) and geodetic (round-earth). With spatial data (represented by geometric or geographic coordinates) stored in these data types, you can determine intersections and calculate length, area, and distance measurements against that data.

  The chapter first quickly covers the basics and then provides walkthroughs in which you build several geospatial database applications, including one that integrates mapping with Microsoft Bing Maps. We also examine the significant spatial enhancements added in SQL Server 2012. Although entire books have been written on this vast and ever-expanding topic, our chapter delves into sufficient depth so you can get busy working with geospatial data right away.

Applied SQL

After we've covered so much information about what you can do on the server and in the database, we move to Part III of the book, where we explore technologies and demonstrate techniques for building client/server, n-tier, and cloud solutions on top of your databases. Whatever your scenario, these chapters show you the most effective ways to extend your data's reach. We then conclude with coverage of SQL Azure, the BI stack, and the new columnstore technology known as xVelocity.

- Chapter 10 The Microsoft Data Access Juggernaut

  Chapter 10 covers every client/server data access strategy available in the .NET Framework today. We begin with earliest Microsoft ADO.NET techniques using raw data access objects and the *DataSet* abstraction, and discuss the ongoing relevance of these .NET 1.0 technologies. We then examine later data access technologies, including the concepts and syntax of language-integrated query (LINQ). We look at LINQ to DataSet and LINQ to SQL, and then turn our focus heavily on the ADO.NET Entity Framework (EF), Microsoft's current recommended data access solution for .NET. You will learn Object Relational Mapping (ORM) concepts, and discover how EF's Entity Data Model (EDM) provides a powerful abstraction layer to dramatically streamline the application development process.

- Chapter 11 WCF Data Access Technologies

  After you have mastered the client/server techniques taught in Chapter 10, you are ready to expose your data as services to the world. Chapter 11 provides you with detailed explanations and code samples to get the job done using two technologies based on Windows Communications Foundation (WCF).

  The first part of Chapter 11 covers WCF Data Services, which leverages Representational State Transfer Protocol (REST) and Open Data Protocol (OData) to implement services over your data source. After explaining these key concepts, you will see them put to practical use with concrete examples. As you monitor background network and database activity, we zone in and lock down on the critical internals that make it all work. The second part of the chapter demonstrates data access using WCF RIA Services, a later technology that targets Silverlight clients in particular (although it can support other clients as well). We articulate the similarities and differences between these two WCF-based technologies, and arm you with the knowledge of how and when to use each one.

- Chapter 12 Moving to the Cloud with SQL Azure

  In Chapter 12, we look at the world of cloud database computing with SQL Azure. We explain what SQL Azure is all about, how it is similar to SQL Server and how it differs. We look at how SQL Azure is priced, how to sign up for it, and how to provision SQL Azure servers and databases. We examine the SQL Azure tooling and how to work with SQL Azure from SSMS and SSDT. We explain the many ways that Data-Tier Applications (DACs) can be used to migrate databases between SQL Server and SQL Azure, using SSMS, SSDT, and the native tooling of SQL Azure as well. We finish up the chapter by examining a special partitioning feature called SQL Azure Federations and we look at SQL Azure Reporting, too.

- Chapter 13 SQL Azure Data Sync and Windows Phone Development

  Chapter 13 covers the broad topic of so-called occasionally connected systems by building out a complete solution that incorporates SQL Azure Data Sync, Windows Azure, and the Windows Phone 7 development platform. On the back end, an on-premise SQL Server database is kept synchronized with a public-facing SQL Azure database in the cloud using SQL Azure Data Sync. The cloud database is exposed using WCF Data Services (also hosted in the cloud by deploying to Windows Azure), and consumed via OData by a mobile client application running on a Windows Phone 7 device. The end-to-end solution detailed in this chapter demonstrates how these technologies work to keep data in sync across on-premise SQL Server, SQL Azure databases in the cloud, and local storage on Windows Phone 7 devices.

- Chapter 14 Pervasive Insight

  In Chapter 14, we provide an overview of the entire SQL Server BI stack, including SQL Server Fast Track Data Warehouse appliances, SQL Server Parallel Data Warehouse edition, SQL Server Integration Services, Analysis Services, Master Data Services, Data Quality Services, Reporting Services, Power View, PowerPivot, and StreamInsight. In the interest of completeness, we also provide brief overviews of Excel Services and PerformancePoint Services in SharePoint and how they complement SQL Server. We explain what each BI component does, and how they work together. Perhaps most important, we show you how these technologies from the BI arena are relevant to your work with relational data, and how, in that light, they can be quite approachable. These technologies shouldn’t be thought of as segregated or tangential. They are integral parts of SQL Server, and we seek to make them part of what you do with the product.

- Chapter 15 xVelocity In-Memory Technologies

  In Chapter 15, we look at Microsoft's xVelocity column store technology, and how to use it from the SQL Server relational database, as well as PowerPivot and Analysis Services. We explain how column-oriented databases work, we examine the new columnstore indexes in SQL Server 2012, and discuss its batch processing mode, too. We look at how easy it is for relational database experts to work with PowerPivot and SSAS Tabular mode, and we show how to bring all these technologies together with the SQL Server Power View data analysis, discovery, and visualization tool.

Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow.

- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold.
- Code elements in text (apart from code blocks) appear in italic.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.
- A vertical bar between two or more menu items (for example, File | Close), means that you should select the first menu or menu item, then the next, and so on.

System Requirements

To follow along with the book's text and run its code samples successfully, we recommend that you install the Developer edition of SQL Server 2012, which is available to a great number of developers through Microsoft's MSDN Premium subscription, on your PC. You will also need Visual Studio 2010; we recommend that you use the Professional edition or one of the Team edition releases, each of which is also available with the corresponding edition of the MSDN Premium subscription product. All the code samples will also work with the upcoming Visual Studio 11, in beta at the time of this writing.

Important To cover the widest range of features, this book is based on the Developer edition of SQL Server 2012. The Developer edition possesses the same feature set as the Enterprise edition of the product, although Developer edition licensing terms preclude production use. Both editions are high-end platforms that offer a superset of the features available in other editions (Standard, Workgroup, Web, and Express). We believe that it is in the best interest of developers for us to cover the full range of developer features in SQL Server 2012, including those available only in the Enterprise and Developer editions.

To run these editions of SQL Server and Visual Studio, and thus the samples in this book, you'll need the following 32-bit hardware and software. (The 64-bit hardware and software requirements are not listed here but are very similar.)

- 1 GHz or faster (2 GHz recommended) processor.
- Operating system, any of the following:
    - Microsoft Windows Server 2008 R2 SP1
    - Windows 7 SP1 (32- or 64-bit)
    - Windows Server 2008 SP2
    - Windows Vista SP2
- For SQL Server 2012, 4 GB or more RAM recommended for all editions (except the Express edition, which requires only 1 GB).
- For SQL Server 2012, approximately 1460 MB of available hard disk space for the recommended installation. Approximately 375 MB of additional available hard disk space for SQL Server Books Online, SQL Server Mobile Everywhere Books Online, and sample databases.
- For Visual Studio 2010, maximum of 20 GB available space required on installation drive. Note that this figure includes space for installing the full set of MSDN documentation.
- A working Internet connection (required to download the code samples from the companion website). A few of the code samples also require an Internet connection to run.

- Super VGA (1024 × 768) or higher resolution video adapter and monitor recommended.
- Microsoft Mouse or compatible pointing device recommended.
- Microsoft Internet Explorer 9.0 or later recommended.

Installing SQL Server Data Tools

SSDT does not get installed with either Visual Studio or SQL Server. Instead, SSDT ships separately via the Web Platform Installer (WebPI). This enables Microsoft to distribute timely SSDT updates out-of-band with (that is, without waiting for major releases of) Visual Studio or SQL Server. Before you follow along with the procedures in Chapter 1, download and install SSDT from *http://msdn.microsoft.com/en-us/data/hh297027*.

Using the Book's Companion Website

Visit the book's companion website at the following address:

http://www.microsoftpressstore.com/title/9780735658226

Code Samples

All the code samples discussed in this book can be downloaded from the book's companion website.

Within the companion materials parent folder on the site is a child folder for each chapter. Each child folder, in turn, contains the sample code for the chapter. Because most of the code is explained in the text, you might prefer to create it from scratch rather than open the finished version supplied in the companion sample code. However, the finished version will still prove useful if you make a small error along the way or if you want to run the code quickly before reading through the narrative that describes it.

Sample *AdventureWorks* Databases

As of SQL Server 2005, and updated through SQL Server 2012, Microsoft provides the popular AdventureWorks family of sample databases. Several chapters in this book reference the *AdventureWorks2012* online transaction processing (OLTP) database, and Chapter 15 references the *AdventureWorksDW2012* data warehousing database.

To follow along with the procedures in these chapters, you can download these databases directly from the book's companion website. The databases posted there are the exact versions that this book was written against, originally obtained from CodePlex, which is Microsoft's open source website (in fact, all of Microsoft's official product code samples are hosted on CodePlex). To ensure you receive the same results as you follow along with certain chapters in this book, we recommend downloading the *AdventureWorks2012* OLTP and *AdventureWorksDW2012* data warehousing databases from the book's companion website rather than from CodePlex (where updated versions may cause different results than the original versions).

You can find the directions to attach (use) the sample databases on the sample database download page.

Previous Edition Chapters

In addition to all the code samples, the book's companion website also contains several chapters from the 2008 and 2005 editions of this book that were not updated for this edition in order to accommodate coverage of new SQL Server 2012 features.

You can download SQL Server 2005 chapters that cover Service Broker, native XML Web Services, SQL Server Management Studio, SQL Server Express edition, Integration Services, and debugging, as well as SQL Server 2008 chapters on data warehousing, online analytical processing (OLAP), data mining, and Reporting Services.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/ 9780735658226

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com.*

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let’s keep the conversation going! We’re on Twitter: *http://twitter.com/MicrosoftPress*

Acknowledgements

It's hard to believe I first began research for this book at an early Software Design Review for SQL Server "Denali" in Redmond back in October 2010. This is my second edition as lead author of this book, and although I enjoyed the work even more this time around, it was certainly no easier than the 2008 edition. My goal—upfront—was to produce the most comprehensive (yet approachable) SQL Server 2012 developer resource that I could, one that best answers, "How many ways can I program SQL Server?" I could not have even contemplated pursuing that goal without the aid of numerous other talented and caring individuals—folks who deserve special recognition. Their generous support was lent out in many different yet equally essential ways. So the order of names mentioned below is by no means an indication of degree or proportion; simply put, I couldn't have written this book without everyone's help.

With so many people to thank, Craig Branning (CEO of Tallan, Inc.) is at the top of my most wanted list. Back in mid-2010, Craig was quick to approach me about taking on this project. Next thing I knew, I was on board and we were scarfing down lunch (smooth work!). Thank you (and all the other wonderful folks at Tallan) for getting this book off the ground in the first place, and providing a continuous source of support throughout its production.

I'm also extremely fortunate to have teamed up with my colleague and friend, co-author Andrew Brust (Microsoft MVP/RD). This is actually Andrew's third time around contributing his knowledge and expertise to this resource; he not only co-authored the 2008 edition, but was lead author of the first edition for SQL Server 2005. So I thank him once again for writing four stellar chapters in this new 2012 edition. And Paul Delcogliano (who also contributed to the 2008 edition) did a superb job confronting the topic of end-to-end cloud development with SQL Azure Data Sync, Windows Azure, and Windows Phone 7—all in a single outstanding chapter. Paul, your ambition is admirable, and I thank you for your tireless work and the great job done!

Ken Jones, my pal at O'Reilly Media, gets special mention for his expert guidance, plus his steady patience through all the administrative shenanigans. Thank you Ken, and to your lovely wife, Andrea, as well, for her insightful help with the geospatial content. I was also very lucky to have worked closely with Russell Jones,Melanie Yarbrough, John Mueller, and Christian Holdener, whose superb editorial contributions, technical review, and overall guidance were vital to the successful production of this book.

The assistance provided by a number of people from various Microsoft product teams helped tackle the challenge of writing about new software as it evolved through several beta releases. Thank you to Roger Doherty, for inviting me out to Redmond for the SDR in 2010, as well as for connecting me with the right people I needed to get my job done. Gert Drapers and Adam Mahood were particularly helpful for the inside scoop on SSDT as it changed from one CTP to the next. Adam's always direct and always available lines of communication turned an entire chapter's hard work into fun work. Doug Laudenschlager also provided valuable insight, which enhanced new coverage of unstructured FILESTREAM data. And naturally, a great big thank you to the entire product team for creating the best release of SQL Server ever!

I'm also particularly proud of all the brand new .NET data access coverage in this book, and would like to give special thanks to my pal Marcel de Vries, Microsoft MVP and RD in the Netherlands. Marcel is a master of distributed architectures, and his invaluable assistance greatly helped shape coverage in the WCF data access chapter. *Ik ben heel dankbaar voor jouw inbreng!*

This book could not have been written, of course, without the love and support of my family. I have been consumed by this project for much of the past eighteen months—which has at times transformed me into an absentee. I owe an enormous debt of gratitude to my wonderful partner Mark, and our awesome kids Adam, Jacqueline, Josh, and Sonny, for being so patient and tolerant with me. And greatest thanks of all go out to my dear Mom, bless her soul, for always encouraging me to write with "expression."

—*Leonard Lobel*

When you're not a full-time author, there's really never a "good" time to write a book. It's always an added extra, and it typically takes significantly more time than anticipated. That creates burdens for many people, including the author's family, the book's editors, and co-authors as well. When one of the authors is starting a new business, burdens double, all around. I'd like to thank my family, the Microsoft Press team and especially this book's lead author, Lenni Lobel, for enduring these burdens, with flexibility and uncommonly infinite patience.

—*Andrew Brust*

PART I

Core SQL Server Development

CHAPTER 1

Introducing SQL Server Data Tools

—Leonard Lobel

With the release of SQL Server 2012, Microsoft delivers a powerful, new integrated development environment (IDE) for designing, testing, and deploying SQL Server databases—locally, offline, or in the cloud—all from right inside Microsoft Visual Studio (both the 2010 version and the upcoming Visual Studio 11, currently in beta at the time of this writing). This IDE is called SQL Server Data Tools (SSDT), and it represents a major step forward from previously available tooling—notably, SQL Server Management Studio (SSMS) and Visual Studio Database Professional edition (DbPro).

SSDT is not intended to be a replacement for SSMS, but instead can be viewed much more as a greatly evolved implementation of DbPro. Indeed, SSMS is alive and well in SQL Server 2012, and it continues to serve as the primary management tool for *database administrators* who need to configure and maintain healthy SQL Server installations. And although DbPro was a good first step towards offline database development, its relatively limited design-time experience has precluded its widespread adoption. So for years, programmers have been primarily using SSMS to conduct development tasks (and before SQL Server 2005, they typically used *two* database administrator [DBA] tools—SQL Enterprise Manager and Query Analyzer). It's always been necessary for programmers to use management-oriented tools (such as SSMS) rather than developer-focused tools (such as Visual Studio) as a primary database development environment when building database applications—until now.

The release of SSDT provides a single environment hosted in Visual Studio, with database tooling that specifically targets the development process. Thus, you can now design and build your databases without constantly toggling between Visual Studio and other tools. In this chapter, you'll learn how to use SSDT inside Visual Studio to significantly enhance your productivity as a SQL Server developer. We begin with an overview of key SSDT concepts and features, and then walk you through demonstrations of various connected and disconnected scenarios.

Introducing SSDT

Database Tooling Designed for Developers

The inconvenient truth is: database development is hard. Getting everything done correctly is a huge challenge—proper schema and relational design, the intricacies of Transact-SQL (T-SQL) as a language, performance tuning, and more, are all difficult tasks in and of themselves. However, with respect to the development process—the way in which you create and change a database—there are some particular scenarios that the right tooling can improve greatly. SSDT delivers that tooling.

The SSDT Umbrella of Services and Tools

SSDT encompasses more than just the new database tooling covered in this chapter; it is actually a packaging of what was formerly the Visual Studio 2008–based Business Intelligence Developer Studio (BIDS) tool. SSDT supports the traditional BIDS project types for SQL Server Analysis Services (SSAS), Reporting Services (SSRS), and Integration Services (SSIS), in addition to the new database tooling. So with SSDT, Microsoft has now brought together all of the SQL Server database development experiences inside a single version of Visual Studio.

Despite its broader definition, this chapter uses the term SSDT specifically as it pertains to the new database development tools that SSDT adds to the Visual Studio IDE.

Here are some of the challenges that developers face when designing databases.

- **Dependencies** By its very nature, the database is full of dependencies between different kinds of schema objects. This complicates development, as even the simplest changes can very quickly become very complex when dependencies are involved.
- **Late Error Detection** You can spend a lot of time building complex scripts, only to find out that there are problems when you try to deploy them to the database. Or, your script may deploy without errors, but you have an issue somewhere that doesn’t manifest itself until the user encounters a run-time error.
- **“Drift“ Detection** The database is a constantly moving target. After deployment, it’s fairly common for a DBA to come along and tweak or patch something in the production database; for example, adding indexes to improve query performance against particular tables. When the environments fall out of sync, the database is in a different state than you and your application expect it to be—and those differences need to be identified and reconciled.
- **Versioning** Developers have grown so accustomed to working with “change scripts“ that it makes you wonder, where is the *definition* of the database? Of course you can rely on it being *in* the database, but where is it from the standpoint of preserving and protecting it? How do you maintain the definition across different versions of your application? It’s very difficult to revert to a point in time and recall an earlier version of the database that matches up with an

earlier version of an application. So you can't easily synchronize versions and version history between your database and application.

- **Deployment** Then there are the challenges of targeting different versions, including most recently, SQL Azure. You may need to deploy the same database out to different locations, and must account for varying compatibility levels when different locations are running different versions of SQL Server (such as SQL Server 2005, 2008, 2008 R2, 2012, and SQL Azure).

Many of these pain points are rooted in the notion that the database is "stateful." Every time you build and run a .NET application in Visual Studio, it is always initialized to the same "new" state but as soon as the application goes off to access the database, it's the same "old" database with the same schema and data in it. Thus, you are forced to think not only about the design of the database, but also about how you implement that design—how you actually get that design moved into the database given the database's current state.

If the root of these problems lies in the database being stateful, then the heart of the solution lies in working declaratively rather than imperatively. So rather than just working with change scripts, SSDT lets you work with *a declaration of what you believe (or want) the database to be*. This allows you to focus on the design, while the tool takes care of writing the appropriate change scripts that will safely apply your design to an actual target database. SSDT takes a declarative, model-based approach to database design—and as you advance through this chapter, you'll see how this approach remedies the aforementioned pain points.

Declarative, Model-Based Development

We've started explaining that SSDT uses a declarative, model-based approach. What this means is that there is always an in-memory representation of what a database looks like—an SSDT database *model*—and all the SSDT tools (designers, validations, IntelliSense, schema compare, and so on) operate on that model. This model can be populated by a live connected database (on-premise or SQL Azure), an offline database project under source control, or a point-in-time *snapshot* taken of an offline database project (you will work with snapshots in the upcoming exercises). But to reiterate, the tools are agnostic to the model's backing; they work exclusively against the model itself. Thus, you enjoy a rich, consistent experience in any scenario—regardless of whether you're working with on-premise or cloud databases, offline projects, or versioned snapshots.

The T-SQL representation of any object in an SSDT model is always expressed in the form of a *CREATE* statement. An *ALTER* statement makes no sense in a declarative context—a *CREATE* statement declares what an object should look like, and that's the only thing that you (as a developer) need to be concerned with. Depending on the state of the target database, of course, a change script containing either a *CREATE* statement (if the object doesn't exist yet) or an appropriate *ALTER* statement (if it does) will be needed to properly deploy the object's definition. Furthermore, if dependencies are involved (which they very often are), other objects need to be dropped and re-created in the deployment process. Fortunately, you can now rely on SSDT to identify any changes (the "difference") between your model definition and the actual database in order to compose the necessary change script. This keeps you focused on just the definition. Figure 1-1 depicts the SSDT model-based approach to database development.

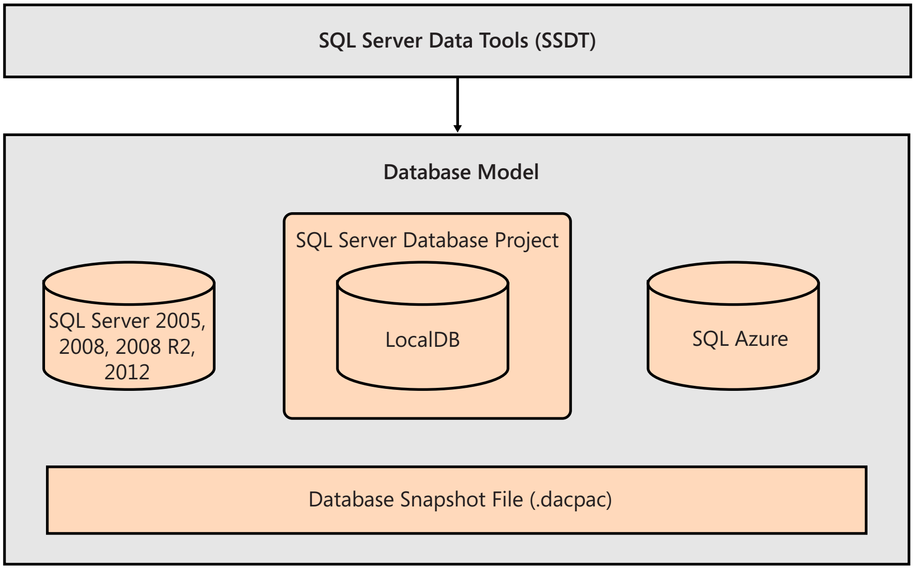

SQL Server Data Tools (SSDT)
Database Model
SQL Server Database Project
SQL Server 2005, 2008, 2008 R2, 2012
LocalDB
SQL Azure
Database Snapshot File (.dacpac)

FIGURE 1-1 SSDT works with a model backed by connected databases (on-premise or in the cloud), offline database projects, or database snapshot files.

Connected Development

Although SSDT places great emphasis on the declarative model, it in no way prevents you from working imperatively against live databases when you want or need to. You can open query windows to compose and execute T-SQL statements directly against a connected database, with the assistance of a debugger if desired, just as you can in SSMS.

The connected SSDT experience is driven off the new SQL Server Object Explorer in Visual Studio. You can use this new dockable tool window to accomplish common database development tasks that formerly required SSMS. Using the new SQL Server Object Explorer is strikingly similar to working against a connected database in SSMS's Object Explorer—but remember that (and we'll risk overstating this) the SSDT tools operate only on a database *model*. So when working in connected mode, SSDT actually creates a model from the real database—on the fly—and then lets you edit that model. This "buffered" approach is a subtle, yet key, distinction from the way that SSMS operates.

When you save a schema change made with the new table designer, SSDT works out the necessary script needed to update the real database so it reflects the change(s) made to the model. Of course, the end result is the same as the connected SSMS experience, so it isn't strictly necessary to understand this buffered behavior that's occurring behind the scenes. But after you do grasp it, the tool's offline project development and snapshot versioning capabilities will immediately seem natural and intuitive to you. This is because offline projects and snapshots are simply different backings of the very same SSDT model. When you're working with the SQL Server Object Explorer, the model's backing just happens to be a live connected database.

There's an additional nuance to SSDT's buffered-while-connected approach to database development that bears mentioning. There are in fact *two* models involved in the process of applying schema changes to a database. Just before SSDT attempts to apply your schema changes, it actually creates a new model of the currently connected database. SSDT uses this model as the target for a model comparison with the model you've been editing. This dual-model approach provides the "drift detection" mechanism you need to ensure that the schema compare operation (upon which SSDT will be basing its change script) accounts for any schema changes that may have been made by another user since you began editing your version of the model. Validation checks will then catch any problems caused by the other user's changes (which would not have been present when you began making your changes).

Disconnected Development

The new SQL Server Object Explorer lets you connect to and interact with any database right from inside Visual Studio. But SSDT offers a great deal more than a mere replacement for the connected SSMS experience. It also delivers a rich offline experience with the new SQL Server Database Project type and local database runtime (LocalDB).

The T-SQL script files in a SQL Server Database Project are all declarative in nature (only *CREATE* statements; no *ALTER* statements). This is a radically different approach than you're accustomed to when "developing" databases in SSMS (where you execute far more *ALTER* statements than *CREATE* statements). Again, you get to focus on defining "this is how the database should look," and let the tool determine the appropriate T-SQL change script needed to actually update the live database to match your definition.

If you are familiar with the Database Professional (DbPro) edition of Visual Studio, you will instantly recognize the many similarities between DbPro's Database Projects and SSDT's SQL Server Database Projects. Despite major overlap however, SSDT project types are different than DbPro project types, and appear as a distinct project template in Visual Studio's Add New Project dialog. The new table designer, buffered connection mechanism, and other SSDT features covered in this chapter work only with SSDT SQL Server Database Projects. However, and as you may have guessed, it's easy to upgrade existing DbPro projects to SSDT projects. Just right-click the project in Solution Explorer and choose Convert To SQL Server Database Project. Note that this is a one-way upgrade, and that DbPro artifacts that are not yet supported by SSDT (such as data generation plans, see the following Note) will not convert.

Note There are several important features still available only in DbPro, most notably data generation, data compare, schema view, and database unit testing. Eventually, SSDT plans on providing key elements of the DbPro feature set and will completely replace the Database Professional edition of Visual Studio. For now though, you must continue to rely on DbPro for what's still missing in SSDT.

The new SQL Server Database Project type enjoys many of the same capabilities and features as other Visual Studio project types. This includes not only source control for each individual database object definition, but many of the common code navigation and refactoring paradigms that developers have grown to expect of a modern IDE (such as Rename, Goto Definition, and Find All References). The SSDT database model's rich metadata also provides for far better IntelliSense than what SSMS has been offering since SQL Server 2008, giving you much more of that "strongly-typed" confidence factor as you code. You can also set breakpoints, single step through T-SQL code, and work with the Locals window much like you can when debugging .NET project types. With SSDT, application and database development tooling has now finally been unified under one roof: Visual Studio.

A major advantage of the model-based approach is that models can be generated from many different sources. When connected directly via SQL Server Object Explorer, SSDT generates a model from the connected database, as we explained already. When you create a SQL Server Database Project (which can be imported from any existing database, script, or snapshot), you are creating an offline, source-controlled project inside Visual Studio that fully describes a real database. But it's actually a project—not a real database. Now, SSDT generates a model that's backed by your SQL Server Database Project. So the experience offline is just the same as when connected—the designers, IntelliSense, validation checks, and all the other tools work exactly the same way.

As you conduct your database development within the project, you get the same "background compilation" experience that you're used to experiencing with typical .NET development using C# or Visual Basic (VB) .NET. For example, making a change in the project that can't be submitted to the database because of dependency issues will result in design-time warnings and errors in the Error List pane. You can then click on the warnings and errors to navigate directly to the various dependencies so they can be dealt with. Once all the build errors disappear, you'll be able to submit the changes to update the database.

Versioning and Snapshots

A database project gives you an offline definition of a database. As with all Visual Studio projects, each database object (table, view, stored procedure, and every other distinct object) exists as a text file that can be placed under source code control. The project system combined with source control enables you to secure the definition of a database in Visual Studio, rather than relying on the definition being stored in the database itself.

At any point in time, and as often as you'd like, you can create a *database snapshot*. A snapshot is nothing more than a file (in the Data-tier Application Component Package, [dacpac] format) that holds the serialized state of a database model, based on the current project at the time the snapshot is taken. It is essentially a single-file representation of your entire database schema. Snapshots can later be deserialized and used with any SSDT tool (schema compare, for example). So you can develop, deploy, and synchronize database structures across local/cloud databases and differently versioned offline database projects, all with consistent tooling.

Pause for a moment to think about the powerful capabilities that snapshots provide. A snaphot encapsulates an entire database structure into a single *.dacpac* file that can be instantly deserialized back into a model at any time. Thus, they can serve as either the source or target of a schema compare operation against a live database (on-premise or SQL Azure), an offline SQL Server Database Project, or some other snapshot taken at any other point in time.

Snapshots can also be helpful when you don't have access to the target database, but are expected instead to hand a change script off to the DBA for execution. In addition to the change script, you can also send the DBA a snapshot of the database project taken just before any of your offline work was performed. That snapshot is your change script's assumption of what the live database looks like. So the DBA, in turn, can perform a schema compare between your snapshot and the live database (this can be done from SSDT's command-line tool without Visual Studio). The results of that schema compare will instantly let the DBA know if it's safe to run your change script. If the results reveal discrepancies between the live database and the database snapshot upon which your change script is based, the DBA can reject your change script and alert you to the problem.

Targeting Different Platforms

SQL Server Database Projects have a target platform switch that lets you specify the specific SQL Server version that you intend to deploy the project to. All the validation checks against the project-backed model are based on this setting, making it trivial for you to test and deploy your database to any particular version of of SQL Server (2005 and later), including SQL Azure. It's simply a matter of choosing SQL Azure as the target to ensure that your database can be deployed to the cloud without any problems. If your database project defines something that is not supported in SQL Azure (a table with no clustered index, for example), it will get flagged as an error automatically.

Working with SSDT

Our introduction to the SSDT toolset is complete, and it's now time to see it in action. The rest of this chapter presents a sample scenario that demonstrates, step-by-step, how to use many of the SSDT features that you've just learned about. Practicing along with this example will solidify your knowledge and understanding of the tool, and prepare you for using SSDT in the real world with real database projects.

Important SSDT does not get installed with either Visual Studio or SQL Server. Instead, SSDT ships separately via the Web Platform Installer (WebPI). This enables Microsoft to distribute timely SSDT updates out-of-band with (that is, without waiting for major releases of) Visual Studio or SQL Server. Before proceeding, download and install SSDT from *http://msdn.microsoft.com/en-us/data/hh297027*.

Connecting with SQL Server Object Explorer

The journey will start by creating a database. You will first use SSDT to execute a prepared script that creates the database in a query window connected to a SQL Server instance. Then you will start working with SSDT directly against the connected database. This experience is similar to using previous tools, so it's the perfect place to start. Later on, you'll switch to working disconnected using an offline database project.

Launch Visual Studio 2010, click the View menu, and choose SQL Server Object Explorer. This displays the new SQL Server Object Explorer in a Visual Studio panel (docked to the left, by default). This new tool window is the main activity hub for the connected development experience in SSDT. From the SQL Server Object Explorer, you can easily connect to any server instance for which you have credentials. In our scenario, the *localhost* instance running on the development machine is a full SQL Server 2012 Developer edition installation. This instance is assuming the role of a live production database server that you can imagine is running in an on-premise datacenter. You're now going to connect to that "production" database server.

Right-click the *SQL Server* node at the top of the SQL Server Object Explorer, choose Add SQL Server, and type in your machine name as the server to connect to. Although you can certainly, alternatively, type *localhost* instead (or even simply the single-dot shorthand syntax for *localhost*), we're directing you to use the machine name instead. This is because you'll soon learn about the new local database runtime (LocalDB) that SSDT provides for offline testing and debugging. The LocalDB instance *always* runs locally, whereas the production database on the other hand just *happens* to be running locally. Because it can be potentially confusing to see both *localhost* and *(localdb)* in the SQL Server Object Explorer, using the machine name instead of *localhost* makes it clear that one represents the production database while the other refers to the database used for local (offline) development and testing with SSDT. The screen snapshots for the figures in this chapter were taken on a Windows Server 2008 R2 machine named *SQL2012DEV*, so we'll be using that machine name throughout the chapter when referring to the production database running on *localhost*. Of course, you'll need to replace the assumed *SQL2012DEV* machine name with your own machine name wherever you see it mentioned.

If you have installed SQL Server to use mixed-mode authentication and you are not using Windows authentication, then you'll also need to choose SQL Server authentication and supply your credentials at this point, before you can connect. Once connected, SQL Server Object Explorer shows the production server and lets you drill down to show all the databases running on it, as shown in Figure 1-2.

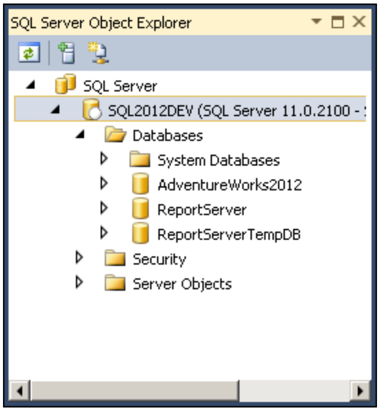
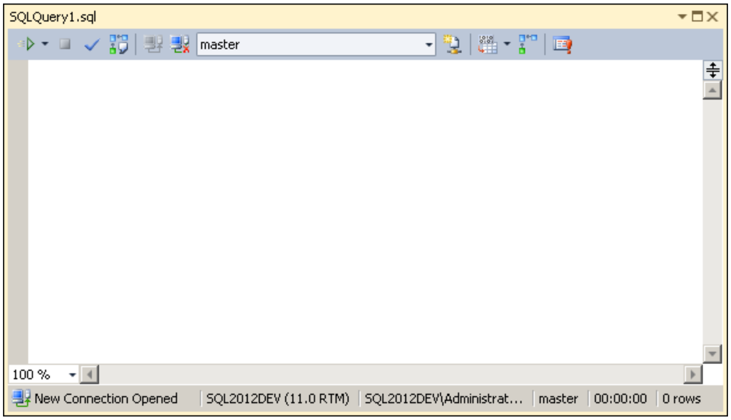


FIGURE 1-2 The new SQL Server Object Explorer in Visual Studio expanded to show several connected databases.

Once connected, right-click the server instance node *SQL2012DEV* and choose New Query. Visual Studio opens a new T-SQL code editor window, like the one shown in Figure 1-3.



New Connection Opened | SQL2012DEV (11.0 RTM) | SQL2012DEV\Administrat... | master | 00:00:00 | 0 rows

FIGURE 1-3 A connected query window.

This environment should seem very familiar to anyone experienced with SSMS or Query Analyzer. Notice the status bar at the bottom indicating that the query window is currently connected to the *SQL2012DEV* instance (again, it will actually read your machine name). The toolbar at the top includes a drop-down list indicating the current default database for the instance you're connected to. As with previous tools, this will be the *master* database (as shown at the top of Figure 1-3). You must still take care to change this setting (or issue an appropriate *USE* statement) so that you don't inadvertently access the *master* database when you really mean to access your application's database. In this exercise, you're creating a brand new database, so it's fine that the current database is set to *master* at this time.

Tip This concern stems from the fact that, by default, the *master* database is established as every login's default database. A great way to protect yourself from accidentally trampling over the *master* database when working with SSDT (or any other tool) is to change your login's default database to be your application's database, which will then become the default database (rather than *master*) for every new query window that you open. This will greatly reduce the risk of unintentional data corruption in *master*, which can have disasterous consequences.

When you navigate to your login from the Security node in SQL Server Object Explorer, you'll be able to see its default database set to *master* in the Properties window, but you won't be able to change it. This is a management task that is not supported in the SSDT tooling, although you can still use SSDT to execute the appropriate *ALTER LOGIN* statement in a query window. Alternatively, you can easily make the change in SSMS as follows. Start SSMS, connect to your server instance, and drill down to your login beneath the Security and Logins nodes in the SSMS Object Explorer. Then right-click the login and choose Properties. Click the default database drop-down list, change its value from *master* to your application's database, and click OK. From then on, your database (not *master*) will be set as the default for every new SSDT query window that you open.

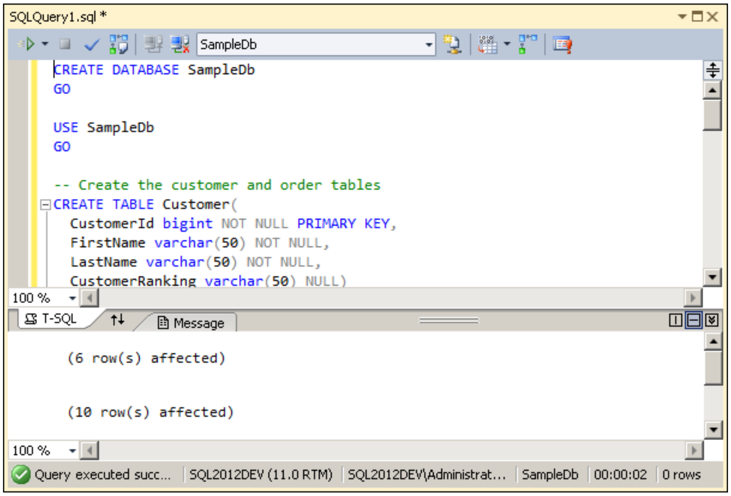
Type the code shown in Listing 1-1 into the query window (or open the script file available in the downloadable code on this book's companion website; see the section “Code Samples“ in the “Introduction“ for details on how to download the sample code). You might next be inclined to press F5 to execute the script, but that won't work. With SSDT in Visual Studio, pressing F5 *builds and deploys* a SQL Server Database Project to a debug instance (you'll be creating such a project later on, but you don't have one yet). This is very different to the way F5 is used in SSMS or Query Analyzer to immediately execute the current script (or currently selected script).

SSDT uses a different keyboard shortcut for this purpose. In fact, there are two keyboard shortcuts (with corresponding toolbar buttons and right-click context menu items); one to execute without a debugger (Ctrl+Shift+E) and one to execute using an attached debugger (Alt+F5). You'll practice debugging later on, so for now just press **Ctrl+Shift+E** to immediately execute the script and create the database (you can also click the Execute Query button in the toolbar, or right-click anywhere within the code window and choose Execute from the context menu).

LISTING 1-1 T-SQL script for creating the *SampleDb* database

```
CREATE DATABASE SampleDb
GO

USE SampleDb
GO

-- Create the customer and order tables
CREATE TABLE Customer(
  CustomerId bigint NOT NULL PRIMARY KEY,
  FirstName varchar(50) NOT NULL,
  LastName varchar(50) NOT NULL,
  CustomerRanking varchar(50) NULL)

CREATE TABLE OrderHeader(
  OrderHeaderId bigint NOT NULL,
  CustomerId bigint NOT NULL,
  OrderTotal money NOT NULL)

-- Create the relationship
ALTER TABLE OrderHeader ADD CONSTRAINT FK_OrderHeader_Customer
 FOREIGN KEY(CustomerId) REFERENCES Customer(CustomerId)

-- Add a few customers
INSERT INTO Customer (CustomerId, FirstName, LastName, CustomerRanking) VALUES
 (1, 'Lukas', 'Keller', NULL),
 (2, 'Jeff', 'Hay', 'Good'),
 (3, 'Keith', 'Harris', 'so-so'),
 (4, 'Simon', 'Pearson', 'A+'),
 (5, 'Matt', 'Hink', 'Stellar'),
 (6, 'April', 'Reagan', '')

-- Add a few orders
INSERT INTO OrderHeader(OrderHeaderId, CustomerId, OrderTotal) VALUES
 (1, 2, 28.50), (2, 2, 169.00),  -- Jeff's orders
 (3, 3, 12.99),  -- Keith's orders
 (4, 4, 785.75), (5, 4, 250.00),  -- Simon's orders
 (6, 5, 6100.00), (7, 5, 4250.00),  -- Matt's orders
 (8, 6, 18.50), (9, 6, 10.00), (10, 6, 18.00)  -- April's orders
GO

-- Create a handy view summarizing customer orders
CREATE VIEW vwCustomerOrderSummary WITH SCHEMABINDING AS
 SELECT
   c.CustomerID, c.FirstName, c.LastName, c.CustomerRanking,
   ISNULL(SUM(oh.OrderTotal), 0) AS OrderTotal
  FROM
   dbo.Customer AS c
   LEFT OUTER JOIN dbo.OrderHeader AS oh ON c.CustomerID = oh.CustomerID
  GROUP BY
   c.CustomerID, c.FirstName, c.LastName, c.CustomerRanking
GO
```

This is a very simple script that we'll discuss in a moment. But first, notice what just happened. SSDT executed the script directly against a connected SQL Server instance, and then split the code window horizontally to display the resulting server messages in the bottom pane. The green icon labeled Query Executed Successfully in the lower-left corner offers assurance that all went well with the script execution. Because of the two multi-row *INSERT* statements used to create sample customers and order data, you can see the two "rows affected" messages in the bottom Message pane, as shown in Figure 1-4. Overall, the experience thus far is very similar to previous tools, ensuring a smooth transition to SSDT for developers already familiar with the older tooling.



```
CREATE DATABASE SampleDb
GO

USE SampleDb
GO

-- Create the customer and order tables
CREATE TABLE Customer(
  CustomerId bigint NOT NULL PRIMARY KEY,
  FirstName varchar(50) NOT NULL,
  LastName varchar(50) NOT NULL,
  CustomerRanking varchar(50) NULL)
```

```
  (6 row(s) affected)


  (10 row(s) affected)
```

FIGURE 1-4 The query window after successfully executing a T-SQL script.

This simple script created a database named *SampleDb*, with the two tables *Customer* and *OrderHeader*. It also defined a foreign key on the *CustomerId* column in both tables, which establishes the parent-child (one-to-many) relationship between them. It then added a few customer and related order rows into their respective tables. Finally, it created a view summarizing each customer's orders by aggregating all their order totals.

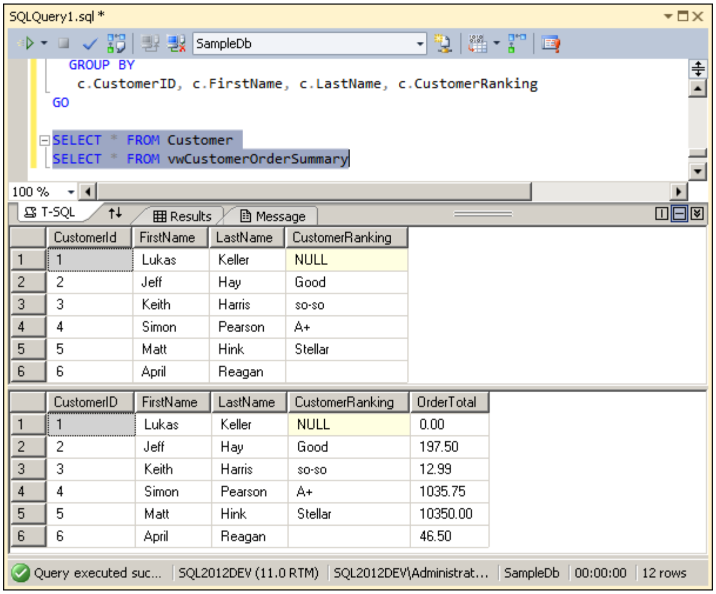
Now run two queries to view some data. At the bottom of the code window, type the following two SELECT statements:

```
SELECT * FROM Customer
SELECT * FROM vwCustomerOrderSummary
```

Notice the IntelliSense as you type. After you finish typing, hover the cursor over *Customer*, and then again over *vwCustomerOrderSummary*. Visual Studio displays tooltips describing those objects respectively as a table and a view. Now hover the cursor over the star symbol in each *SELECT* statement. Visual Studio displays a tooltip listing all the fields represented by the star symbol in each query. This functionality is provided by the SSDT T-SQL language services running in the background that continuously query the database model backed by the connected *SampleDb* database.

Now select just the two *SELECT* statements (leave the entire script above them unselected) and press **Ctrl+Shift+E**. The result is similar to pressing F5 in SSMS or Query Analyzer: only the selected text is executed (which is what you'd expect). SSDT runs both queries and displays their results, as shown in Figure 1-5.

You don't need the query window any longer, so go ahead and close it now (you also don't need to save the script). Right-click the *Databases* node in SQL Server Object Explorer and choose Refresh. You'll see the new *SampleDb* database node appear. Expand it to drill down into the database. As Figure 1-6 shows, the environment is similar to the Object Explorer in SSMS, and lets you carry out most (but not all) of the developer-oriented tasks that SSMS lets you perform.



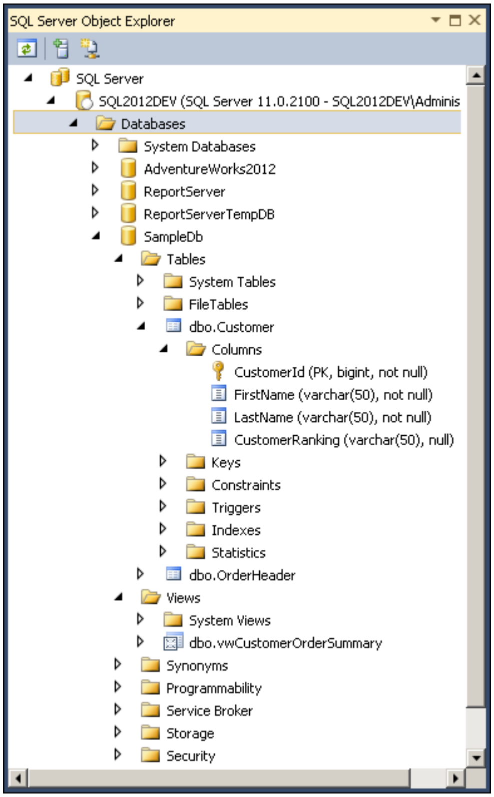
```
    GROUP BY
      c.CustomerID, c.FirstName, c.LastName, c.CustomerRanking
  GO

  SELECT * FROM Customer
  SELECT * FROM vwCustomerOrderSummary
```

	CustomerId	FirstName	LastName	CustomerRanking
1	1	Lukas	Keller	NULL
2	2	Jeff	Hay	Good
3	3	Keith	Harris	so-so
4	4	Simon	Pearson	A+
5	5	Matt	Hink	Stellar
6	6	April	Reagan	

	CustomerID	FirstName	LastName	CustomerRanking	OrderTotal
1	1	Lukas	Keller	NULL	0.00
2	2	Jeff	Hay	Good	197.50
3	3	Keith	Harris	so-so	12.99
4	4	Simon	Pearson	A+	1035.75
5	5	Matt	Hink	Stellar	10350.00
6	6	April	Reagan		46.50

Query executed suc... | SQL2012DEV (11.0 RTM) | SQL2012DEV\Administrat... | SampleDb | 00:00:00 | 12 rows

FIGURE 1-5 Viewing the results of selected statements executed in the query window.



```
SQL Server
  SQL2012DEV (SQL Server 11.0.2100 - SQL2012DEV\Adminis
    Databases
      System Databases
      AdventureWorks2012
      ReportServer
      ReportServerTempDB
      SampleDb
        Tables
          System Tables
          FileTables
          dbo.Customer
            Columns
              CustomerId (PK, bigint, not null)
              FirstName (varchar(50), not null)
              LastName (varchar(50), not null)
              CustomerRanking (varchar(50), null)
            Keys
            Constraints
            Triggers
            Indexes
            Statistics
          dbo.OrderHeader
        Views
          System Views
          dbo.vwCustomerOrderSummary
        Synonyms
        Programmability
        Service Broker
        Storage
        Security
```

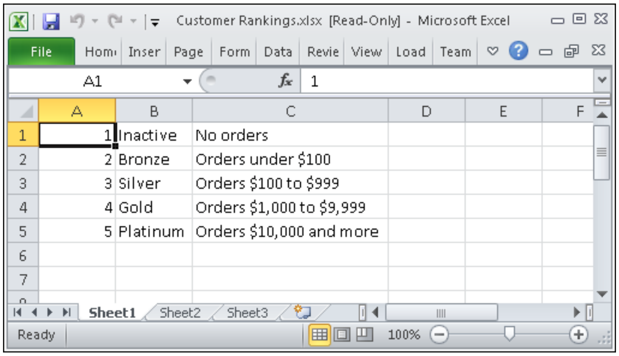
FIGURE 1-6 The *SampleDb* database in SQL Server Object Explorer expanded to show several of its objects.

The database is now up and running on *SQL2012DEV*. Everything is perfect—until that email from marketing arrives. Their team has just put together some new requirements for you, and now there's more work to be done.

Gathering New Requirements

The new requirements pertain to the way customers are ranked in the *Customer* table. Originally, the marketing team had requested adding the *CustomerRanking* column as a lightweight mechanism for data entry operators to rank customer performance. This ad-hoc rating was supposed to be loosely based on the customer's total purchases across all orders, but you can see from the *CustomerRanking* values in Figure 1-5 that users followed no consistency whatsoever as they entered data (no surprise there). They've typed things like **A+**, **so-so**, and **Good**. And some customers have no meaningful data at all, such as empty strings, whitespace, or *NULL* values.

To improve the situation, the marketing team would like to retire the ad-hoc data entry column and replace it with a formalized customer ranking system that is more aligned with their original intent. In their change request email (which is naturally flagged Urgent), they have attached the spreadsheet shown in Figure 1-7 containing new reference data for various pre-defined ranking levels. They've scribbled something about deploying to SQL Azure as well, and then they sign off with "P.S., We need it by Friday" (and no surprise there, either).



	A	B	C
1	1	Inactive	No orders
2	2	Bronze	Orders under $100
3	3	Silver	Orders $100 to $999
4	4	Gold	Orders $1,000 to $9,999
5	5	Platinum	Orders $10,000 and more

FIGURE 1-7 Reference data for the new customer ranking system.

After giving the matter some thought, you organize a high-level task list. Your list itemizes the development steps you plan on taking to fulfill the marketing department's requirements:

1. Remove the *CustomerRanking* column from the *Customer* table.
2. Create a new *CustomerRanking* table based on the spreadsheet in Figure 1-7, with a primary key column *CustomerRankingId* storing the values 1 through 5.
3. Add a new column *CustomerRankingId* to the *Customer* table.
4. Create a foreign key on the *CustomerRankingId* column to establish a relationship between the *Customer* table and the new *CustomerRanking* table.

5. Update the *vwCustomerOrderSummary* view to join on the new *CustomerRanking* table.

6. Create a new *uspRankCustomers* stored procedure to update the *Customer* table's new foreign key column, based on each customer's total order value.

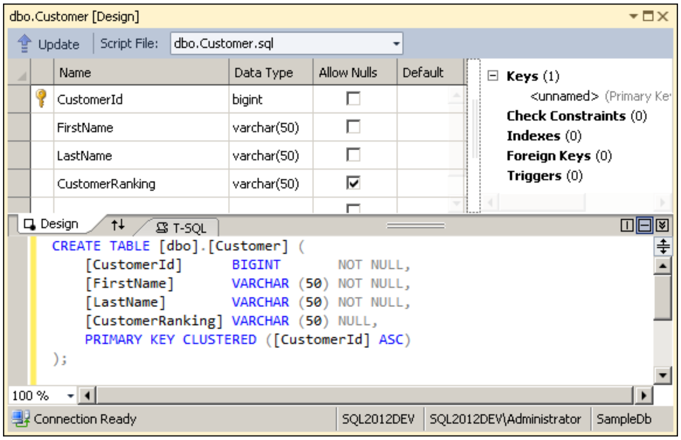
7. Validate for SQL Azure, then deploy to the cloud.

The rest of this chapter walks through this procedure in detail, step by step. Along the way, you'll learn to leverage many important SSDT features and will gain insight into the way the new tooling works. It's time to get started with the first step: removing a column from a table.

Note The scenario we've presented here is admittedly somewhat artificial. We are not necessarily advocating these particular steps as the best way to solve a given problem, and certainly hope you are working with better designs than this in your own database. But for the purpose of this exercise—namely, learning how to use SSDT—we ask that you go along with it. The premise may be contrived, but the steps we've outlined for the solution are in fact quite representative of typical recurring activities in the everyday life of an average SQL Server developer.

Using the Table Designer (Connected)

In SQL Server Object Explorer, right-click the *Customer* table and choose View Designer to open the SSDT table designer, as shown in Figure 1-8.



Name	Data Type	Allow Nulls	Default
CustomerId	bigint	□	
FirstName	varchar(50)	□	
LastName	varchar(50)	□	
CustomerRanking	varchar(50)	☑	
		□	

Keys (1)
<unnamed> (Primary Ke
Check Constraints (0)
Indexes (0)
Foreign Keys (0)
Triggers (0)

```
CREATE TABLE [dbo].[Customer] (
    [CustomerId]      BIGINT       NOT NULL,
    [FirstName]       VARCHAR (50) NOT NULL,
    [LastName]        VARCHAR (50) NOT NULL,
    [CustomerRanking] VARCHAR (50) NULL,
    PRIMARY KEY CLUSTERED ([CustomerId] ASC)
);
```

FIGURE 1-8 The new SSDT table designer.

The top-left pane of this designer window lists the defined columns in a grid just as in the SSMS table designer, but the similarity ends there. A very different mechanism is at play with the new SSDT designer, one that should be easy to understand after all the discussion we've had around declarative, model-based design. The *CREATE TABLE* statement in the bottom T-SQL pane gives it away. Knowing that the table already exists in the database, why is this a *CREATE* statement? Well, that's because this isn’t actually T-SQL code that you intend to execute against the database as-is (which would fail of course, because the table exists). Rather, it’s a T-SQL *declaration* of “how this table should look,” whether it exists or not—and indeed, whether it exists with a *different schema* or not—in the target database.

Here’s what’s actually happening. The designer is operating over a memory-resident database model inside its working environment. Because you are connected at the moment, that model is backed by the live *SampleDb* database. But when you switch to working offline with a SQL Server Database Project (as you will in the next section of this chapter), you’ll interact with the very same table designer over a model backed by a *project* instead of a real database. A model can also be backed by a database snapshot. Because the table designer just operates over a model, the same tool works consistently in any of these scenarios.

You want to remove the *CustomerRanking* column, and that can be done either by deleting it from the grid in the top pane or editing it out of the declarative T-SQL code in the bottom pane. Both panes are merely views into the same table, so any changes appear bidirectionally. Throughout this exercise, you’ll experiment with different editing techniques in the table designer, starting with the quickest method. Just right-click *CustomerRanking* in the top grid and choose Delete. The column is removed from the grid and, as you’d expect, the T-SQL code is updated to reflect the change.

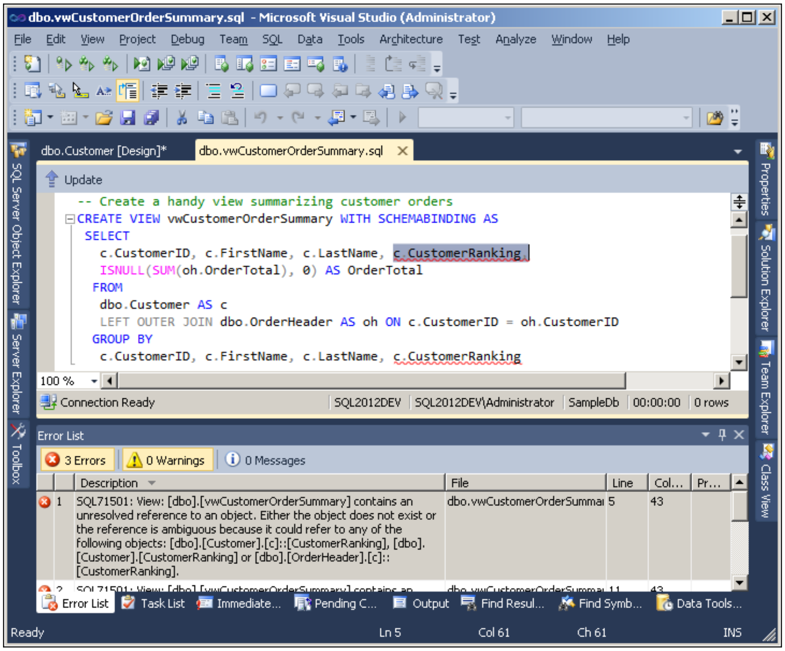
That was a pretty easy change. Applying that change to the database should be easy, too. Go ahead and click the Update button on the toolbar. Unfortunately, instead of just working as you’d like, you receive the following error message:

```
Update cannot proceed due to validation errors.
Please correct the following errors and try again.

SQL71501 :: View: [dbo].[vwCustomerOrderSummary] contains an unresolved reference to an
object. Either the object does not exist or the reference is ambiguous because it could refer
to any of the following objects: [dbo].[Customer].[c]::[CustomerRanking], [dbo].[Customer].
[CustomerRanking] or [dbo].[OrderHeader].[c]::[CustomerRanking].
SQL71501 :: View: [dbo].[vwCustomerOrderSummary] contains an unresolved reference to an
object. Either the object does not exist or the reference is ambiguous because it could refer
to any of the following objects: [dbo].[Customer].[c]::[CustomerRanking], [dbo].[Customer].
[CustomerRanking] or [dbo].[OrderHeader].[c]::[CustomerRanking].
SQL71558 :: The object reference [dbo].[Customer].[CustomerID] differs only by case from the
object definition [dbo].[Customer].[CustomerId].
SQL71558 :: The object reference [dbo].[OrderHeader].[CustomerID] differs only by case from the
object definition [dbo].[OrderHeader].[CustomerId].
```

What went wrong? Referring back to Listing 1-1, notice that the view definition for *vwCustomerOrderSummary* specifies the *WITH SCHEMABINDING* clause. This means that the columns of the view are bound to the underlying tables exposed by the view, which protects you from “breaking“ the view with schema changes—as you’ve done just now. The problem, as reported by the first two errors, is that the schema-bound view’s *CustomerRanking* column has suddenly been removed from the *Customer* table that underlies the view. The second two errors are actually only case-sensitivity warnings that, on their own, would not prevent the update from succeeding. We will explain these case-sensitivity warnings a bit later; for now, remain focused on the dependency issue that’s blocking the update.

The interesting thing worth noting at this point is that SSDT caught the condition before even *attempting* to apply your changes to the live database (which would certainly have thrown an error). In fact, you could have been aware of this issue even before clicking Update if you had previously opened the Error List pane, because SSDT constantly validates changes to the model in the background while you edit it in the designer.

Click the Cancel button to dismiss the error window. Then click the View menu and choose Error List to open the pane. Notice how the errors and warnings appear, just like compilation errors appear for C# and VB .NET projects. And just like those project types, you can double-click items in the Error List and instantly navigate to the offending code to deal with the errors. In this case, both dependency errors are in *vwCustomerOrderSummary*, so double-click either one now to open a code editor into the view, as shown in Figure 1-9.


dbo.vwCustomerOrderSummary.sql - Microsoft Visual Studio (Administrator)
File Edit View Project Debug Team SQL Data Tools Architecture Test Analyze Window Help
dbo.Customer [Design]*
dbo.vwCustomerOrderSummary.sql
Update
SQL Server Object Explorer
Server Explorer
Toolbox
Properties
Solution Explorer
Team Explorer
Class View

```
-- Create a handy view summarizing customer orders
CREATE VIEW vwCustomerOrderSummary WITH SCHEMABINDING AS
 SELECT
  c.CustomerID, c.FirstName, c.LastName, c.CustomerRanking,
  ISNULL(SUM(oh.OrderTotal), 0) AS OrderTotal
 FROM
  dbo.Customer AS c
  LEFT OUTER JOIN dbo.OrderHeader AS oh ON c.CustomerID = oh.CustomerID
 GROUP BY
  c.CustomerID, c.FirstName, c.LastName, c.CustomerRanking
```

		Description	File	Line	Col...	Pr...
⊗	1	SQL71501: View: [dbo].[vwCustomerOrderSummary] contains an unresolved reference to an object. Either the object does not exist or the reference is ambiguous because it could refer to any of the following objects: [dbo].[Customer].[c]::[CustomerRanking], [dbo].[Customer].[CustomerRanking] or [dbo].[OrderHeader].[c]::[CustomerRanking].	dbo.vwCustomerOrderSummai	5	43	
⊗	2	SQL71501: View: [dbo].[vwCustomerOrderSummary] contains an	dbo.vwCustomerOrderSummai	11	43	

FIGURE 1-9 Detecting and navigating validation errors.

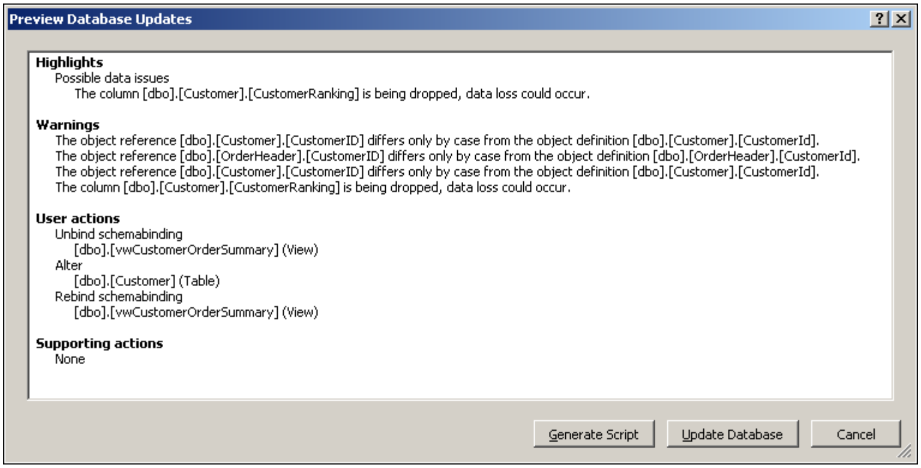
You want to revise this view to join against the new *CustomerRanking* table, but that's not coming up until step 4 in your task list. So for now, just perform the minimum edits necessary to clear the validation errors (which are identified by red squigglies like you've seen in other Visual Studio code windows) so you can update the database and move on. Delete (by commenting out) the two references to *c.CustomerRanking* column from the view (one is in the column list, the other in the *GROUP BY* clause). Notice how the errors disappear from the Error List pane as you correct the code. You're now beginning to experience the effects of model-based development with SSDT in Visual Studio.

With a clear Error List, you know that all your changes are valid. You have altered a table and a view, but those changes have been made only to the memory-resident model. The changed objects are currently open in separate Visual Studio windows, and both windows have an Update button.

Yet it makes no difference which of the two buttons you click—in either case, Update means that *all* changes that have been buffered get sent to the database. So whichever Update button you click, your edits to both the table and the view are going to get applied to the database at once.

How is that going to happen? The edits were simple enough, but the T-SQL change script needed to apply those edits is actually a bit more complex. And therein lay the beauty of this new tooling—all of that scripting complexity is handled for you by SSDT. The tool compares the edited model with a brand-new model based on the live database, and then works out the change script automatically. Creating a fresh model from the database at this time makes sure you're working with its latest state, in case it drifted because it was modeled for the current editing session. Then it runs an internal *schema compare* operation between the edited model (the source) and the latest model based on the live database (the target) to identify all their differences. Finally, SSDT generates a change script that can be executed on the live database to apply the changes. Click Update now to generate the change script.

Before running the change script, SSDT displays an informative report of all the actions that the change script is going to take. Click Update now to display the Preview Database Updates window, as shown in Figure 1-10.



```
Highlights
    Possible data issues
        The column [dbo].[Customer].[CustomerRanking] is being dropped, data loss could occur.

Warnings
    The object reference [dbo].[Customer].[CustomerID] differs only by case from the object definition [dbo].[Customer].[CustomerId].
    The object reference [dbo].[OrderHeader].[CustomerID] differs only by case from the object definition [dbo].[OrderHeader].[CustomerId].
    The object reference [dbo].[Customer].[CustomerID] differs only by case from the object definition [dbo].[Customer].[CustomerId].
    The column [dbo].[Customer].[CustomerRanking] is being dropped, data loss could occur.

User actions
    Unbind schemabinding
        [dbo].[vwCustomerOrderSummary] (View)
    Alter
        [dbo].[Customer] (Table)
    Rebind schemabinding
        [dbo].[vwCustomerOrderSummary] (View)

Supporting actions
    None
```

FIGURE 1-10 The Preview Database Updates window.

You should always scrutinize this preview to make sure it's consistent with the actions and results you would expect of the edits you've made. In this case, you're being warned about data loss in the *Customer* table by dropping the *CustomerRanking* column. You're also being told that the script will drop and then re-create the schema binding of the *vwCustomerOrderSummary* view, before and after the table is altered. All of this is expected. Now you can click Update Database to immediately execute the change script, or you can click Generate Script to load the change script into a code editor so you can view, possibly modify, and choose to either execute it or not.

In most cases, you'll feel comfortable just clicking Update Database, particularly if you've reviewed the warnings and actions reported by the database update preview. Doing so will immediately execute the change script to update the live database. But being that this is your very first update, click Generate Script instead so you can examine the script before you run it. The script is shown in Listing 1-2 (to conserve space, error-checking code has been commented out).

LISTING 1-2 The change script for the altered table and view automatically generated by SSDT.

```
/*
Deployment script for SampleDb
*/

// ...
:setvar DatabaseName "SampleDb"
GO
// ...
USE [$(DatabaseName)];
GO
// ...
BEGIN TRANSACTION
GO
PRINT N'Removing schema binding from [dbo].[vwCustomerOrderSummary]...';
GO
ALTER VIEW [dbo].[vwCustomerOrderSummary]
AS
SELECT   c.CustomerID,
         c.FirstName,
         c.LastName,
         c.CustomerRanking,
         ISNULL(SUM(oh.OrderTotal), 0) AS OrderTotal
FROM     dbo.Customer AS c
         LEFT OUTER JOIN
         dbo.OrderHeader AS oh
         ON c.CustomerID = oh.CustomerID
GROUP BY c.CustomerID, c.FirstName, c.LastName, c.CustomerRanking;
// ...
GO
PRINT N'Altering [dbo].[Customer]...';
GO
ALTER TABLE [dbo].[Customer] DROP COLUMN [CustomerRanking];
GO
// ...
PRINT N'Adding schema binding to [dbo].[vwCustomerOrderSummary]...';
GO

-- Create a handy view summarizing customer orders
ALTER VIEW vwCustomerOrderSummary WITH SCHEMABINDING AS
 SELECT
   c.CustomerID, c.FirstName, c.LastName,
   ISNULL(SUM(oh.OrderTotal), 0) AS OrderTotal
  FROM
   dbo.Customer AS c
   LEFT OUTER JOIN dbo.OrderHeader AS oh ON c.CustomerID = oh.CustomerID
  GROUP BY
   c.CustomerID, c.FirstName, c.LastName
```

```
GO
// ...
IF @@TRANCOUNT>0 BEGIN
PRINT N'The transacted portion of the database update succeeded.'
COMMIT TRANSACTION
END
ELSE PRINT N'The transacted portion of the database update failed.'
GO
DROP TABLE #tmpErrors
GO
PRINT N'Update complete.'
GO
```

It’s great that you didn’t have to *write* the change script, but it’s still important that you *understand* the change script. Let’s look it over quickly now.

Using variable substitution, the script first issues a *USE* statement that sets *SampleDb* as the current database and then it begins a transaction. The transaction will get committed only if the entire change script completes successfully. Then the script issues an *ALTER VIEW* statement that removes the schema binding from *vwCustomerOrderSummary* without yet changing its definition. So it still contains those two references to the *CustomerRanking* column that’s about to get dropped from the *Customer* table, but that will not present a problem because *WITH SCHEMABINDING* has been removed from the view. Next, the script issues the *ALTER TABLE* statement that actually drops the column from the table. After the column is dropped, another *ALTER VIEW* statement is issued on *vwCustomerOrderSummary* with the new version that no longer references the dropped column and is once again schemabound. Finally, the transaction is committed and the update is complete.

Press **Ctrl+Shift+E**. The script is executed and output describing actions taken by the script are displayed in the Messages pane:

```
Removing schema binding from [dbo].[vwCustomerOrderSummary]...
Altering [dbo].[Customer]...
Adding schema binding to [dbo].[vwCustomerOrderSummary]...
The transacted portion of the database update succeeded.
Update complete.
```

You can close all open windows now. Visual Studio will prompt to save changes, but it’s not necessary to do so because the database has just been updated. Right-click on the database and choose Refresh, and then drill down into *SampleDb* in SQL Server Object Explorer to confirm that the table and view changes have been applied. You will see that the *CustomerRanking* column has been removed from the database, and that completes step 1.

Working Offline with a SQL Server Database Project

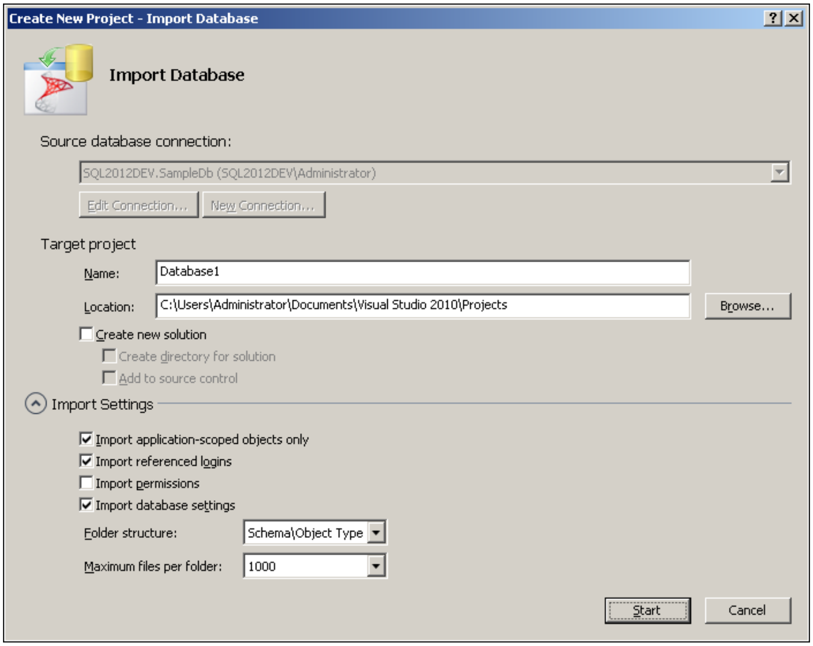
With SQL Server Database Projects, you can develop databases with no connection whatsoever to a SQL Server instance. A SQL Server Database Project is a project that contains individual, declarative, T-SQL source code files. These source files collectively define the complete structure of a database.

Because the database definition is maintained this way inside a Visual Studio project, it can be preserved and protected with source code control (SCC) just like the artifacts in all your other Visual Studio project types. SSDT generates a model from the project structure behind the scenes, just like it generates a model from the live database when working connected. This lets you use the same SSDT tools whether working offline or connected.

You carried out your first task online while connected directly to a live database. Now you'll create a SQL Server Database Project for the database so that you can continue your work offline. Although (as you've seen) it's easy to use SSDT for connected development, you should ideally develop your databases offline with SQL Server Database Projects, and publish to live servers whenever you're ready to deploy your changes. By doing so, you will derive the benefits of source control, snapshot versioning, and integration with the rest of your application's code through the Visual Studio solution and project system.

There are several ways to create a SQL Server Database Project. You can start with an empty project, design a database structure from the ground up, and then publish the entire structure to a new database on a SQL Server instance locally or in the cloud on SQL Azure. Or, as in this scenario, you have an existing database on a local SQL Server instance from which you want to generate a SQL Server Database Project. And you want this project populated with all the declarative T-SQL source files that completely define the existing database structure.

It's easy to do this with the Import Database dialog. Right-click the *SampleDb* database under the *SQL2012DEV* instance in SQL Server Object Explorer and choose Create New Project to display the Import Database dialog, as shown in Figure 1-11.



Source database connection:

SQL2012DEV.SampleDb (SQL2012DEV\Administrator)

Target project

Name: Database1

Location: C:\Users\Administrator\Documents\Visual Studio 2010\Projects

☐ Create new solution

☐ Create directory for solution

☐ Add to source control

Import Settings

☑ Import application-scoped objects only

☑ Import referenced logins

☐ Import permissions

☑ Import database settings

Folder structure: Schema\Object Type

Maximum files per folder: 1000

FIGURE 1-11 Creating a SQL Server Database Project from an existing database.

The source database connection confirms that your new project will be generated from the *SampleDb* database on *SQL2012DEV*. Change the target project name from Database1 to **SampleDb** (and set the location, too, if you wish). Check the Create New Solution checkbox, and if you have an SCC provider for Visual Studio, check Add To Source Control as well. Then click Start.

If you checked Add To Source Control, you will be prompted at this point to supply credentials and server information (this will depend on your default SCC provider in Visual Studio). It takes a few moments for Visual Studio to scour the database, discover its schema, and generate the declarative T-SQL source files for the new project. When done, Visual Studio displays a window with information describing all the actions it took to create the project. Click Finish to close this window. The new project is then opened in the Solution Explorer automatically (docked to the right, by default). The dbo folder represents the *dbo* schema in the database. Expand it, and then expand the Tables and Views folders, as shown in Figure 1-12.

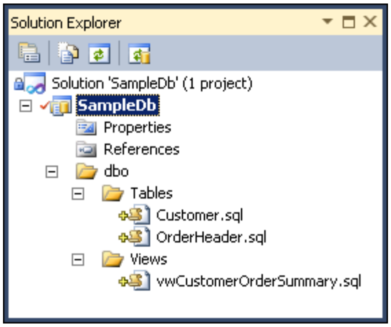


FIGURE 1-12 The source-controlled SQL Server Database Project after importing a database.

SSDT set up your project this way because the Folder Structure setting in the Import Database dialog (Figure 1-11) was set to Schema\Object Type. This tells SSDT to create a folder for each schema, and then a folder for each object type (table, view, and so on) contained in that schema. You are free to create additional folders as you extend your project. Unless you have a very specific or unique convention, it is best practice to maintain a consistent project structure based on the schema organization in the database like we've shown here.

More Information Schemas in SQL Server bear similarity to namespaces in .NET. Our simple example database has only a single namespace (*dbo*, short for database owner), but more complex databases typically consolidate database objects into multiple schemas. Just as namespaces are used to organize many classes in large .NET applications, schemas help manage many objects defined in large databases.

Like classes and namespaces, two database objects can be assigned the same name if they are contained in two different schemas. For example, both *Sales.Person* and *Billing.Person* refer to two completely different *Person* tables (one in the *Sales* schema and one in the *Billing* schema). SQL Server schemas can define objects at just a single level however, whereas .NET namespaces can be nested in as many levels as desired to define an elaborate hierarchy of classes.

Taking a Snapshot

Before you make any offline database changes, take a snapshot. This will create a single-file image of the current database schema that you can refer to or revert to at any time in the future. You'll take another snapshot when you've completed all your database changes, and thereby preserve the two points in time— just before, and just after, the changes are made. And because they are maintained as part of the project, snapshot files are also protected under source control.

Right-click the *SampleDb* project in Solution Explorer and choose Snapshot Project. After validating the project, Visual Studio creates a new Snapshots folder and, inside the Snapshots folder, it creates a new *.dacpac* file with a filename based on the current date and time. You'll usually want to give the snapshot a better name, so rename the file to **Version1Baseline.dacpac**.

Using the Table Designer (Offline Database Project)

With your "baseline" snapshot saved, you're ready to create the new *CustomerRanking* table. Recall that this is the new reference table based on the spreadsheet in Figure 1-7. In Solution Explorer, right-click the project's Tables folder (under the dbo schema folder) and choose Add | Table. Name the table **CustomerRanking** and click Add.

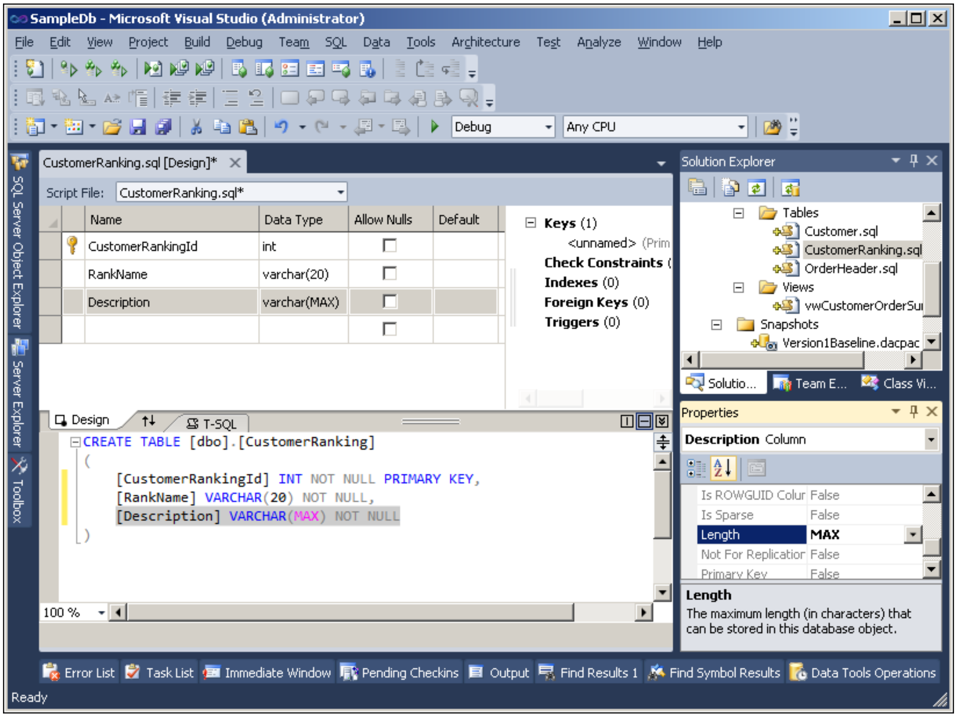
A new offline table designer window opens. You'll find that it looks and feels exactly the same as the one in Figure 1-8 that you used when working online. That's because it *is* the same tool, only this time it's the designer over a model backed by a source-controlled project file (*CustomerRanking.sql*) rather than a model backed by a live table. Because there's no connected database, the table designer has no Update button—instead, when working offline, schema changes are saved to the project script file for that table. This in turn updates the model, and then the same validation checks and IntelliSense you experienced when working while connected are run against the project. So you can find out right away if and when you make a breaking change, before deploying to a real database.

Earlier, when you removed the *CustomerRanking* column from the *Customer* table, we mentioned that you can design the table using either the grid in the top pane or the T-SQL code window in the bottom pane. You can also view and change parts of the schema definition from the Properties grid. We'll demonstrate all of these techniques now as you lay out the schema of the new *CustomerRanking* table.

SSDT starts you off with a new table that has one column in it: an *int* primary key named *Id*. To rename this column to *CustomerRankingId*, select the column name *Id* in the top pane's grid, replace it with **CustomerRankingId**, and press **Enter**. Beneath it, add the **RankName** column, set its data type to **varchar(20)**, and uncheck Allow Nulls. You can see that SSDT updates the T-SQL code in the bottom pane with a *CREATE TABLE* statement that reflects the changes made in the top pane.

Add the third column by editing the T-SQL code in the bottom pane. Append a comma after the second column and type **[Description] VARCHAR(200) NOT NULL**. As expected, the grid in the top pane is updated to show the new *Description* column you just added in code.

Finally, tweak the data type using the Properties grid. Click the *Description* column in the top pane and scroll to the Length property in the Properties grid (to display the Properties grid if it's not currently visible, click View and choose Properties Window). Click the drop-down list and select MAX to change the data type from *varchar(200)* to *varchar(max)*. When you're done, the table designer should look similar to Figure 1-13.



Script File: CustomerRanking.sql*

Name	Data Type	Allow Nulls	Default
CustomerRankingId	int	☐	
RankName	varchar(20)	☐	
Description	varchar(MAX)	☐	
		☐	

Keys (1)
<unnamed> (Prim
Check Constraints (
Indexes (0)
Foreign Keys (0)
Triggers (0)

```
CREATE TABLE [dbo].[CustomerRanking]
(
    [CustomerRankingId] INT NOT NULL PRIMARY KEY,
    [RankName] VARCHAR(20) NOT NULL,
    [Description] VARCHAR(MAX) NOT NULL
)
```

Properties

Description Column

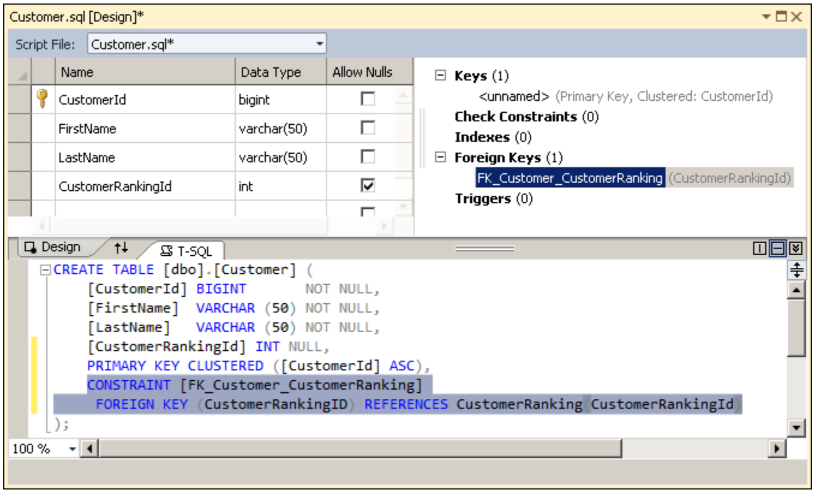
Is ROWGUID Colur	False
Is Sparse	False
Length	MAX
Not For Replicatior	False
Primary Key	False

Length
The maximum length (in characters) that can be stored in this database object.

FIGURE 1-13 The table designer for the new *CustomerRanking* table in an offline SQL Server Database Project.

Save *CustomerRanking.sql* and close the table designer. This completes step 2. You are now ready to add the foreign key column to the *Customer* table (step 3) that joins to this new *CustomerRanking* table. Double-click *Customer.sql* in Solution Explorer to open the table designer for the *Customer* table. Use any technique you'd like to add a new nullable *int* column named **CustomerRankingId** (it must be nullable at this point, because it doesn't have any data yet).

Now you can establish the foreign key relationship to the *CustomerRanking* table (step 4). In the upper-righthand corner of the Table Designer is a Context View area that summarizes other pertinent objects related to the table. In the Context View, right-click Foreign Keys and choose Add New Foreign Key. Name the new foreign key **FK_Customer_CustomerRanking** (it is best practice to assign foreign key names that indicate which tables participate in the relationship). Then edit the *FOREIGN KEY* template code added to the T-SQL code window in the bottom pane to be **FOREIGN KEY (CustomerRankingID) REFERENCES CustomerRanking(CustomerRankingId)**. The table designer window should now look similar to Figure 1-14. After reviewing the table schema, save the *Customer.sql* file and close the designer.



	Name	Data Type	Allow Nulls
🔑	CustomerId	bigint	□
	FirstName	varchar(50)	□
	LastName	varchar(50)	□
	CustomerRankingId	int	☑
			□

- **Keys** (1)
  - <unnamed> (Primary Key, Clustered: CustomerId)
- **Check Constraints** (0)
- **Indexes** (0)
- **Foreign Keys** (1)
  - FK_Customer_CustomerRanking (CustomerRankingId)
- **Triggers** (0)

```
CREATE TABLE [dbo].[Customer] (
    [CustomerId] BIGINT        NOT NULL,
    [FirstName]  VARCHAR (50) NOT NULL,
    [LastName]   VARCHAR (50) NOT NULL,
    [CustomerRankingId] INT NULL,
    PRIMARY KEY CLUSTERED ([CustomerId] ASC),
    CONSTRAINT [FK_Customer_CustomerRanking]
     FOREIGN KEY (CustomerRankingID) REFERENCES CustomerRanking(CustomerRankingId)
);
```

FIGURE 1-14 The table designer for the *Customer* table after creating the foreign key on *CustomerRankingId*.

Introducing LocalDB

Your next tasks involve altering a view (step 5) and creating a stored procedure (step 6). It will be very helpful to have a test SQL Server environment available as you implement these steps. You don't want to use *SQL2012DEV*, because that's the "live" server. You need another SQL Server that can be used just for testing offline.

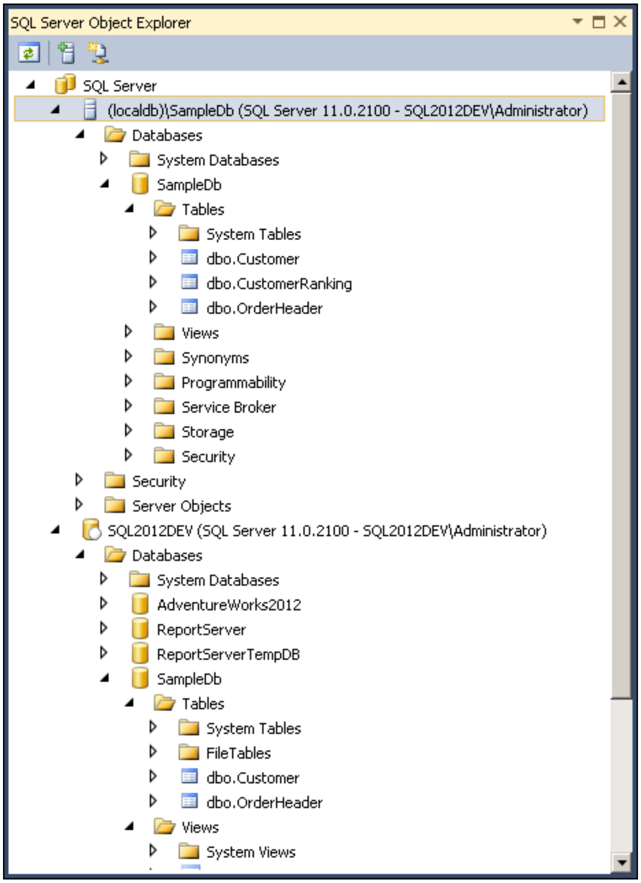
LocalDB gives you that test environment. This is a new, lightweight, single-user instance of SQL Server that spins up on demand when you build your project. This is extremely handy when working offline and there is no other development server available to you. The official name for this new variant of SQL Server is "SQL Express LocalDB," which can be misleading because it is distinct from the Express edition of SQL Server. To avoid confusion, we refer to it simply as "LocalDB."

> **Note** The new LocalDB does not support every SQL Server feature (for example, it can't be used with FILESTREAM). However, it does support most functionality required for typical database development.

Press **F5** to build the project. This first validates the entire database structure defined by the project and then deploys it to LocalDB. Note, however, that this is just the default behavior; you can change the project properties to target another available server for testing if you require features not supported by LocalDB (for example, FILESTREAM, which we cover in Chapter 8).

The deployment is carried out by performing a schema compare between the project and LocalDB on the target server. More precisely, and as already explained, models of the source project and target database are generated, and the schema compare actually works on the two models. Being your very first build, the database does not exist yet on the target server, so the schema compare generates a script that creates the whole database from scratch. As you modify the project, subsequent builds will generate incremental change scripts that specify just the actions needed to bring the target database back in sync with the project.

Look back over in SQL Server Object Explorer and you'll see that SSDT has started a new LocalDB instance. The host name is *(localdb)\SampleDb*, and it is a completely separate instance than the *SQL2012DEV* instance (which has not yet been updated with the new *CustomerRanking* table and the *CustomerRankingId* foreign key in the *Customer* table). Figure 1-15 shows *SampleDb* deployed to LocalDB, expanded to reveal its tables. Notice that it does include the new *CustomerRanking* table.

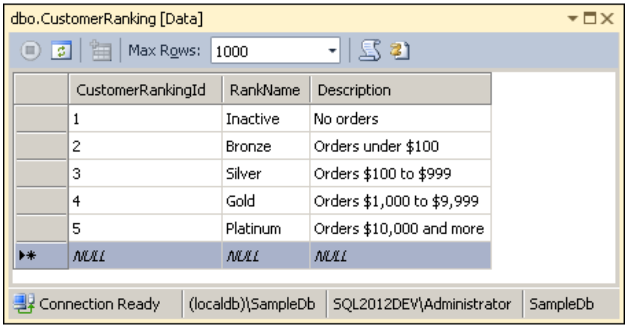


```
◢ SQL Server
  ◢ (localdb)\SampleDb (SQL Server 11.0.2100 - SQL2012DEV\Administrator)
    ◢ Databases
      ▷ System Databases
      ◢ SampleDb
        ◢ Tables
          ▷ System Tables
          ▷ dbo.Customer
          ▷ dbo.CustomerRanking
          ▷ dbo.OrderHeader
        ▷ Views
        ▷ Synonyms
        ▷ Programmability
        ▷ Service Broker
        ▷ Storage
        ▷ Security
    ▷ Security
    ▷ Server Objects
  ◢ SQL2012DEV (SQL Server 11.0.2100 - SQL2012DEV\Administrator)
    ◢ Databases
      ▷ System Databases
      ▷ AdventureWorks2012
      ▷ ReportServer
      ▷ ReportServerTempDB
      ◢ SampleDb
        ◢ Tables
          ▷ System Tables
          ▷ FileTables
          ▷ dbo.Customer
          ▷ dbo.OrderHeader
        ◢ Views
          ▷ System Views
```

FIGURE 1-15 The local database runtime (LocalDB) after building and deploying the SQL Server Database Project.

Now you have a test database to play around with, but of course there's no data in it. You will add some now so that you can test the view you're about to alter and the stored procedure you're about to create. Using simple copy/paste, SSDT lets you import small sets of rows from any "table" source (including Microsoft Word and Excel) into a database table that has compatible columns.

First, bring in the reference data from the ranking definitions provided by the spreadsheet in Figure 1-7. You can easily grab the data straight out of the spreadsheet and dump it right into the new *CustomerRanking* table. Open the spreadsheet in Excel, select the five rows of data (complete rows, not cells or columns), then right-click the selection and choose Copy. Back in SQL Server Object Explorer, right-click the *CustomerRanking* table and choose View Data. The Editable Data Grid in SSDT opens with a template for inserting a new row. Right-click the row selector for the new row template and choose Paste (be sure to right-click the row selector in the left gray margin area, and not a cell, before pasting). As shown in Figure 1-16, SSDT faithfully copies the data from Excel into the *CustomerRanking* table.



Max Rows: 1000

	CustomerRankingId	RankName	Description
	1	Inactive	No orders
	2	Bronze	Orders under $100
	3	Silver	Orders $100 to $999
	4	Gold	Orders $1,000 to $9,999
	5	Platinum	Orders $10,000 and more
▶*	*NULL*	*NULL*	*NULL*

Connection Ready | (localdb)\SampleDb | SQL2012DEV\Administrator | SampleDb

FIGURE 1-16 Reference data imported into a database table from Excel via the clipboard.

You also need a few customers to work with. Using the same copy/paste technique, you will transfer rows from the *Customer* table in the production database to the test database on LocalDB (for this exercise, you won't transfer related order data in the *OrderHeader* table). There are only a handful of customers, so you'll just copy them all over. Typically though, you'd extract just the subset of data that provides a representative sampling good enough for testing purposes. Expand the production server (*SQL2012DEV*) node in SQL Server Object Explorer and drill down to the *Customer* table. Right-click the table and choose View Data. Select all the customer rows, then right-click the selection and choose Copy. Next, right-click the *Customer* table in the test database on *(localdb)\SampleDb* and choose View Data. Right-click the new row selector and choose Paste to copy in the six customer rows.

You are now at step 5, which is to update the *vwCustomerOrderSummary* view. Recall that this is the same view you edited back in step 1 (while connected), when you removed the schema-bound reference to the old *CustomerRanking* column that was being dropped from the *Customer* table. With the new reference table and foreign key relationship now in place, you will revise the view once again (offline, this time) to join with the *CustomerRanking* table on *CustomerRankingId*, so that it can expose the display name in the reference table's *RankName* column.

In Solution Explorer, double-click *vwCustomerOrderSummary.sql* in the project's Views folder (under the dbo schema folder). The view opens up in a new code window, and your attention may first be drawn to several squigglies that Visual Studio paints in the view's code. They're not red, because there is really nothing significantly wrong with the view, and so these are just warnings. Hover the cursor over one of them to view the warning text in a tooltip (you can also see all of them listed as warning items in the Error List pane). The warnings indicate that the view uses *CustomerID* (ending in a capital D) to reference a column that is actually defined as *CustomerId* (ending in a lowercase d). These are the same case-sensitivity warnings you saw earlier when attempting to update the database with dependency issues. Object names in SQL Server are normally not case-sensitive (like VB .NET), but non-default collation settings can change that behavior so that they *are* case-sensitive (like C#). This would cause a problem if you deployed the view to a SQL Server instance configured for a case-sensitive collation of object names.

Add another *LEFT OUTER JOIN* to the view to add in the *CustomerRanking* table joined on the *CustomerRankingId* of the *Customer* table, and add *RankName* to the *SELECT* and *GROUP BY* column lists. You want your code to be squeaky-clean, so now is also a good time to resolve those case-sensitivity warnings. Replace *CustomerID* with *CustomerId* in the four places that it occurs (once in the *SELECT* column list, twice in the first *JOIN*, and once more in the *GROUP BY* column list). Listing 1-3 shows the view definition after making the changes.

LISTING 1-3 The updated *vwCustomerOrderSummary* view definition joining on the new *CustomerRanking* table.

```
-- Create a handy view summarizing customer orders
CREATE VIEW vwCustomerOrderSummary WITH SCHEMABINDING AS
 SELECT
   c.CustomerId, c.FirstName, c.LastName, r.RankName,
   ISNULL(SUM(oh.OrderTotal), 0) AS OrderTotal
  FROM
   dbo.Customer AS c
   LEFT OUTER JOIN dbo.OrderHeader AS oh ON c.CustomerId = oh.CustomerId
   LEFT OUTER JOIN dbo.CustomerRanking AS r ON c.CustomerRankingId =
      r.CustomerRankingId
   GROUP BY
      c.CustomerId, c.FirstName, c.LastName, r.RankName
```

Save the *vwCustomerOrderSummary.sql* file to update the offline project. You know that pressing F5 now will deploy the changed view to the test database on LocalDB. But what if you attempt to execute the script directly by pressing **Ctrl+Shift+E**, right here in the code window? Go ahead and try. You'll receive this error message in response:

```
Msg 2714, Level 16, State 3, Procedure vwCustomerOrderSummary, Line 2
There is already an object named 'vwCustomerOrderSummary' in the database.
```

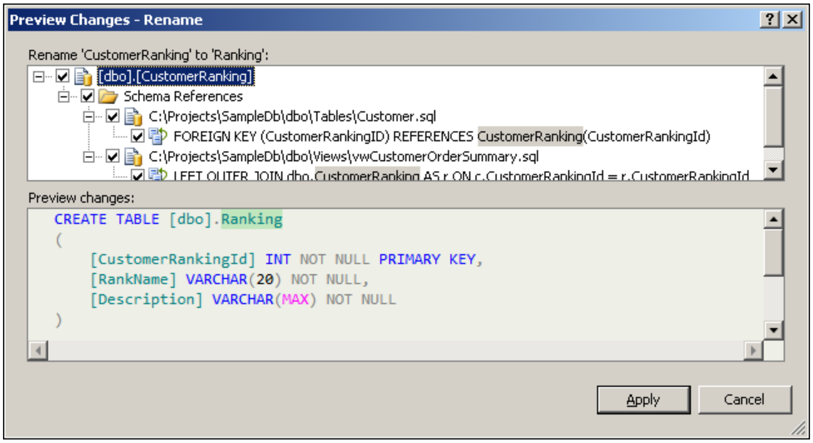
Here's what happened. First, SSDT connected the query window to the *(localdb)\SampleDb* instance. Then it attempted to execute the script imperatively against the connected database, just as you've already seen with Ctrl+Shift+E. But being part of an offline project, this script is declarative and so it's expressed as a *CREATE VIEW* statement. The view already exists in the database, so the error message makes perfect sense. Again, the proper way to update the database is to deploy it via an incremental deployment script by debugging with F5.

However, you are indeed connected to the test database on LocalDB, even though you're working inside the query window of an offline project that hasn't yet been deployed to LocalDB. This means that you can actually test the view before you deploy it. Select all the text from *SELECT* until the end of the script (that is, leave only the *CREATE VIEW* portion of the window unselected) and press **Ctrl+Shift+E** again. This time, you get a much better result. Only the chosen *SELECT* statement executes, which is perfectly valid T-SQL for the connected database. The query results show you what the view is going to return, and you got that information without having to deploy first. In this mode, you are actually working connected and offline simultaneously! You can select any T-SQL to instantly execute it, test and debug stored procedures, and even get execution plans, all while "offline."

Refactoring the Database

The view is ready to be deployed, but now you decide to change some names first. Customers are the only thing being ranked, so shortening the table name *CustomerRanking* to *Ranking* and column names *CustomerRankingId* to *RankingId* is going to make your T-SQL more readable (which is important!). Without the proper tooling, it can be very tedious to rename objects in the database. But the refactoring capabilities provided by SSDT make this remarkably easy.

In the new *LEFT OUTER JOIN* you just added, right-click on the *CustomerRanking* table reference, and then choose Refactor, Rename. Type **Ranking** for the new table name and click OK. You are presented with a preview window (Figure 1-17) that will appear very familiar if you've ever used the refactoring features in Visual Studio with ordinary .NET projects.



Rename 'CustomerRanking' to 'Ranking':

- ☑ [dbo].[CustomerRanking]
  - ☑ Schema References
    - ☑ C:\Projects\SampleDb\dbo\Tables\Customer.sql
      - ☑ FOREIGN KEY (CustomerRankingID) REFERENCES CustomerRanking(CustomerRankingId)
    - ☑ C:\Projects\SampleDb\dbo\Views\vwCustomerOrderSummary.sql
      - ☑ LEFT OUTER JOIN dbo.CustomerRanking AS r ON c.CustomerRankingId = r.CustomerRankingId

Preview changes:

```
CREATE TABLE [dbo].Ranking
(
    [CustomerRankingId] INT NOT NULL PRIMARY KEY,
    [RankName] VARCHAR(20) NOT NULL,
    [Description] VARCHAR(MAX) NOT NULL
)
```

FIGURE 1-17 Previewing changes before refactoring is applied to the database.

This dialog shows all the references to the *CustomerRanking* table that will be changed to *Ranking* when you click Apply (notice that checkboxes are provided so that you can also choose which references should get updated and which should not). Scroll through the change list to preview each one, and then click Apply to immediately invoke the rename operation. Every affected project file is updated accordingly, but Visual Studio won't actually rename project files themselves. The project script file defining the newly renamed *Ranking* table is still named *CustomerRanking.sql*. Right-click the file in Solution Explorer, choose Rename, and change the filename to **Ranking.sql**.

Now rename the primary key column in the *Ranking* table along with its corresponding foreign key column in the *Customer* table, both of which are currently named *CustomerRankingId*. The two key columns are referenced on the same *LEFT OUTER JOIN* line, so this will be easy. Right-click the *r.CustomerRankingId* key column in the join and choose Refactor, Rename. Type **RankingId** for the new name, click OK, preview the changes, and click Apply to update the primary key column name in the *Ranking* table. Then repeat for the *c.CustomerRankingId* key column to update the foreign key column name in the *Customer* table (the actual order in which you refactor the column names in these tables is immaterial).

There's one more thing to rename, and that's the foreign key definition in the *Customer* table. This isn't strictly necessary of course, but the (self-imposed) convention to name foreign keys definitions after the tables they join dictates that *FK_Customer_CustomerRanking* should be renamed to *FK_Customer_Ranking*. The *Customer* table is specified first in the view's *FROM* clause, so right-click on it now and choose Go to Definition. This navigates directly to the *Customer* table definition in a new query window. In the *CONSTRAINT* clause, right-click *FK_Customer_CustomerRanking* and choose Refactor, Rename. Type **FK_Customer_Ranking** for the new name, click OK, preview the changes (just one, in this case), and click Apply.

You're all set to deploy the changes with another build, so press **F5** once again. After the build completes, click Refresh in the SQL Server Object Explorer toolbar and look at the test database running under *(localdb)\SampleDb* to confirm that the *CustomerRanking* table has been renamed to *Ranking*. Right-click the *Ranking* table and choose View Data to confirm that all the data in the renamed table is intact. When you rename objects in a SQL Server Database Project, SSDT generates a change script with corresponding *EXECUTE sp_rename* statements in it, as opposed to dropping one object and creating another (which, for tables, would result in irrevocable data loss). So the tool does the right thing, relying ultimately on the SQL Server *sp_rename* stored procedure to properly change the object's name internally within the database.

It's time to create the stored procedure that ranks the customers. First, create a **Stored Procedures** folder beneath the dbo folder in Solution Explorer (to do this, right-click the dbo folder, and choose Add | New Folder). This folder would have already been created when you imported the database into the project, had there been any stored procedures in the database at the time. Then right-click the new Stored Procedures folder and choose Add | Stored Procedure. Name the stored procedure **uspRankCustomers** and click Add. SSDT creates a new file named *uspRankCustomers.sql* and opens it in a new T-SQL editor window. Replace the template code with the script shown in Listing 1-4 and save it, but keep the window open. Now press **F5** to perform another build and push the new stored procedure out to the test database on LocalDB.

LISTING 1-4 The stored procedure to rank customers based on their total order amount.

```
CREATE PROCEDURE uspRankCustomers
AS
        DECLARE @CustomerId int
        DECLARE @OrderTotal money
        DECLARE @RankingId int

        DECLARE curCustomer CURSOR FOR
         SELECT CustomerId, OrderTotal FROM vwCustomerOrderSummary

        OPEN curCustomer
        FETCH NEXT FROM curCustomer INTO @CustomerId, @OrderTotal
        WHILE @@FETCH_STATUS = 0
        BEGIN
                IF @OrderTotal = 0 SET @RankingId = 1
                ELSE IF @OrderTotal < 100 SET @RankingId = 2
                ELSE IF @OrderTotal < 1000 SET @RankingId = 3
                ELSE IF @OrderTotal < 10000 SET @RankingId = 4
                ELSE SET @RankingId = 5

                UPDATE Customer
                 SET RankingId = @RankingId
                 WHERE CustomerId = @CustomerId

                FETCH NEXT FROM curCustomer INTO @CustomerId, @OrderTotal
        END

        CLOSE curCustomer
        DEALLOCATE curCustomer
```

This stored procedure “ranks“ the customers, examining them individually and assigning each a value based on their order total. It does this by opening a cursor against the order summary view, which returns one row per customer with their individual orders aggregated into a single order total. Based on the dollar value of the total, it then updates the customer with a ranking value between one and five. Then it advances to the next customer until it reaches the end of the cursor. As mentioned at the outset, this solution may be a bit contrived (and we’re sure you can think of a better approach), but it suits our demonstration purposes here just fine.

Testing and Debugging

Are you in the habit of running new or untested code on live databases? We certainly hope not. Though you could, you should not simply push all of the changes you’ve made in the project (steps 2 through 6) back to the live database on *SQL2012DEV*, and then run the stored procedure there for the very first time. It’s much safer to test the stored procedure offline first with LocalDB. You will now learn how to do that using the integrated debugger in Visual Studio. Then you can confidently deploy everything back to *SQL2012DEV*, and finally (step 7), to the cloud!

The *uspRankCustomers* stored procedure is still open in the code editor. Click inside the left margin on the *OPEN curCustomer* line to set a breakpoint just before the cursor is opened. The breakpoint appears as a red bullet in the margin where you clicked. This is exactly how breakpoints are set in C# or VB .NET code, and SSDT now delivers a similar debugging experience for T-SQL code as well. In SQL Server Object Explorer, expand the *Stored Procedures* node (located beneath Programmability, just as in SSMS) for *SampleDb* beneath the LocalDB instance. Right-click the *Stored Procedures* node, choose Refresh, and you will see the *uspRankCustomers* stored procedure you just deployed. Right-click on the stored procedure and choose Debug Procedure. SSDT generates an *EXEC* statement to invoke *uspRankCustomers* and opens it in a new query window. The debugger is already started, and is paused on the *USE [SampleDb]* statement above the *EXEC* statement.

More Info The debugging session began instantly in this case because the *uspRankCustomers* stored procedure being debugged has no parameters. When stored procedure parameters are expected, SSDT will first display a dialog to solicit the parameter values, and then plug those values into the EXEC statement before starting the debugger.

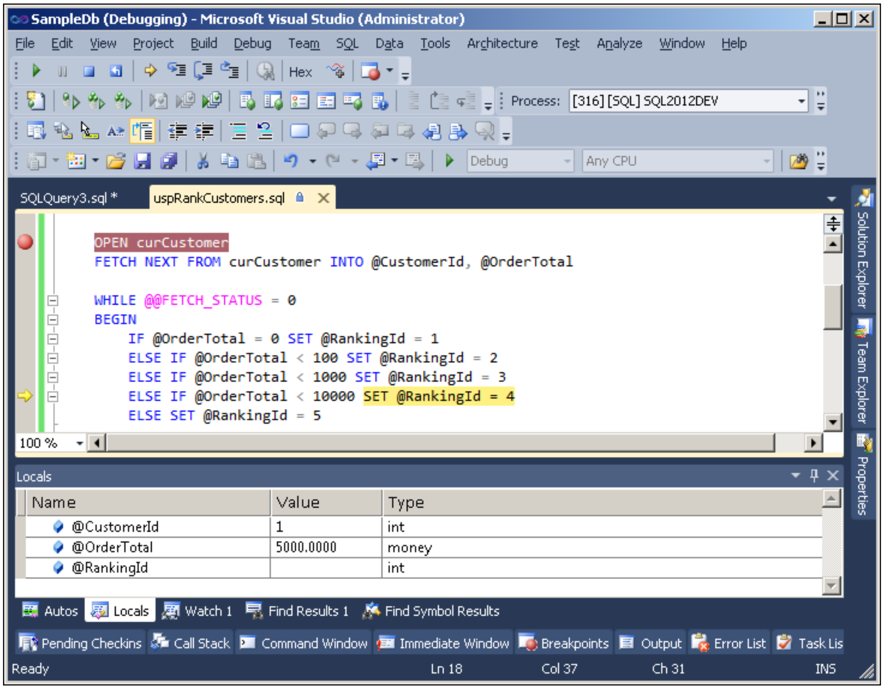
Press **F5** to continue execution. The debugger reaches the *EXEC* statement, enters the stored procedure, and then breaks on the *OPEN curCustomer* statement where you previously set the breakpoint. Now start single stepping through the stored procedure's execution with the debugger's F10 keystroke. Press **F10** three times to step over the next three statements. This opens the cursor, fetches the first customer from it, and you are now paused on the first *IF* statement that tests the first customer's order total for zero dollars.

Earlier, you copied six customer rows from the *SQL2012DEV* database to LocalDB, but we specifically instructed you not to copy any order data. So this loop will iterate each customer, and (based on an order total of zero dollars) assign a ranking value of 1 for every customer. Rather than interrupting your debugging session now to import some sample order data and start over, you will use the debugger's Locals window to simulate non-zero order totals for the first two customers. Click the Debug menu, and then choose Windows | Locals.

In the Locals window, you can see that *@CustomerId* is 1 (this is the first customer) and *@OrderTotal* is 0 (expected, because there's no sample order data). *@RankingId* is not yet set, but if you allow execution to continue as-is, the customer will be ranked with a 1. Double-click the 0.0000 value for *@OrderTotal* in the Locals window, type **5000** and press **Enter**. Now the stored procedure thinks that the customer actually has $5,000 in total orders. Press **F10** to single step. Because *@OrderTotal* no longer equals zero, execution advances to the next *IF* condition that tests the order total for being under $100. Press **F10** again and execution advances to the next *IF* condition that tests for under $1,000. Press **F10** once more to reach the *IF* condition testing for under $10,000. This condition yields true (there are $5,000 in total orders), so pressing **F10** to single step once more advances to the *SET* statement that assigns a ranking value of 4. This is the correct value for orders in the range of $1,000 to $10,000. Figure 1-18 shows the debugging session paused at this point.

Continue pressing **F10** to single step through the remaining *SET*, *UPDATE*, and *FETCH NEXT* statements, and then back up again to the first *IF* statement testing the second customer's order total value for zero dollars. Use the Locals window to fake another amount; this time change *@OrderTotal* to **150**. Single step a few more times to make sure that this results in the stored procedure assigning a ranking value of 3 this time, which is the correct value for orders in the range of $100 to $1,000. Now press **F5** to let the stored procedure finish processing the rest of the customers with no more intervention on your part.

When the stored procedure completes execution, right-click the *Customer* table in SQL Server Object Explorer (be sure to pick the LocalDB instance and not *SQL2012DEV*) and choose View Data. The table data confirms that the first customer's ranking was set to 4, the second customer's ranking was set to 3, and all the other customer rankings were set to 1 (if you already have a *Customer* table window open from before, the previous values will still be displayed; you need to click the Refresh button in the toolbar to update the display).


SampleDb (Debugging) - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team SQL Data Tools Architecture Test Analyze Window Help
Hex
Process: [316] [SQL] SQL2012DEV
Debug
Any CPU
SQLQuery3.sql *
uspRankCustomers.sql
100 %
Solution Explorer
Team Explorer
Properties
Autos Locals Watch 1 Find Results 1 Find Symbol Results
Pending Checkins Call Stack Command Window Immediate Window Breakpoints Output Error List Task Lis
Ready Ln 18 Col 37 Ch 31 INS

```
OPEN curCustomer
FETCH NEXT FROM curCustomer INTO @CustomerId, @OrderTotal

WHILE @@FETCH_STATUS = 0
BEGIN
    IF @OrderTotal = 0 SET @RankingId = 1
    ELSE IF @OrderTotal < 100 SET @RankingId = 2
    ELSE IF @OrderTotal < 1000 SET @RankingId = 3
    ELSE IF @OrderTotal < 10000 SET @RankingId = 4
    ELSE SET @RankingId = 5
```

Locals

Name	Value	Type
@CustomerId	1	int
@OrderTotal	5000.0000	money
@RankingId		int

FIGURE 1-18 T-SQL debugging session of a stored procedure in Visual Studio.

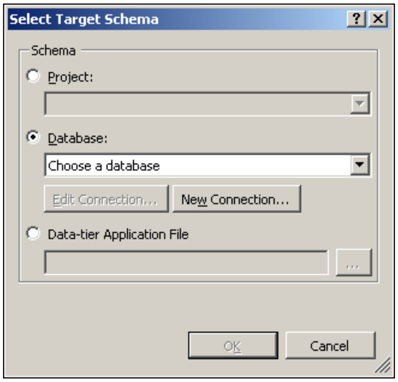
This was by no means exhaustive testing, but it will suffice for demonstration purposes. The key point is that SSDT provides an environment you can use for debugging and testing as you develop your database offline, until you're ready to deploy to a live environment (as you are now).

Comparing Schemas

You are ready to deploy to the database back to the live server on *SQL2012DEV*. As you may have correctly surmised by now, the process is fundamentally the same as working offline with LocalDB each time F5 is pressed: SSDT runs a schema compare to generate a change script. The project properties (by default) specify a connection string that points to LocalDB. So building with F5 uses the test database as the target for the schema compare with the project as the source, and then executes the generated change script against the test database on LocalDB. This all happens as a completely unattended set of operations every time you press F5.

Now you will carry out those very same steps once again, only this time you'll get more involved in the process. In particular, you will specify the live *SQL2012DEV* instance as the target for the schema compare, rather than LocalDB. You will also review the results of the schema compare, and have the chance to choose to deploy or not deploy specific detected changes. Finally, you'll get the opportunity to view, edit, save, and execute the change script after it is generated, rather than having it execute automatically. So there's a bit more intervention involved in the process now, but you *want* it that way. The schema compare process itself is the same as the F5 build—you just get to exercise more control over it to support different deployment scenarios.

Right-click the *SampleDb* project in Solution Explorer and choose Schema Compare to open a new schema compare window. You need to specify a source and target for any schema compare, naturally. Because you launched the window from the SQL Server Database Project context menu in Solution Explorer, Visual Studio sets the source to the project automatically, leaving you to set just the target. To set the target, click its drop-down list and choose Select Target to display the Select Target Schema dialog, shown in Figure 1-19.



Select Target Schema

Schema

○ Project:

● Database:

Choose a database

○ Data-tier Application File

FIGURE 1-19 SSDT lets you choose between projects, databases, and snapshots for schema compare operations.

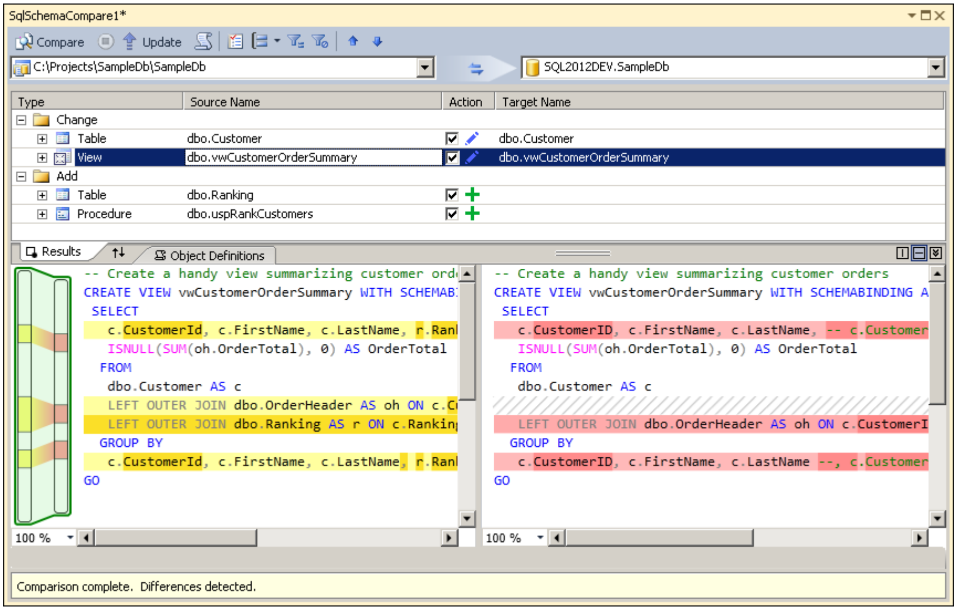
Notice how you can choose between three schemas for the target—a project, a database, or a data-tier application file (snapshot). The same choices are also supported for the source, although the SQL Server Database Project was assumed as the source automatically in this case. Any combination of source and target source schemas is supported; SSDT simply creates source and target models from your choice of backings. Then, working off the models, it shows you the differences and generates a change script for you. This flexibility is a major benefit of model-based database development with SSDT.

The Select Target Schema dialog has correctly assumed that you want to use a database as the target. All you need to do is choose the live *SampleDb* database running on *SQL2012DEV*. Click New Connection to open a Connection Properties dialog, type your actual machine name for the server name (which is *SQL2012DEV* in the current example), choose *SampleDb* from the database name drop-down list, and click OK. (Visual Studio will remember this connection for the future, and make it available for recall in the database dropdown the next time you run a schema compare.) Click OK once more, and then click the Compare button in the toolbar to start the schema compare.

It takes just a few moments for the operation to complete. When it finishes, the schema compare displays all the changes you've made offline since creating the project (steps 2 through 6). The report lets you see each added, changed, and dropped object, and it can be organized by type (table, view, and so on), schema, or action (change, add, or delete). Selecting any object in the top pane presents its T-SQL declaration in the bottom pane, side-by-side (source on the left, target on the right), with synchronized scrollbars. The T-SQL is color-coded to highlight every one of the object's differences.

If desired, you can exclude specific objects from the change script (which hasn't been generated yet) by clearing their individual checkboxes back up in the top pane.

Select the *vwCustomerOrderSummary* view in the top pane to see the source and target versions of the view in the bottom pane. As shown in Figure 1-20, the rich visual display rendered by the schema compare tool makes it easy to identify all the changes made to the view.



Type	Source Name	Action	Target Name
⊟ Change			
⊞ Table	dbo.Customer	☑ ✎	dbo.Customer
⊞ View	dbo.vwCustomerOrderSummary	☑ ✎	dbo.vwCustomerOrderSummary
⊟ Add			
⊞ Table	dbo.Ranking	☑ +	
⊞ Procedure	dbo.uspRankCustomers	☑ +	

```
-- Create a handy view summarizing customer ord
CREATE VIEW vwCustomerOrderSummary WITH SCHEMAB
 SELECT
   c.CustomerId, c.FirstName, c.LastName, r.Ranl
   ISNULL(SUM(oh.OrderTotal), 0) AS OrderTotal
  FROM
   dbo.Customer AS c
   LEFT OUTER JOIN dbo.OrderHeader AS oh ON c.Cu
   LEFT OUTER JOIN dbo.Ranking AS r ON c.Rankin
  GROUP BY
   c.CustomerId, c.FirstName, c.LastName, r.Ranl
GO
```

```
-- Create a handy view summarizing customer orders
CREATE VIEW vwCustomerOrderSummary WITH SCHEMABINDING A
 SELECT
   c.CustomerID, c.FirstName, c.LastName, -- c.Customer
   ISNULL(SUM(oh.OrderTotal), 0) AS OrderTotal
  FROM
   dbo.Customer AS c

   LEFT OUTER JOIN dbo.OrderHeader AS oh ON c.CustomerI
  GROUP BY
   c.CustomerID, c.FirstName, c.LastName --, c.Customer
GO
```

Comparison complete. Differences detected.

FIGURE 1-20 Viewing the schema differences between a SQL Server Database Project and a live database.

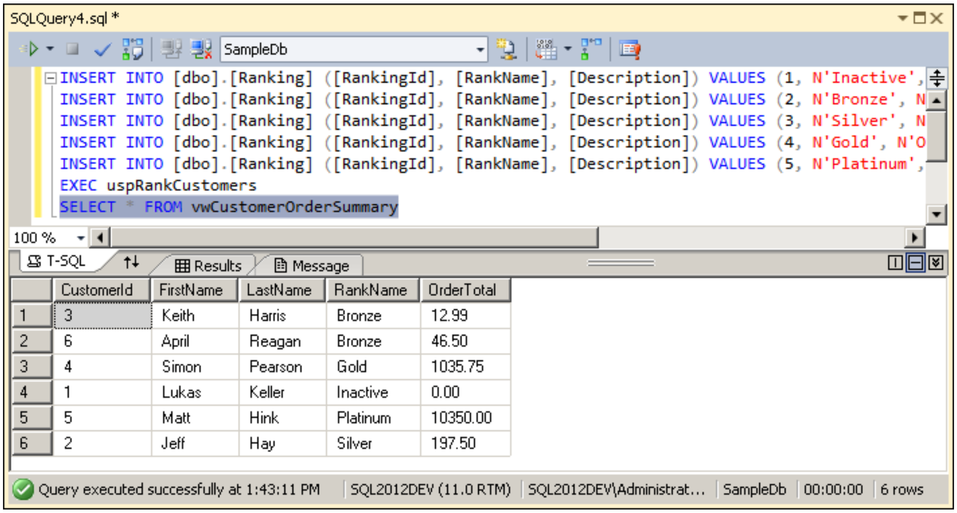
As with the table designer, you can choose to update the live database immediately by generating and running the change script without previewing it. Or you can choose to be more cautious, and just generate the change script. Then you can view, edit, and ultimately decide whether or not to execute it. Your confidence level should be very high by now, so just click the Update button in the toolbar (and then click Yes to confirm) to let it run. SSDT updates the target database and displays a completion message when it's done. Click OK to dismiss the message. The differences from before the update are still displayed in the window, now dimmed in gray (you can click Compare again to confirm that there are no longer any differences between the project and the live database on *SQL2012DEV*). In SQL Server Object Explorer, drill down on *SampleDb* under *SQL2012DEV* (or refresh already drilled down nodes) to verify that it reflects all the work performed in the project for steps 2 through 6 on your task list.

You are almost ready to run the new *uspRankCustomers* stored procedure and update the live *Customer* table, but there's one more thing to do before that. Although the deployment created the *schema* of the *Ranking* table, it didn't copy its *data*. You need to import the reference data from the spreadsheet in Figure 1-7 again, this time into the live database on *SQL2012DEV*. You can certainly use the same copy/paste trick we showed earlier when you imported the spreadsheet into the test database on LocalDB, but we'll take this opportunity now to show you how to script table data with SSDT.

Note The *Ranking* table is a typical example of reference data. Databases often rely on reference data, which are usually small sets of read-only entries, to help define their structure. Although technically not schema, it would certainly be convenient to mark the contents of the *Ranking* table as such, so that the five entries it contains become part of the SSDT database model and travel with the schema definition wherever you deploy it to. Unfortunately, this feature could not make it in time for the final product release, but Microsoft is evaluating plans to add this capability to a future version of SSDT.

Under the *(localdb)\SampleDb* node (the LocalDB instance) in SQL Server Object Explorer, right-click the *Ranking* table and choose View Data to open a window showing the five rows in the table. Next, click the Script button on the toolbar (the next to the last button). SSDT generates *INSERT* statements for the five rows of data in the *Ranking* table, and displays them in a new query window. You want to execute these *INSERT* statements in a query window connected to the live database on *SQL2012DEV*, so select all the *INSERT* statements and press **Ctrl+C** to copy them to the clipboard. Then under the *SQL2012DEV* node in SQL Server Object Explorer, right-click the *SampleDb* database and choose New Query. Press **Ctrl+V** to paste the *INSERT* statements into the new query window and then press **Ctrl+Shift+E** to execute them. The reference data has now been imported into the live database and you're ready to update the customers.

In the same query window, type **EXEC uspRankCustomers**, select the text of the statement, and press **Ctrl+Shift+E**. The stored procedure executes and updates the customers. (You can ignore the null value warning; it refers to the *SUM* aggregate function in the view, which does not affect the result.) To see the final result, type **SELECT * FROM vwCustomerOrderSummary**, select it, and press **Ctrl+Shift+E** once again. As shown in Figure 1-21, each customer's ranking is correctly assigned based on their total order amount.



```
INSERT INTO [dbo].[Ranking] ([RankingId], [RankName], [Description]) VALUES (1, N'Inactive',
INSERT INTO [dbo].[Ranking] ([RankingId], [RankName], [Description]) VALUES (2, N'Bronze', N
INSERT INTO [dbo].[Ranking] ([RankingId], [RankName], [Description]) VALUES (3, N'Silver', N
INSERT INTO [dbo].[Ranking] ([RankingId], [RankName], [Description]) VALUES (4, N'Gold', N'O
INSERT INTO [dbo].[Ranking] ([RankingId], [RankName], [Description]) VALUES (5, N'Platinum',
EXEC uspRankCustomers
SELECT * FROM vwCustomerOrderSummary
```

	CustomerId	FirstName	LastName	RankName	OrderTotal
1	3	Keith	Harris	Bronze	12.99
2	6	April	Reagan	Bronze	46.50
3	4	Simon	Pearson	Gold	1035.75
4	1	Lukas	Keller	Inactive	0.00
5	5	Matt	Hink	Platinum	10350.00
6	2	Jeff	Hay	Silver	197.50

Query executed successfully at 1:43:11 PM | SQL2012DEV (11.0 RTM) | SQL2012DEV\Administrat... | SampleDb | 00:00:00 | 6 rows

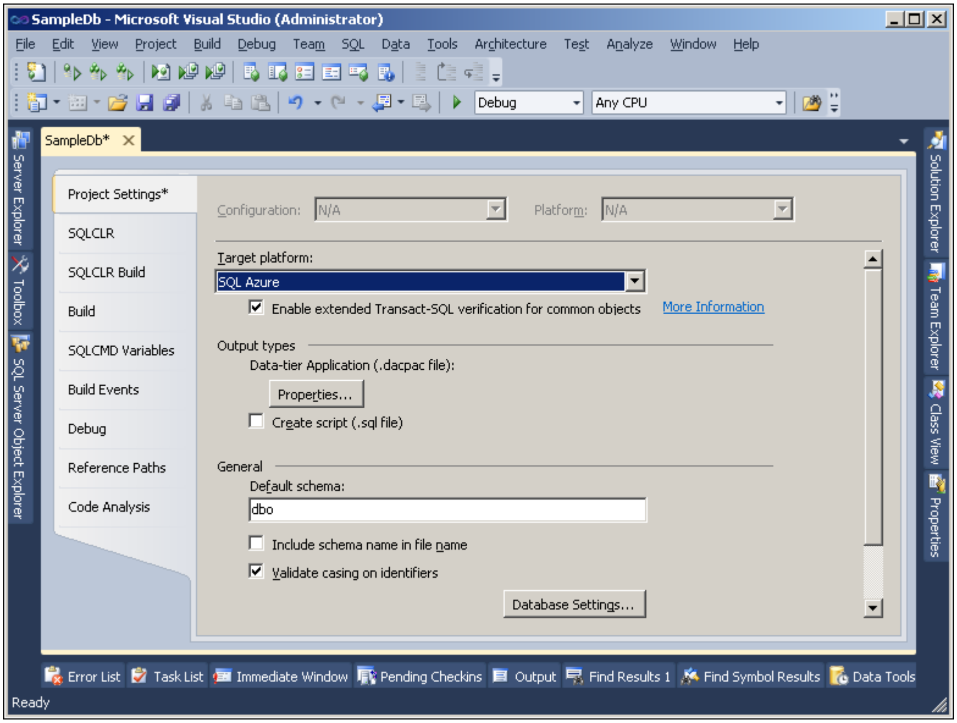
FIGURE 1-21 Viewing the final results of offline development in the live database.

Publishing to SQL Azure

The marketing team's last request was that you deploy a copy of the database to SQL Azure. To ensure that the database is cloud-ready, you just need to tell SSDT that you are targeting the SQL Azure platform by changing a property of the project. Then, if any SQL Azure compatibility issues are identified, they can be resolved before you deploy. As you might expect by now, you will use the very same techniques you've learned throughout this chapter to deploy the SQL Server Database Project to SQL Azure.

> **Note** Our discussion in this section assumes you already have an available SQL Azure server instance that you can publish to. SQL Azure server names always begin with a unique identifier randomly assigned just to you, followed by *.database.windows.net*. Chapter 12 (which is dedicated to SQL Azure) explains how to use the Azure Management Portal to create your own cloud databases on SQL Azure, after setting up a Windows Azure account.

Right-click the *SampleDb* project in Solution Explorer and choose Properties. In the Project Settings tab, you'll notice that the Target Platform is currently set to SQL Server 2012. Change it to SQL Azure as shown in Figure 1-22, press **Ctrl+S** to save the properties, and then close the properties window.



Configuration: N/A Platform: N/A

Target platform: SQL Azure

☑ Enable extended Transact-SQL verification for common objects

Output types

Data-tier Application (.dacpac file):

□ Create script (.sql file)

General

Default schema: dbo

□ Include schema name in file name

☑ Validate casing on identifiers

FIGURE 1-22 Changing the target platform of a SQL Server Database Project to SQL Azure.

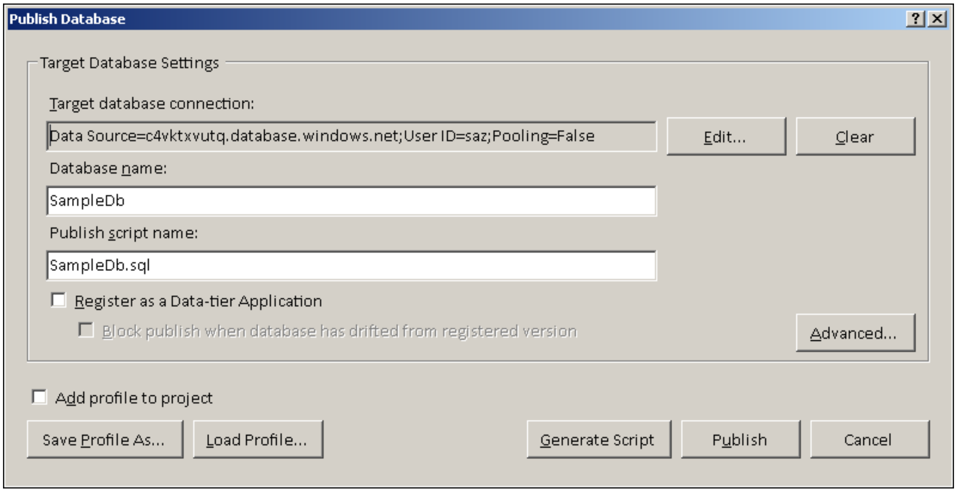
Now press **F5** to once again build the project and deploy it to LocalDB. The build fails, and the Error List pane shows the following error:

```
SQL71560: Table [dbo].[OrderHeader] does not have a clustered index.  Clustered indexes are
required for inserting data in this version of SQL Server.
```

This error informs you that the *OrderHeader* table is missing a clustered index. The astute reader might have noticed back in Listing 1-1 that the *OrderHeaderId* column in this table does not specify *PRIMARY KEY* (like the *Customer* table does on its *CustomerId* column), and so *OrderHeader* has no clustered index. This was an oversight that might not have been caught so easily because tables in on-premise editions of SQL Server do not require a clustered index. But SQL Azure databases absolutely require a clustered index on every table, so now that you're targeting the cloud specifically, the problem is brought to your attention inside the project.

This is a quick and easy fix to make using the table designer. Back in the SQL Server Database Project (in Solution Explorer), double-click the *OrderHeader.sql* table (under the dbo and Tables folders) to open the project's definition of the table in the designer. Right-click the *OrderHeaderId* column, choose Set Primary Key, save, and then close the table designer. The primary key definition results in the creation of a clustered index on the table. This resolves the issue, and you'll see the error disappear from the Error List pane immediately.

Now that you know the database is cloud-compatible, you're ready to deploy it to SQL Azure. Right-click the SQL Server Database Project in Solution Explorer and choose Publish to display the Publish Database dialog. Click Edit, enter the server and login information for your SQL Azure database, and click OK. Figure 1-23 shows the Publish Database dialog with the target connection string pointing to a SQL Azure database.



Publish Database

Target Database Settings

Target database connection: Data Source=c4vktxvutq.database.windows.net;User ID=saz;Pooling=False

Database name: SampleDb

Publish script name: SampleDb.sql

☐ Register as a Data-tier Application

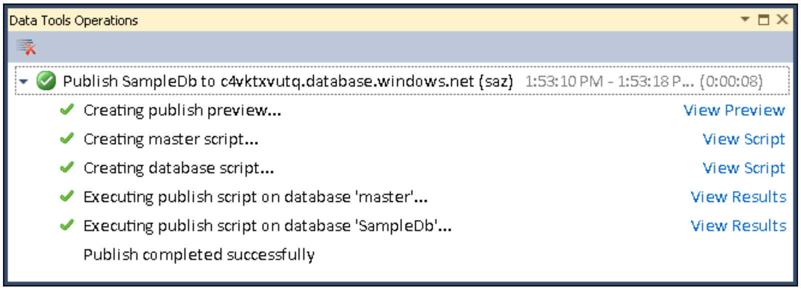
☐ Block publish when database has drifted from registered version

☐ Add profile to project

FIGURE 1-23 The Publish Database dialog set to deploy the project to a SQL Azure target instance.

As we've been noting all along, you can script the deployment without executing it by clicking Generate Script. But you're ready to deploy to SQL Azure right now. Click Publish, and Visual Studio spins up the same familiar process. It performs a schema compare between the source SQL Server Database Project and target SQL Azure instance, and then generates and executes the resulting change script on the target. As with your very first build to LocalDB, the database does not exist yet on the target, so the change script creates the whole database in the cloud from scratch. Subsequent deployments will generate incremental change scripts that specify just the actions needed to synchronize the SQL Azure database with the project.

During the deployment process, the Data Tools Operations window in Visual Studio provides a dynamic display of what's happening. Figure 1-24 shows the Data Tools Operations window after the publish process is complete.


Data Tools Operations

```
Publish SampleDb to c4vktxvutq.database.windows.net (saz)   1:53:10 PM - 1:53:18 P... (0:00:08)
    ✓ Creating publish preview...                                   View Preview
    ✓ Creating master script...                                      View Script
    ✓ Creating database script...                                    View Script
    ✓ Executing publish script on database 'master'...              View Results
    ✓ Executing publish script on database 'SampleDb'...            View Results
      Publish completed successfully
```

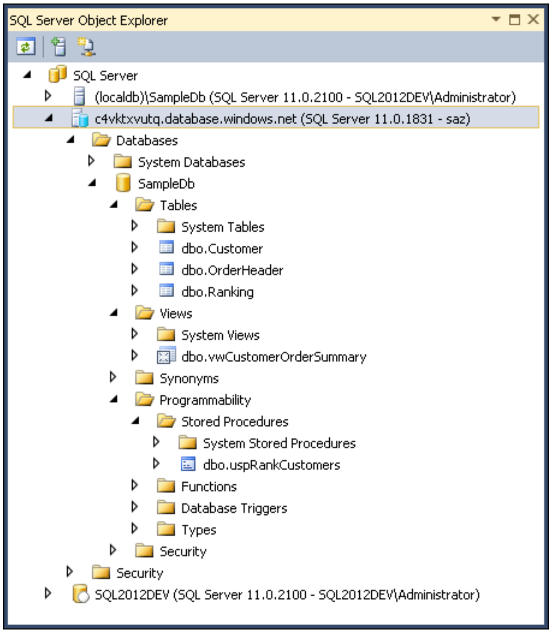
FIGURE 1-24 The Data Tools Operations pane reports all the actions taken to deploy to SQL Azure.

A really nice feature of the Data Tools Operations pane is the ability to see the scripts that were just executed inside query windows and view their execution results. Click the various links (View Preview, View Script, and View Results) to review the deployment you just ran.

After deploying, SSDT automatically adds your SQL Azure server instance to the SQL Server Object Explorer, as shown in Figure 1-25. You can drill down on SQL Azure databases in SQL Server Object Explorer and work with them using the very same development tools and techniques that we've shown throughout this chapter. It's exactly the same model-based, buffered experience you have with connected development of on-premise databases, only now it's a SQL Azure database backing the model. Thus, SQL Server Object Explorer functions as a single access point for connected development against any SQL Server database, wherever it's located.

You've used SSDT to successfully implement all the tasks to fulfill your requirements. Before concluding your work, take another snapshot. Right-click the project in Solution Explorer one last time and choose Snapshot Project. SSDT serializes the database model (based on the project's current state) into another *.dacpac* file in the Snapshots folder, which you should rename to **Version1Complete.dacpac**.

Now your project has two snapshots, *Version1Baseline.dacpac* and *Version1Complete.dacpac*, and each represents the database structure at two different points in time. The collection will grow over time as you take new snapshots during future development, and thus your project accumulates an historical account of its database structure as it changes with each new version. And because any snapshot can serve as either the source or target model of a schema compare operation, it's very easy to difference between any two points in time, or between any single point in time and either a live database (on-premise or SQL Azure) or an offline SQL Server Database Project.



```
SQL Server
  (localdb)\SampleDb (SQL Server 11.0.2100 - SQL2012DEV\Administrator)
  c4vktxvutq.database.windows.net (SQL Server 11.0.1831 - saz)
    Databases
      System Databases
      SampleDb
        Tables
          System Tables
          dbo.Customer
          dbo.OrderHeader
          dbo.Ranking
        Views
          System Views
          dbo.vwCustomerOrderSummary
        Synonyms
        Programmability
          Stored Procedures
            System Stored Procedures
            dbo.uspRankCustomers
          Functions
          Database Triggers
          Types
        Security
    Security
  SQL2012DEV (SQL Server 11.0.2100 - SQL2012DEV\Administrator)
```

FIGURE 1-25 A SQL Azure database connected in SQL Server Object Explorer.

Adopting SSDT

No tool is perfect, and SSDT is no exception. Yet even as we call out those areas where the tool is lacking, we'll still emphasize what big believers we are in this new technology, and that we greatly encourage SSDT adoption over traditional database development methods. The SSDT team has done a fantastic job with the model-based design, and there is a lot more tooling still that can be provided by leveraging the model's rich metadata, such as database diagrams and query designers that are not yet provided. There is also no spatial viewer to graphically display spatial query results, such as the one provided in SQL Server Management Studio (we cover spatial queries and the spatial viewer in Chapter 9).

Although SSDT is intimately aware of database schema, it does not provide data-oriented functionality. So it can't generate data or compare data in the database, nor does it support database unit testing. These are important features supported by the Visual Studio Database Professional edition (DbPro) that are still missing from SSDT. This means that, although SSDT is positioned to obsolesce DbPro, that won't happen until it achieves parity with key components of the DbPro feature set.

Summary

This chapter began with a high-level overview describing the many challenges developers face working with databases during the development process. Through hands-on exercises, you then saw how the new SQL Server Data Tools (SSDT) provides features that can help you tackle those challenges.

You worked through a number of scenarios using SSDT for connected development, offline development with the local database runtime (LocalDB), source control, debugging, and testing—and then deployed to a local environment as well as the cloud—all from within Visual Studio. Along the way, you learned how to use many important SSDT features, such as schema compare, refactoring, snapshot versioning, and multi-platform targeting. Although you can always learn more about the new tools just by using them, this chapter has prepared you to use SSDT as you encounter the many challenging SQL Server development tasks that lie ahead.

CHAPTER 6

XML and the Relational Database

—Leonard Lobel

Ever since it exploded on the world scene in 1998, eXtensible Markup Language (XML) has served (and continues to serve) as *the* de facto text-based standard for exchanging information between different systems and across the Internet. XML is a markup language (derived from SGML) for documents that contain semi-structured hierarchical information. In XML, data is organized as a tree of parent and child nodes, which is quite different than the way data is structured in the tables and columns of a traditional relational database. The emerging relevance of this markup format first inspired the database to support XML in Microsoft SQL Server 2000, which was capable of reading XML into tables using the *OPENXML* function, and returning query results as XML using the *FOR XML* clause. But it was SQL Server 2005 that really positioned XML as a first-class citizen in the relational database world with the native *xml* data type, and all of the rich XML support that comes along with it, such as XML Schema Definition (XSD) validation, querying with the XML Query (XQuery) and XML Path (XPath) languages, and updating with XML DML (all of which we explore in this chapter).

Why would you want to store and work with XML in the database? Database purists would insist that you should never store XML in the database because they view XML strictly as a transfer mechanism, not a storage mechanism. They would argue that you should only use XML to transport data from one database or application to another, deconstruct the XML on import and store it in relational tables, and reconstruct it on export from the relational tables back to XML for transport. On the extreme other end of the spectrum, XML proponents view the world as just a bunch of XML files and use XML technologies (such as XSD, XQuery, XPath) to store and manipulate their data, with little interest in relational technologies and Transact SQL (T-SQL).

Both camps have good arguments and valid points. A relational database has features such as primary keys, indexes, and referential integrity that make it a far superior storage and querying mechanism for raw data. Some applications, or even databases themselves, shred XML data into relational data to store it in the database and compose XML when data is retrieved. At other times, the XML data is simply persisted as (unstructured) text in the database. When Microsoft SQL Server 2000 was introduced, it offered both of these options, yet neither is necessarily the desirable solution today. Today, the rich XML support in SQL Server 2012 makes it a compelling feature to exploit in a variety of situations.

So which do you use, a “pure” relational approach or a hybrid approach where you store XML in the database and work with it there? The answer, as with so much in SQL Server, is “it depends.” When you are architecting a highly transactional application system (traditionally referred to as an online transaction processing, or OLTP, system) where many simultaneous reads and writes are performed by users, the most suitable choice is a full relational database technology that includes features such as primary keys, referential integrity, and transactions. Or, if you have a massive data warehouse and want to provide users with access to trend analysis and data mining algorithms, you will still use the traditional relational model in conjunction with the online analytical processing (OLAP) technology.

Conversely, there are certain times when you should definitely consider using XML in your database. One situation that's particularly suited to XML storage is when you are persisting objects that are being serialized and deserialized as XML in the application layer. Using the *xml* data type, as you'll learn in this chapter, provides a natural storage space for such data. It's particularly well-suited for XML-centric applications—that is, applications that work heavily with XML content storage and retrieval. XML in the database can also provide a vastly simpler solution than attribute (key/value pair) tables, when you require a flexible schema that can change without disturbing the schemas of your relational tables. And regardless of the nature or source of your XML content, you can seamlessly query against it at the database level by extending the *WHERE* clause of your ordinary relational queries with the XQuery functions that you will learn about in this chapter.

Even if you never actually store data using the *xml* data type in your underlying tables, the rich XML support in SQL Server offers powerful benefits. So conversely, you can design views, stored procedures, and table-valued functions (TVFs) that package and return complex structures (such as child entities) inside a single XML snippet as a scalar *xml* data-typed column in the query result set—while the source data is all persisted relationally in the database. For example, you can write a stored procedure that returns a single result set of orders, where each order has an *OrderDetails* column describing multiple detail rows as a single *xml* data-typed value. The stored procedure can easily manufacture the *OrderDetails* column on the fly from the related detail rows it joins on for each order. Thus you can return a set of orders with details in a single result set, rather than the more conventional approach of returning multiple result sets or making additional round-trips to the server to retrieve child entities. Similarly, you can accept hierarchical structures as an *xml* data type and shred them into rows inserted into relational tables. These are just a few of many examples where using a native *xml* data type in SQL Server can greatly simplify the processing (including storage, query, manipulation, and transport) of complex data structures.

Character Data as XML

XML, in all its dialects, is stored ultimately as string (character) data. Before the *xml* data type, XML data could only be stored in SQL Server using ordinary string data types, such as *varchar(max)* and *text*, and doing so raises several challenges. The first issue is validating the XML that is persisted (and by this we mean validating the XML against an XSD schema). SQL Server has no means of performing such a validation using ordinary strings, so the XML data can't be validated except by an outside application which can be a risky proposition (the true power of a relational database management system, or RDBMS, is applying rules at the server level).

The second issue is querying the data. Sure, you could look for data using character and pattern matching by using functions such as *CharIndex* or *PatIndex*, but these functions cannot efficiently or dependably find specific data in a structured XML document. The developer could also implement full-text search, which could also index the text data, but this solution would make things only a little better while adding the overhead of the full-text search engine. It would still be very difficult to extract data from a specific attribute in a specific child element in the XML content, and it certainly wouldn't be very efficient. You would not be able to write a query that said "Show me all data where the 'Author' attribute is set to 'Lukas Keller'."

The third issue is modifying the XML data. The developer could simply replace the entire XML contents—which is not at all efficient—or use the *UpdateText* function to do in-place changes. However, *UpdateText* requires that you know the exact locations and length of data you are going to replace, which, as we just stated, would be difficult and slow to do.

The natural evolution of persisting native XML data in the database has been realized since SQL Server 2005, with powerful T-SQL extensions that address all three of the aforementioned issues. Not only can SQL Server persist native XML data in the database, but it can index the data, query it using XPath and XQuery, and even modify it efficiently.

The *xml* Data Type

Using the *xml* data type, you can store XML in its native format, query the data within the XML, efficiently and easily modify data within the XML without having to replace the entire contents, and index the data in the XML. You can use *xml* as any of the following:

- A variable
- A parameter in a stored procedure or a user-defined function (UDF)
- A return value from a UDF
- A column in a table

There are some limitations of the *xml* data type to be aware of. Although this data type can contain and be checked for null values, unlike other native types, you cannot directly compare an instance of an *xml* data type to another instance of an *xml* data type. (You can, however, convert that instance to a *text* data type and then do a compare.) Any such equality comparisons require first casting the *xml* type to a *character* type. This limitation also means that you cannot use *ORDER BY* or *GROUP BY* with an *xml* data type. There are several other restrictions, which we will discuss in more detail later.

These might seem like fairly severe restrictions, but they don't really affect the *xml* data type when it is used appropriately. The *xml* data type also has a rich feature set that more than compensates for these limitations.

Working with the *xml* Data Type as a Variable

Let's start by writing some code that uses the *xml* data type as a variable. As with any other T-SQL variable, you simply declare it and assign data to it. Listing 6-1 shows an example that uses a generic piece of XML to represent basic order information.

LISTING 6-1 Creating XML and storing it in an *xml* variable using T-SQL.

```
DECLARE @XmlData AS xml = '
<Orders>
  <Order>
    <OrderId>5</OrderId>
    <CustomerId>60</CustomerId>
    <OrderDate>2008-10-10T14:22:27.25-05:00</OrderDate>
    <OrderAmount>25.90</OrderAmount>
  </Order>
</Orders>'

SELECT @XmlData
```

Listing 6-1 shows an *xml* variable being declared and assigned like any other native SQL Server *character* data type by using the *DECLARE* statement. The XML is then returned to the caller via a *SELECT* statement, and the results appear with the XML in a single column in a single row of data. Another benefit of having the database recognize that you are working with XML (rather than raw text that happens to be XML) is that XML results in SQL Server Developer Tools (SSDT) and SQL Server Management Studio (SSMS) are rendered as a hyperlink. Clicking the hyperlink then opens a new window displaying nicely formatted XML with color-coding and collapsible/expandable nodes.

Working with XML in Tables

Now you will define an actual column as XML in a new *AdventureWorks* database table. Execute the code shown in Listing 6-2 to create the new *OrdersXML* table.

LISTING 6-2 Creating a table to store XML in the database.

```
USE AdventureWorks2012
GO

CREATE TABLE OrdersXML(
  OrdersId int PRIMARY KEY,
  OrdersDoc xml NOT NULL DEFAULT '<Orders />')
GO
```

As we stated earlier, the *xml* data type has a few other restrictions—in this case, when it is used as a column in a table:

- It cannot be used as a primary key.
- It cannot be used as a foreign key.
- It cannot be declared with a *UNIQUE* constraint.
- It cannot be declared with the *COLLATE* keyword.

We also stated earlier that you can’t compare two instances of the *xml* data type. Primary keys, foreign keys, and unique constraints all require that you must be able to compare any included data types; therefore, XML cannot be used in any of those situations. The SQL Server *COLLATE* statement is meaningless with the *xml* data type because SQL Server does not store the XML as text; rather, it uses a distinct type of encoding particular to XML. Note however that you can designate a *DEFAULT* value, as in this case, where an empty *<Orders />* element will be assigned by default if no value is supplied for *OrdersDoc* in an *INSERT* statement.

Now get some data into the column. Listing 6-3 takes some simple static XML and inserts it into the *OrdersXML* table you just created, using the xml data type as a variable.

LISTING 6-3 Storing XML in the database.

```
DECLARE @XmlData AS xml = '
<Orders>
  <Order>
    <OrderId>5</OrderId>
    <CustomerId>60</CustomerId>
    <OrderDate>2008-10-10T14:22:27.25-05:00</OrderDate>
    <OrderAmount>25.90</OrderAmount>
  </Order>
</Orders>'

INSERT INTO OrdersXML (OrdersId, OrdersDoc) VALUES (1, @XmlData)
```

You can insert data into *xml* columns in a variety of other ways: XML Bulk Load (which we will discuss later in this chapter), loading from an XML variable (as shown here), or loading from a *SELECT* statement using the *FOR XML TYPE* feature, which we will discuss shortly. Only well-formed XML (including fragments) can be inserted—any attempt to insert malformed XML will result in an exception, as shown in this fragment where there is a case-sensitivity problem in the end tag (the word *Orders* is not capitalized, as it is in the start tag):

```
INSERT INTO OrdersXML (OrdersId, OrdersDoc) VALUES (2, '<Orders></orders>')
```

The results produce the following error from SQL Server:

```
Msg 9436, Level 16, State 1, Line 1
XML parsing: line 1, character 17, end tag does not match start tag
```

XML Schema Definitions (XSDs)

One very important feature of XML is its ability to strongly type data in an XML document. The XSD language—itself composed in XML—defines the expected format for all XML documents validated against a particular XSD. You can use XSD to create an XML schema for your data, requiring that your data conform to a set of rules that you specify. This gives XML an advantage over just about all other data transfer/data description methods and is a major contributing factor to the success of the XML standard.

Without XSD, your XML data would just be another unstructured, text-delimited format. An XSD defines what your XML data should look like, what elements are required, and what data types those elements will have. Analogous to how a table definition in SQL Server provides structure and type validation for relational data, an XML schema provides structure and type validation for the XML data.

We won't fully describe all the features of the XSD language here—that would require a book of its own. You can find the XSD specifications at the World Wide Web Consortium (W3C), at *http://www.w3.org/2001/XMLSchema*. Several popular schemas are publicly available, including one for Really Simple Syndication (RSS), Atom Publishing Protocol (APP, based on RSS), which are protocols that power weblogs, blogcasts, and other forms of binary and text syndication, as well as one for SOAP, which dictates how XML Web Services exchange information.

You can choose how to structure your XSD. Your XSD can designate required elements and set limits on what data types and ranges are allowed. It can even allow document fragments.

SQL Server Schema Collections

SQL Server lets you create your own schemas and store them in the database as database objects, and to then enforce a schema on any XML instance, including columns in tables and SQL Server variables. This gives you precise control over the XML that is going into the database and lets you strongly type your XML instance.

To get started, you can create the following simple schema and add it to the *schemas* collection in *AdventureWorks2012,* as shown in Listing 6-4.

LISTING 6-4 Creating an XML Schema Definition (XSD).

```
CREATE XML SCHEMA COLLECTION OrdersXSD AS '
  <xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
    <xsd:simpleType name="OrderAmountFloat" >
      <xsd:restriction base="xsd:float" >
        <xsd:minExclusive value="1.0" />
        <xsd:maxInclusive value="5000.0" />
     </xsd:restriction>
    </xsd:simpleType>
    <xsd:element name="Orders">
      <xsd:complexType>
        <xsd:sequence>
         <xsd:element name="Order">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="OrderId" type="xsd:int" />
                <xsd:element name="CustomerId" type="xsd:int" />
                <xsd:element name="OrderDate" type="xsd:dateTime" />
                <xsd:element name="OrderAmount" type="OrderAmountFloat" />
              </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>'
```

This schema is named *OrdersXSD*, and you can use it on any *xml* type, including variables, parameters, return values, and especially columns in tables. This schema defines elements named *OrderId*, *CustomerId*, *OrderDate*, and *OrderAmount*. The *OrderAmount* element references the *OrderAmountFloat* type, which is defined as a *float* data type whose minimum value is anything greater than (but not including) 1 and whose maximum value is 5000.

Next, create a simple table and apply the schema to the XML column by referring to the schema name in parentheses after your *xml* data type in the *CREATE TABLE* statement, as shown in Listing 6-5.

LISTING 6-5 Creating a table with an *xml* column bound to an XML Schema Definition (XSD).

```
IF EXISTS(SELECT name FROM sys.tables WHERE name = 'OrdersXML' AND type = 'U')
 DROP TABLE OrdersXML

CREATE TABLE OrdersXML(
  OrdersId int PRIMARY KEY,
  OrdersDoc xml(OrdersXSD) NOT NULL)
```

As you can see in this example, the *OrdersDoc* column is defined not as simply *xml,* but as *xml(OrdersXSD)*. The *xml* data type has an optional parameter that allows you to specify the bound schema. This same usage also applies if you want to bind a schema to another use of an *xml* data type, such as a variable or a parameter. SQL Server now allows only a strongly typed XML document in the *OrdersDoc* column. This is much better than a *CHECK* constraint (which you can still add to this column, but only with a function). An advantage of using an XML schema is that your data is validated against it and you can enforce *xml* data types (at the XML level) and make sure that only valid XML data is allowed into the particular elements. If you were using a *CHECK* constraint, for example, you would need a separate *CHECK* constraint for each validation you wanted to perform. In this example, without an XSD, several *CHECK* constraints would be needed just to enforce the minimum and maximum ages. You would need one constraint requiring the element and then another constraint to verify the allowed low end of the range and another one to verify the high end of the allowed range.

To see the schema in action, execute the code in Listing 6-6.

LISTING 6-6 Validating XML data against an XSD.

```
-- Works because all XSD validations succeed
INSERT INTO OrdersXML VALUES(5, '
  <Orders>
    <Order>
```

```
      <OrderId>5</OrderId>
      <CustomerId>60</CustomerId>
      <OrderDate>2011-10-10T14:22:27.25-05:00</OrderDate>
      <OrderAmount>25.90</OrderAmount>
    </Order>
  </Orders>')
GO

-- Won't work because 6.0 is not a valid int for CustomerId
UPDATE OrdersXML SET OrdersDoc = '
  <Orders>
    <Order>
      <OrderId>5</OrderId>
      <CustomerId>6.0</CustomerId>
      <OrderDate>2011-10-10T14:22:27.25-05:00</OrderDate>
      <OrderAmount>25.90</OrderAmount>
    </Order>
  </Orders>'
 WHERE OrdersId = 5
GO

-- Won't work because 25.9O uses an O for a 0 in the OrderAmount
UPDATE OrdersXML SET OrdersDoc = '
  <Orders>
    <Order>
      <OrderId>5</OrderId>
      <CustomerId>60</CustomerId>
      <OrderDate>2011-10-10T14:22:27.25-05:00</OrderDate>
      <OrderAmount>25.9O</OrderAmount>
    </Order>
  </Orders>'
 WHERE OrdersId = 5
GO

-- Won't work because 5225.75 is too large a value for OrderAmount
UPDATE OrdersXML SET OrdersDoc = '
  <Orders>
    <Order>
      <OrderId>5</OrderId>
      <CustomerId>60</CustomerId>
      <OrderDate>2011-10-10T14:22:27.25-05:00</OrderDate>
      <OrderAmount>5225.75</OrderAmount>
    </Order>
  </Orders>'
 WHERE OrdersId = 5
GO
```

SQL Server enforces the schema on inserts and updates, ensuring data integrity. The data provided for the *INSERT* operation at the top of Listing 6-6 conforms to the schema, so the *INSERT* works just fine. Each of the three *UPDATE* statements that follow all attempt to violate the schema with various invalid data, and SQL Server rejects them with error messages that show the offending data (and location) that's causing the problem:

```
Msg 6926, Level 16, State 1, Line 106
XML Validation: Invalid simple type value: '6.0'. Location: /*:Orders[1]/*:Order[1]/*:Customer
Id[1]
Msg 6926, Level 16, State 1, Line 119
XML Validation: Invalid simple type value: '25.9O'. Location: /*:Orders[1]/*:Order[1]/*:Order
Amount[1]
Msg 6926, Level 16, State 1, Line 132
XML Validation: Invalid simple type value: '5225.75'. Location: /*:Orders[1]/*:Order[1]/*:Order
Amount[1]
```

Lax Validation

XSD also supports *lax validation*. Say that you want to add an additional element to the XML from the preceding example, after <*OrderAmt*>, that is not part of the same schema. Schemas can use *processContents* values of *skip* and *strict* for *any* and *anyAttribute* declarations as a wildcard (if you're unfamiliar with these schema attributes and values, they're used to dictate how the XML parser should deal with XML elements not found in the schema). If *processContents* is set to *skip*, SQL Server will skip completely the validation of the additional element, even if a schema is available for it. If *processContents* is set to *strict*, SQL Server will require that it has an element or namespace defined in the current schema against which the element will be validated. Lax validation provides an additional "in-between" validation option. By setting the *processContents* attribute for this wildcard section to *lax*, you can enforce validation for any elements that have a schema associated with them but ignore any elements that are not defined in the schema.

Consider the schema you just worked with in Listing 6-4. You can modify this XSD to tolerate additional elements after *OrderAmount* that are defined in another schema, whether or not that schema is available. A schema needs to be dropped before you can re-create a modified version of it, and objects bound to the schema must be dropped before you can drop the schema. Therefore, before re-creating the schema for lax validation, you must execute the following statements:

```
DROP TABLE OrdersXML
DROP XML SCHEMA COLLECTION OrdersXSD
```

Now re-create the XSD in Listing 6-4 with one small difference—add the following additional line just after the last *xsd:element* line for *OrderAmount*:

```
<xsd:any namespace="##other" processContents="lax"/>
```

With this small change in place, arbitrary XML elements following <*OrderAmt*> will be allowed to be stored without failing validation, if the external XSD is not accessible. To see this in action, first re-create the same test table as shown in Listing 6-5. Then run the code in Listing 6-7, which inserts an order containing an additional <*Notes*> element not defined as part of the *OrdersXSD* schema.

LISTING 6-7 Using lax schema validation with XML data.

```
-- Works because all XSD validations succeed
INSERT INTO OrdersXML VALUES(6, '
  <Orders>
```

```
    <Order>
      <OrderId>6</OrderId>
      <CustomerId>60</CustomerId>
      <OrderDate>2011-10-10T14:22:27.25-05:00</OrderDate>
      <OrderAmount>25.90</OrderAmount>
      <Notes xmlns="sf">My notes for this order</Notes>
    </Order>
  </Orders>')
```

Because of the *processContents="lax"* setting in the XSD, SQL Server permits additional elements defined in another XSD (the *sf* namespace in this example, as denoted by the *xmlns* attribute). The *lax* setting in the XSD tells SQL Server to validate the *<Notes>* element in the XML using the *sf* namespace if available, but to allow the element without any validation if the *sf* namespace is not available.

Union and List Types

SQL Server also supports the union of lists with *xsd:union*, so you can combine multiple lists into one simple type. For example, in the schema shown in Listing 6-8, the *shiptypeList* accepts strings such as *FastShippers* but also allows alternative integer values.

LISTING 6-8 Using union and list types in XSD.

```
-- Cleanup previous objects
DROP TABLE OrdersXML
DROP XML SCHEMA COLLECTION OrdersXSD
GO

-- Union and List types in XSD
CREATE XML SCHEMA COLLECTION OrdersXSD AS '
  <xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
    <xsd:simpleType name="shiptypeList">
      <xsd:union>
        <xsd:simpleType>
          <xsd:list>
            <xsd:simpleType>
              <xsd:restriction base="xsd:integer">
                <xsd:enumeration value="1" />
                <xsd:enumeration value="2" />
                <xsd:enumeration value="3" />
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:list>
        </xsd:simpleType>
        <xsd:simpleType>
          <xsd:list>
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
```

```
                <xsd:enumeration value="FastShippers" />
                <xsd:enumeration value="SHL" />
                <xsd:enumeration value="PSU" />
              </xsd:restriction>
          </xsd:simpleType>
          </xsd:list>
        </xsd:simpleType>
      </xsd:union>
    </xsd:simpleType>
    <xsd:element name="Orders">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="Order">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="OrderId" type="xsd:int" />
                <xsd:element name="CustomerId" type="xsd:int" />
                <xsd:element name="OrderDate" type="xsd:dateTime" />
                <xsd:element name="OrderAmount" type="xsd:float" />
                <xsd:element name="ShipType" type="shiptypeList"/>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>'
```

If you use this XSD to validate an XML instance with either a numeric value or a string value in the enumerated list, it will validate successfully, as demonstrated by the code in Listing 6-9.

LISTING 6-9 Referencing an XSD list type in XML.

```
-- Works with 1 or FastShippers in ShipType
DECLARE @OrdersXML xml(OrdersXSD) = '
  <Orders>
    <Order>
      <OrderId>6</OrderId>
      <CustomerId>60</CustomerId>
      <OrderDate>2011-10-10T14:22:27.25-05:00</OrderDate>
      <OrderAmount>25.90</OrderAmount>
      <ShipType>1</ShipType>
    </Order>
  </Orders>'
```

This example is fairly basic, but it is useful if you have more than one way to describe something and need two lists to do so. One such possibility is metric and English units of measurement. This technique is useful when you need to restrict items and are writing them from a database.

We have touched only the surface of using XML schemas in SQL Server. These schemas can get quite complex, and further discussion is beyond the scope of this book. You can easily enforce sophisticated XML schemas in your database once you master the syntax. We believe that you should always use an XML schema with your XML data to guarantee consistency in your XML data.

XML Indexes

You can create an XML index on an XML column using almost the same syntax as for a standard SQL Server index. There are four types of XML indexes: a single *primary XML index* that must be created, and three types of optional *secondary XML indexes* that are created over the primary index. An XML index is a little different from a standard SQL index—it is a clustered index on an internal table used by SQL Server to store XML data. This table is called the *node table* and cannot be accessed by programmers.

To get started with an XML index, you must first create the primary index of all the nodes. The primary index is a clustered index (over the node table, not the base table) that associates each node of your XML column with the SQL Primary Key column. It does this by indexing one row in its internal representation (a B+ tree structure) for each node in your XML column, generating an index usually about three times as large as your XML data. For your XML data to work properly, your table must have an ordinary clustered primary key column defined. That primary key is used in a join of the XQuery results with the base table. (XQuery is discussed later on in the section "Querying XML Data Using XQuery.")

To create a primary XML index, you first create a table with a primary key and an XML column, as shown in Listing 6-10.

LISTING 6-10 Creating a primary XML index for XML storage in a table.

```
IF EXISTS(SELECT name FROM sys.tables WHERE name = 'OrdersXML' AND type = 'U')
 DROP TABLE OrdersXML
GO

CREATE TABLE OrdersXML(
  OrdersId int PRIMARY KEY,
  OrdersDoc xml NOT NULL)

CREATE PRIMARY XML INDEX ix_orders
 ON OrdersXML(OrdersDoc)
```

These statements create a new primary XML index named *ix_orders* on the *OrdersXML* table's *OrdersDoc* column. The primary XML index, *ix_orders*, now has the node table populated. To examine the node table's columns, run the T-SQL shown in Listing 6-11.

LISTING 6-11 Creating a primary XML index for XML storage in a table.

```
-- Display the columns in the node table (primary XML clustered index)
SELECT
  c.column_id, c.name, t.name AS data_type
```

```
FROM
 sys.columns AS c
 INNER JOIN sys.indexes AS i ON i.object_id = c.object_id
 INNER JOIN sys.types AS t ON t.user_type_id = c.user_type_id
WHERE
 i.name = 'ix_orders' AND i.type = 1
ORDER BY
 c.column_id
```

The results are shown in Table 6-1.

TABLE 6-1 Columns in a Typical Node Table.

column_id	name	data_type
1	*id*	*varbinary*
2	*nid*	*int*
3	*tagname*	*nvarchar*
4	*taguri*	*nvarchar*
5	*tid*	*int*
6	*value*	*sql_variant*
7	*lvalue*	*nvarchar*
8	*lvaluebin*	*varbinary*
9	*hid*	*varchar*
10	*xsinil*	*bit*
11	*xsitype*	*bit*
12	*pk1*	*int*

The three types of secondary XML indexes are *path*, *value*, and *property*. You can implement a secondary XML index only after you have created a primary XML index because they are both actually indexes over the node table. These indexes further optimize XQuery statements made against the XML data.

A path index creates an index on the *Path ID* (*hid* in Table 6-1) and *Value* columns of the primary XML index, using the *FOR PATH* keyword. This type of index is best when you have a fairly complex document type and want to speed up XQuery XPath expressions that reference a particular node in your XML data with an explicit value (as explained in the section “Understanding XQuery Expressions and XPath” later in this chapter). If you are more concerned about the values of the nodes queried with wildcards, you can create a value index using the *FOR VALUE XML* index. The *VALUE* index contains the same index columns as the *PATH* index, *Value*, and *Path ID (hid)*, but in the reverse order (as shown in Table 6-1). Using the property type index with the *PROPERTY* keyword optimizes hierarchies of elements or attributes that are name/value pairs. The *PROPERTY* index contains the primary key of the base table, *Path ID (hid)*, and *Value*, in that order. The syntax to create these indexes is shown here; you must specify that you are using the primary XML index by using the *USING XML INDEX* syntax as shown in Listing 6-12.

LISTING 6-12 Creating secondary XML indexes on *path*, *value*, and *property* data.

```
-- Create secondary structural (path) XML index
CREATE XML INDEX ix_orders_path ON OrdersXML(OrdersDoc)
 USING XML INDEX ix_orders FOR PATH

-- Create secondary value XML index
CREATE XML INDEX ix_orders_val ON OrdersXML(OrdersDoc)
 USING XML INDEX ix_orders FOR VALUE

-- Create secondary property XML index
CREATE XML INDEX ix_orders_prop ON OrdersXML(OrdersDoc)
 USING XML INDEX ix_orders FOR PROPERTY
```

Be aware of these additional restrictions regarding XML indexes:

- An XML index can contain only one XML column, so you cannot create a composite XML index (an index on more than one XML column).
- Using XML indexes requires that the primary key be clustered, and because you can have only one clustered index per table, you cannot create a clustered XML index.

With the proper XML indexing in place, you can write some very efficient queries using XQuery. Before we get to XQuery, however, let's take a look at some other XML features that will help you get XML data in and out of the database.

FOR XML Commands

SQL Server supports an enhancement to the T-SQL syntax that enables normal relational queries to output their result set as XML, using any of these four approaches:

- *FOR XML RAW*
- *FOR XML AUTO*
- *FOR XML EXPLICIT*
- *FOR XML PATH*

The first three of these options were introduced with the very first XML support in SQL Server 2000. We'll start with these options and then cover later XML enhancements added in SQL Server 2008, which includes the fourth option (*FOR XML PATH*).

FOR XML RAW

FOR XML RAW produces *attribute-based XML*. *FOR XML RAW* essentially creates a flat representation of the data in which each row returned becomes an element and the returned columns become the attributes of each element. *FOR XML RAW* also doesn't interpret joins in any special way. (Joins become relevant in *FOR XML AUTO*.) Listing 6-13 shows an example of a simple query that retrieves customer and order header data.

LISTING 6-13 Using *FOR XML RAW* to produce flat, attribute-based XML.

```
SELECT TOP 10
  Customer.CustomerID, OrderHeader.SalesOrderID, OrderHeader.OrderDate
 FROM
  Sales.Customer AS Customer
  INNER JOIN Sales.SalesOrderHeader AS OrderHeader
   ON OrderHeader.CustomerID = Customer.CustomerID
 ORDER BY
  Customer.CustomerID
 FOR XML RAW
```

Both SSDT in Visual Studio and SSMS render the query results as a hyperlink that you can click on to see the output rendered as properly formatted XML in a color-coded window that supports expanding and collapsing nodes.

```
<row CustomerID="11000" SalesOrderID="43793" OrderDate="2005-07-22T00:00:00" />
<row CustomerID="11000" SalesOrderID="51522" OrderDate="2007-07-22T00:00:00" />
<row CustomerID="11000" SalesOrderID="57418" OrderDate="2007-11-04T00:00:00" />
<row CustomerID="11001" SalesOrderID="43767" OrderDate="2005-07-18T00:00:00" />
<row CustomerID="11001" SalesOrderID="51493" OrderDate="2007-07-20T00:00:00" />
<row CustomerID="11001" SalesOrderID="72773" OrderDate="2008-06-12T00:00:00" />
<row CustomerID="11002" SalesOrderID="43736" OrderDate="2005-07-10T00:00:00" />
<row CustomerID="11002" SalesOrderID="51238" OrderDate="2007-07-04T00:00:00" />
<row CustomerID="11002" SalesOrderID="53237" OrderDate="2007-08-27T00:00:00" />
<row CustomerID="11003" SalesOrderID="43701" OrderDate="2005-07-01T00:00:00" />
```

As you can see, you get flat results in which each row returned from the query becomes a single element named *row* and all columns are output as attributes of that element. Odds are, however, that you will want more structured XML output, which leads us to *FOR XML AUTO*.

FOR XML AUTO

FOR XML AUTO also produces attribute-based XML (by default), but its output is hierarchical rather than flat—that is, it can create nested results based on the tables in the query's join clause. For example, using the same query just demonstrated, you can simply change the *FOR XML* clause to *FOR XML AUTO*, as shown in Listing 6-14.

LISTING 6-14 Using *FOR XML AUTO* to produce hierarchical, attribute-based XML.

```
SELECT TOP 10 -- limits the result rows for demo purposes
  Customer.CustomerID, OrderHeader.SalesOrderID, OrderHeader.OrderDate
 FROM
  Sales.Customer AS Customer
  INNER JOIN Sales.SalesOrderHeader AS OrderHeader
   ON OrderHeader.CustomerID = Customer.CustomerID
 ORDER BY
  Customer.CustomerID
 FOR XML AUTO
```

Execute this query, click the XML hyperlink in the results, and you will see the following output:

```
<Customer CustomerID="11000">
  <OrderHeader SalesOrderID="43793" OrderDate="2005-07-22T00:00:00" />
  <OrderHeader SalesOrderID="51522" OrderDate="2007-07-22T00:00:00" />
  <OrderHeader SalesOrderID="57418" OrderDate="2007-11-04T00:00:00" />
</Customer>
<Customer CustomerID="11001">
  <OrderHeader SalesOrderID="43767" OrderDate="2005-07-18T00:00:00" />
  <OrderHeader SalesOrderID="51493" OrderDate="2007-07-20T00:00:00" />
  <OrderHeader SalesOrderID="72773" OrderDate="2008-06-12T00:00:00" />
</Customer>
<Customer CustomerID="11002">
  <OrderHeader SalesOrderID="43736" OrderDate="2005-07-10T00:00:00" />
  <OrderHeader SalesOrderID="51238" OrderDate="2007-07-04T00:00:00" />
  <OrderHeader SalesOrderID="53237" OrderDate="2007-08-27T00:00:00" />
</Customer>
<Customer CustomerID="11003">
  <OrderHeader SalesOrderID="43701" OrderDate="2005-07-01T00:00:00" />
</Customer>
```

As you can see, the XML data has main elements named *Customer* (based on the alias assigned in the query) and child elements named *OrderHeader* (again from the alias). Note that *FOR XML AUTO* determines the element nesting order based on the order of the columns in the *SELECT* clause. You can rewrite the *SELECT* clause so that an *OrderHeader* column comes before a *Customer* column, by changing the order of the columns returned by the query, as shown in Listing 6-15.

LISTING 6-15 Changing the hierarchy returned by *FOR XML AUTO* by reordering query columns.

```
SELECT TOP 10
  OrderHeader.SalesOrderID, OrderHeader.OrderDate, Customer.CustomerID
 FROM
  Sales.Customer AS Customer
  INNER JOIN Sales.SalesOrderHeader AS OrderHeader
   ON OrderHeader.CustomerID = Customer.CustomerID
 ORDER BY
  Customer.CustomerID
 FOR XML AUTO
```

The output (as viewed in the XML viewer) now looks like this:

```
<OrderHeader SalesOrderID="43793" OrderDate="2005-07-22T00:00:00">
  <Customer CustomerID="11000" />
</OrderHeader>
<OrderHeader SalesOrderID="51522" OrderDate="2007-07-22T00:00:00">
  <Customer CustomerID="11000" />
</OrderHeader>
<OrderHeader SalesOrderID="57418" OrderDate="2007-11-04T00:00:00">
  <Customer CustomerID="11000" />
</OrderHeader>
<OrderHeader SalesOrderID="43767" OrderDate="2005-07-18T00:00:00">
  <Customer CustomerID="11001" />
</OrderHeader>
<OrderHeader SalesOrderID="51493" OrderDate="2007-07-20T00:00:00">
  <Customer CustomerID="11001" />
</OrderHeader>
<OrderHeader SalesOrderID="72773" OrderDate="2008-06-12T00:00:00">
  <Customer CustomerID="11001" />
</OrderHeader>
<OrderHeader SalesOrderID="43736" OrderDate="2005-07-10T00:00:00">
  <Customer CustomerID="11002" />
</OrderHeader>
<OrderHeader SalesOrderID="51238" OrderDate="2007-07-04T00:00:00">
  <Customer CustomerID="11002" />
</OrderHeader>
<OrderHeader SalesOrderID="53237" OrderDate="2007-08-27T00:00:00">
  <Customer CustomerID="11002" />
</OrderHeader>
<OrderHeader SalesOrderID="43701" OrderDate="2005-07-01T00:00:00">
  <Customer CustomerID="11003" />
</OrderHeader>
```

These results are probably not what you wanted. To keep the XML hierarchy matching the table hierarchy, you must list at least one column from the parent table before any column from a child table. If there are three levels of tables, at least one other column from the child table must come before any from the grandchild table, and so on.

FOR XML EXPLICIT

FOR XML EXPLICIT is the most complex but also the most powerful and flexible of the three original *FOR XML* options. We cover it now for completeness, but recommend using the simpler *FOR XML PATH* feature added in SQL Server 2008 (covered shortly). As you'll see, *FOR XML PATH* can shape query results into virtually any desired XML with much less effort than using *FOR XML EXPLICIT*.

With *FOR XML EXPLICIT*, SQL Server constructs XML based on a *UNION* query of the various levels of output elements. So, if again you have the *Customer* and *SalesOrderHeader* tables and you want to produce XML output, you must have two *SELECT* statements with a *UNION*. If you add the *SalesOrderDetail* table, you must add another *UNION* statement and *SELECT* statement.

As we said, *FOR XML EXPLICIT* is more complex than its predecessors. For starters, you are responsible for defining two additional columns that establish the hierarchical relationship of the

XML: a *Tag* column that acts as a row's identifier and a *Parent* column that links child records to the parent record's *Tag* value (similar to *EmployeeID* and *ManagerID*). You must also alias all columns to indicate the element, *Tag*, and display name for the XML output, as shown in Listing 6-16. Keep in mind that only the first *SELECT* statement must follow these rules; any aliases in subsequent *SELECT* statements in a *UNION* query are ignored.

LISTING 6-16 Shaping hierarchical XML using *FOR XML EXPLICIT*.

```
SELECT
  1 AS Tag, -- Tag this resultset as level 1
  NULL AS Parent,  -- Level 1 has no parent
  CustomerID AS [Customer!1!CustomerID], -- level 1 value
  NULL AS [SalesOrder!2!SalesOrderID],  -- level 2 value
  NULL AS [SalesOrder!2!OrderDate]  -- level 2 value
 FROM Sales.Customer AS Customer
 WHERE Customer.CustomerID IN(11077, 11078)
 UNION ALL
 SELECT
  2, -- Tag this resultset as level 2
  1, -- Link to parent at level 1
  Customer.CustomerID,
  OrderHeader.SalesOrderID,
  OrderHeader.OrderDate
 FROM Sales.Customer AS Customer
  INNER JOIN Sales.SalesOrderHeader AS OrderHeader
          ON OrderHeader.CustomerID = Customer.CustomerID
 WHERE Customer.CustomerID IN(11077, 11078)
 ORDER BY
  [Customer!1!CustomerID], [SalesOrder!2!SalesOrderID]
 FOR XML EXPLICIT
```

Execute this query and click the XML hyperlink to see the following output:

```
<Customer CustomerID="11077">
  <SalesOrder SalesOrderID="44407" OrderDate="2005-10-16T00:00:00" />
  <SalesOrder SalesOrderID="51651" OrderDate="2007-07-29T00:00:00" />
  <SalesOrder SalesOrderID="60042" OrderDate="2007-12-14T00:00:00" />
</Customer>
<Customer CustomerID="11078">
  <SalesOrder SalesOrderID="52789" OrderDate="2007-08-19T00:00:00" />
  <SalesOrder SalesOrderID="53993" OrderDate="2007-09-08T00:00:00" />
  <SalesOrder SalesOrderID="54214" OrderDate="2007-09-12T00:00:00" />
  <SalesOrder SalesOrderID="54268" OrderDate="2007-09-13T00:00:00" />
  <SalesOrder SalesOrderID="56449" OrderDate="2007-10-21T00:00:00" />
  <SalesOrder SalesOrderID="57281" OrderDate="2007-11-02T00:00:00" />
  <SalesOrder SalesOrderID="57969" OrderDate="2007-11-15T00:00:00" />
  <SalesOrder SalesOrderID="58429" OrderDate="2007-11-23T00:00:00" />
  <SalesOrder SalesOrderID="58490" OrderDate="2007-11-24T00:00:00" />
  <SalesOrder SalesOrderID="61443" OrderDate="2008-01-04T00:00:00" />
  <SalesOrder SalesOrderID="62245" OrderDate="2008-01-17T00:00:00" />
  <SalesOrder SalesOrderID="62413" OrderDate="2008-01-20T00:00:00" />
  <SalesOrder SalesOrderID="67668" OrderDate="2008-04-05T00:00:00" />
  <SalesOrder SalesOrderID="68285" OrderDate="2008-04-15T00:00:00" />
  <SalesOrder SalesOrderID="68288" OrderDate="2008-04-15T00:00:00" />
  <SalesOrder SalesOrderID="73869" OrderDate="2008-06-27T00:00:00" />
  <SalesOrder SalesOrderID="75084" OrderDate="2008-07-31T00:00:00" />
</Customer>
```

This result resembles the output generated by the *FOR XML AUTO* sample in Listing 6-14. So what is gained by composing a more complex query with *FOR XML EXPLICIT?* Well, *FOR XML EXPLICIT* allows for some alternative outputs that are not achievable using *FOR XML AUTO*. For example, you can specify that certain values be composed as elements instead of attributes by appending *!ELEMENT* to the end of the aliased column, as shown in Listing 6-17.

LISTING 6-17 Using *!ELEMENT* to customize the hierarchical XML generated by *FOR XML EXPLICIT*.

```
SELECT
  1 AS Tag, -- Tag this resultset as level 1
  NULL AS Parent,  -- Level 1 has no parent
  CustomerID AS [Customer!1!CustomerID], -- level 1 value
  NULL AS [SalesOrder!2!SalesOrderID],  -- level 2 value
  NULL AS [SalesOrder!2!OrderDate!ELEMENT] -- level 2 value rendered as an
element
 FROM Sales.Customer AS Customer
 WHERE Customer.CustomerID IN(11077, 11078)
 UNION ALL
 SELECT
  2, -- Tag this resultset as level 2
  1, -- Link to parent at level 1
  Customer.CustomerID,
  OrderHeader.SalesOrderID,
  OrderHeader.OrderDate
 FROM Sales.Customer AS Customer
  INNER JOIN Sales.SalesOrderHeader AS OrderHeader
         ON OrderHeader.CustomerID = Customer.CustomerID
 WHERE Customer.CustomerID IN(11077, 11078)
 ORDER BY
  [Customer!1!CustomerID], [SalesOrder!2!SalesOrderID]
 FOR XML EXPLICIT
```

Only one minor change was made (the *OrderDate* column alias has *!ELEMENT* appended to the end of it). Aliasing a column with *!ELEMENT* in a *FOR XML EXPLICIT* query results in that column being rendered as an element instead of an attribute, as shown here:

```
<Customer CustomerID="11077">
  <SalesOrder SalesOrderID="44407">
    <OrderDate>2005-10-16T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="51651">
    <OrderDate>2007-07-29T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="60042">
    <OrderDate>2007-12-14T00:00:00</OrderDate>
  </SalesOrder>
```

```
</Customer>
<Customer CustomerID="11078">
  <SalesOrder SalesOrderID="52789">
    <OrderDate>2007-08-19T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="53993">
    <OrderDate>2007-09-08T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="54214">
    <OrderDate>2007-09-12T00:00:00</OrderDate>
  </SalesOrder>
    :
```

Notice that the *OrderDate* is now being rendered as a child element of the *SalesOrder* element. Thus, *FOR XML EXPLICIT* mode enables greater customization, but it also requires creating complex queries to achieve custom results. For example, to add a few more fields from *OrderHeader* and to add some additional fields from *OrderDetail* (a third hierarchical table), you would have to write the query as shown in Listing 6-18.

LISTING 6-18 Using *FOR XML EXPLICIT* to produce three-level hierarchical XML order data.

```
SELECT
  1 AS Tag,
  NULL AS Parent,
  CustomerID AS [Customer!1!CustomerID],
  NULL AS [SalesOrder!2!SalesOrderID],
  NULL AS [SalesOrder!2!TotalDue],
  NULL AS [SalesOrder!2!OrderDate!ELEMENT],
  NULL AS [SalesOrder!2!ShipDate!ELEMENT],
  NULL AS [SalesDetail!3!ProductID],
  NULL AS [SalesDetail!3!OrderQty],
  NULL AS [SalesDetail!3!LineTotal]
 FROM Sales.Customer AS Customer
 WHERE Customer.CustomerID IN(11077, 11078)
 UNION ALL
 SELECT
  2,
  1,
  Customer.CustomerID,
  OrderHeader.SalesOrderID,
  OrderHeader.TotalDue,
  OrderHeader.OrderDate,
  OrderHeader.ShipDate,
  NULL,
  NULL,
  NULL
 FROM Sales.Customer AS Customer
  INNER JOIN Sales.SalesOrderHeader AS OrderHeader
   ON OrderHeader.CustomerID = Customer.CustomerID
 WHERE Customer.CustomerID IN(11077, 11078)
 UNION ALL
SELECT
  3,
  2,
  Customer.CustomerID,
  OrderHeader.SalesOrderID,
  OrderHeader.TotalDue,
  OrderHeader.OrderDate,
  OrderHeader.ShipDate,
  OrderDetail.ProductID,
  OrderDetail.OrderQty,
  OrderDetail.LineTotal
 FROM Sales.Customer AS Customer
  INNER JOIN Sales.SalesOrderHeader AS OrderHeader
   ON OrderHeader.CustomerID = Customer.CustomerID
  INNER JOIN Sales.SalesOrderDetail AS OrderDetail
   ON OrderDetail.SalesOrderID = OrderHeader.SalesOrderID
 WHERE Customer.CustomerID IN(11077, 11078)
 ORDER BY [Customer!1!CustomerID], [SalesOrder!2!SalesOrderID]
 FOR XML EXPLICIT
```

This query produces the following XML:

```
<Customer CustomerID="11077">
  <SalesOrder SalesOrderID="44407" TotalDue="3729.3640">
    <OrderDate>2005-10-16T00:00:00</OrderDate>
    <ShipDate>2005-10-23T00:00:00</ShipDate>
    <SalesDetail ProductID="778" OrderQty="1" LineTotal="3374.990000" />
    <SalesDetail ProductID="781" OrderQty="1" LineTotal="2319.990000" />
    <SalesDetail ProductID="880" OrderQty="1" LineTotal="54.990000" />
  </SalesOrder>
  <SalesOrder SalesOrderID="51651" TotalDue="2624.3529">
    <OrderDate>2007-07-29T00:00:00</OrderDate>
    <ShipDate>2007-08-05T00:00:00</ShipDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="60042" TotalDue="2673.0613">
    <OrderDate>2007-12-14T00:00:00</OrderDate>
    <ShipDate>2007-12-21T00:00:00</ShipDate>
    <SalesDetail ProductID="969" OrderQty="1" LineTotal="2384.070000" />
    <SalesDetail ProductID="707" OrderQty="1" LineTotal="34.990000" />
  </SalesOrder>
</Customer>
<Customer CustomerID="11078">
  <SalesOrder SalesOrderID="52789" TotalDue="71.2394">
    <OrderDate>2007-08-19T00:00:00</OrderDate>
    <ShipDate>2007-08-26T00:00:00</ShipDate>
    <SalesDetail ProductID="923" OrderQty="1" LineTotal="4.990000" />
    <SalesDetail ProductID="707" OrderQty="1" LineTotal="34.990000" />
    <SalesDetail ProductID="860" OrderQty="1" LineTotal="24.490000" />
    <SalesDetail ProductID="922" OrderQty="1" LineTotal="3.990000" />
    <SalesDetail ProductID="877" OrderQty="1" LineTotal="7.950000" />
  </SalesOrder>
    :
```

As you can see, the code has become quite complex, and will become even more complex as you add additional data to the output. Although this query is perfectly valid, the same result can be achieved with far less effort using the *FOR XML PATH* statement.

Additional *FOR XML* Features

Just about all of the original XML support first introduced in SQL Server 2000 XML support revolves around *FOR XML*, a feature that is still very much underused by developers. Since then, SQL Server has enhanced *FOR XML* in the following ways:

- Using the *TYPE* option, *FOR XML* can output to an *xml* data type (as opposed to streamed results) from a *SELECT* statement, which in turn allows you to nest the results of *SELECT…FOR XML* into another *SELECT* statement.
- The *FOR XML PATH* option allows you to more easily shape data and produce element-based XML than the *FOR XML EXPLICIT* option that we just covered.
- You can explicitly specify a *ROOT* element for your output.
- You can produce element-based XML with *FOR XML AUTO*.
- *FOR XML* can produce XML with an embedded, inferred XSD schema.

The *TYPE* Option

As of SQL Server 2005, *xml* is an intrinsic data type of SQL Server. Thus, you can cast the XML output from a *FOR XML* query directly into an *xml* data type instance, as opposed to streaming XML results directly or immediately to the client. You accomplish this by using the *TYPE* keyword after your *FOR XML* statement, as shown in Listing 6-19.

LISTING 6-19 Using the *TYPE* option with *FOR XML AUTO* to cast a subquery result set as an *xml* data type.

```
SELECT
  CustomerID,
  (SELECT SalesOrderID, TotalDue, OrderDate, ShipDate
   FROM Sales.SalesOrderHeader AS OrderHeader
   WHERE CustomerID = Customer.CustomerID
   FOR XML AUTO, TYPE) AS OrderHeaders
 FROM
  Sales.Customer AS Customer
 WHERE
  CustomerID IN (11000, 11001)
```

This query returns two columns. The first is the integer *CustomerID* and the second is an *OrderHeaders* column of type *xml*. The second column is constructed by a subquery that generates

XML using *FOR XML AUTO*, and the *TYPE* option casts the generated XML from the subquery into an *xml* data type that gets returned as the *OrderHeaders* column of the main query.

FOR XML PATH

As we already mentioned, *FOR XML PATH* gives you fine control over the generated XML much like *FOR XML EXPLICIT* does, but is much simpler to use. With *FOR XML PATH*, you simply assign column aliases with XPath expressions that shape your XML output, as shown in Listing 6-20.

LISTING 6-20 Using *FOR XML PATH* to shape XML output with XPath-based column aliases.

```
SELECT
  BusinessEntityID AS [@BusinessEntityID],
  FirstName AS [ContactName/First],
  LastName AS [ContactName/Last],
  EmailAddress AS [ContactEmailAddress/EmailAddress1]
 FROM
  HumanResources.vEmployee
 FOR XML PATH('Contact')
```

The output looks like this:

```
<row BusinessEntityID="263">
  <ContactName>
    <First>Jean</First>
    <Last>Trenary</Last>
  </ContactName>
  <ContactEmailAddress>
    <EmailAddress1>jean0@adventure-works.com</EmailAddress1>
  </ContactEmailAddress>
</row>
<row BusinessEntityID="78">
  <ContactName>
    <First>Reuben</First>
    <Last>D'sa</Last>
  </ContactName>
  <ContactEmailAddress>
    <EmailAddress1>reuben0@adventure-works.com</EmailAddress1>
  </ContactEmailAddress>
</row>
 :
```

Notice that the *BusinessEntityID* column is rendered as an attribute. This is because it was aliased as *@BusinessEntityID*, and the @-symbol in XPath means “attribute.” Also notice that the *FirstName* and *LastName* columns are rendered as *First* and *Last* elements nested within a *ContactName* element. This again is due to the XPath-based syntax of the column aliases, *ContactName/First* and *ContactName/Last*.

Now let’s revisit the three-level hierarchical example we recently demonstrated with *FOR XML EXPLICIT* in Listing 6-18. Using the *TYPE* option in conjunction with *FOR XML PATH*, you can reproduce that awful and complex query with a much simpler version, as shown in Listing 6-21.

LISTING 6-21 Using *FOR XML PATH* to shape XML output for a three-level hierarchy.

```
SELECT
  CustomerID AS [@CustomerID],
  (SELECT
    SalesOrderID AS [@SalesOrderID],
    TotalDue AS [@TotalDue],
    OrderDate,
    ShipDate,
    (SELECT
      ProductID AS [@ProductID],
      OrderQty AS [@OrderQty],
      LineTotal AS [@LineTotal]
     FROM Sales.SalesOrderDetail
     WHERE SalesOrderID = OrderHeader.SalesOrderID
     FOR XML PATH('OrderDetail'), TYPE)
   FROM Sales.SalesOrderHeader AS OrderHeader
   WHERE CustomerID = Customer.CustomerID
   FOR XML PATH('OrderHeader'), TYPE)
 FROM Sales.Customer AS Customer
  INNER JOIN Person.Person AS Contact
   ON Contact.BusinessEntityID = Customer.PersonID
 WHERE CustomerID BETWEEN 11000 AND 11999
 FOR XML PATH ('Customer')
```

Isn’t that much better than the contorted *UNION*-based approach taken by *FOR XML EXPLICIT* in Listing 6-18? In this simpler version that produces the same result, subqueries are used with the *XML PATH* statement in conjunction with *TYPE* to produce element-based XML nested inside a much larger *FOR XML PATH* statement. This returns each separate *Order* for the customer as a new child node of the *CustomerID* node. And again, XPath syntax is used in the column aliases to define element and attribute structure in the generated XML. Here are the results of the query:

```
<Customer CustomerID="11480">
  <OrderHeader SalesOrderID="51053" TotalDue="2288.9187">
    <OrderDate>2007-06-28T00:00:00</OrderDate>
    <ShipDate>2007-07-05T00:00:00</ShipDate>
    <OrderDetail ProductID="779" OrderQty="1" LineTotal="2071.419600" />
  </OrderHeader>
  <OrderHeader SalesOrderID="52329" TotalDue="2552.5169">
    <OrderDate>2007-08-10T00:00:00</OrderDate>
    <ShipDate>2007-08-17T00:00:00</ShipDate>
    <OrderDetail ProductID="782" OrderQty="1" LineTotal="2294.990000" />
    <OrderDetail ProductID="870" OrderQty="1" LineTotal="4.990000" />
    <OrderDetail ProductID="871" OrderQty="1" LineTotal="9.990000" />
  </OrderHeader>
  <OrderHeader SalesOrderID="62813" TotalDue="612.1369">
    <OrderDate>2008-01-26T00:00:00</OrderDate>
    <ShipDate>2008-02-02T00:00:00</ShipDate>
    <OrderDetail ProductID="999" OrderQty="1" LineTotal="539.990000" />
    <OrderDetail ProductID="872" OrderQty="1" LineTotal="8.990000" />
    <OrderDetail ProductID="870" OrderQty="1" LineTotal="4.990000" />
  </OrderHeader>
</Customer>
<Customer CustomerID="11197">
  <OrderHeader SalesOrderID="57340" TotalDue="46.7194">
    :
```

If you are familiar and comfortable with XPath, you will appreciate some additional *XML PATH* features. You can use the following XPath node functions to further control the shape of your XML output:

- data
- comment
- node
- text
- processing-instruction

The following example uses the *data* and *comment* methods of XPath. The *data* method takes the results of the underlying query and places them all inside one element. The *comment* method takes data and transforms it into an XML comment, as demonstrated in Listing 6-22.

LISTING 6-22 Using *FOR XML PATH* with the *comment* and *data* XPath methods.

```
SELECT
  Customer.BusinessEntityID AS [@CustomerID],
  Customer.FirstName + ' ' + Customer.LastName AS [comment()],
  (SELECT
    SalesOrderID AS [@SalesOrderID],
    TotalDue AS [@TotalDue],
    OrderDate,
    ShipDate,
    (SELECT ProductID AS [data()]
     FROM Sales.SalesOrderDetail
     WHERE SalesOrderID = OrderHeader.SalesOrderID
     FOR XML PATH('')) AS [ProductIDs]
   FROM Sales.SalesOrderHeader AS OrderHeader
   WHERE CustomerID = Customer.BusinessEntityID
   FOR XML PATH('OrderHeader'), TYPE)
 FROM Sales.vIndividualCustomer AS Customer
 WHERE BusinessEntityID IN (11000, 11001)
 FOR XML PATH ('Customer')
```

As you can see from the results, the concatenated contact name becomes an XML comment, and the subquery of *Product IDs* is transformed into one element:

```
<Customer CustomerID="11000">
  <!--Mary Young-->
  <OrderHeader SalesOrderID="43793" TotalDue="3756.9890">
```

```
    <OrderDate>2005-07-22T00:00:00</OrderDate>
    <ShipDate>2005-07-29T00:00:00</ShipDate>
    <ProductIDs>771</ProductIDs>
  </OrderHeader>
  <OrderHeader SalesOrderID="51522" TotalDue="2587.8769">
    <OrderDate>2007-07-22T00:00:00</OrderDate>
    <ShipDate>2007-07-29T00:00:00</ShipDate>
    <ProductIDs>779 878</ProductIDs>
  </OrderHeader>
  <OrderHeader SalesOrderID="57418" TotalDue="2770.2682">
    <OrderDate>2007-11-04T00:00:00</OrderDate>
    <ShipDate>2007-11-11T00:00:00</ShipDate>
    <ProductIDs>966 934 923 707 881</ProductIDs>
  </OrderHeader>
</Customer>
<Customer CustomerID="11001">
  <!--Amber Young-->
  <OrderHeader SalesOrderID="43767" TotalDue="3729.3640">
    <OrderDate>2005-07-18T00:00:00</OrderDate>
       :
```

Emitting a *ROOT* Element

Technically, an XML document must be contained inside of a single root element. You've seen many applied uses of *FOR XML* that generate all types of XML, but without a root element, the generated XML can only represent a portion of an XML document. The *ROOT* option allows you to add a main, or root, element to your *FOR XML* output so that the query results can be consumed as a complete XML document. You can combine *ROOT* with other *FOR XML* keywords. In Listing 6-23, *ROOT* is used with *FOR XML AUTO* to wrap a single *Orders* root element around the results of the query.

LISTING 6-23 Using *FOR XML* with *ROOT* to generate a root element.

```
SELECT
  Customer.CustomerID,
  OrderDetail.SalesOrderID,
  OrderDetail.OrderDate
 FROM
  Sales.Customer AS Customer
   INNER JOIN Sales.SalesOrderHeader  AS OrderDetail
    ON OrderDetail.CustomerID = Customer.CustomerID
 WHERE
  Customer.CustomerID IN (11000, 11001)
 ORDER BY
  Customer.CustomerID
 FOR XML AUTO, ROOT('Orders')
```

The output looks like this:

```
<Orders>
  <Customer CustomerID="11000">
    <OrderDetail SalesOrderID="43793" OrderDate="2005-07-22T00:00:00" />
    <OrderDetail SalesOrderID="51522" OrderDate="2007-07-22T00:00:00" />
    <OrderDetail SalesOrderID="57418" OrderDate="2007-11-04T00:00:00" />
  </Customer>
  <Customer CustomerID="11001">
    <OrderDetail SalesOrderID="43767" OrderDate="2005-07-18T00:00:00" />
    <OrderDetail SalesOrderID="51493" OrderDate="2007-07-20T00:00:00" />
    <OrderDetail SalesOrderID="72773" OrderDate="2008-06-12T00:00:00" />
  </Customer>
</Orders>
```

The code output here is the same as any *FOR XML AUTO* output for this query, except that the *XML ROOT* we specified with the *ROOT* keyword now surrounds the data. In this example, we used *ROOT ('Orders')*, so our output is surrounded with an *<Orders>* XML element.

Producing an Inline XSD Schema

As you've seen, schemas provide an enforceable structure for your XML data. When you export data using the *FOR XML* syntax, you might want to include an inline XML schema for the recipient so that the recipient can enforce the rules on their end as well. When you use the *RAW* and *AUTO* modes, you can produce an inline XSD schema as part of the output by using the *XMLSCHEMA* keyword, as shown in Listing 6-24.

LISTING 6-24 Using *FOR XML* with *XMLSCHEMA* to generate an inline XSD schema with the query results.

```
SELECT
  Customer.CustomerID,
  OrderDetail.SalesOrderID,
  OrderDetail.OrderDate
 FROM
  Sales.Customer AS Customer
   INNER JOIN Sales.SalesOrderHeader  AS OrderDetail
    ON OrderDetail.CustomerID = Customer.CustomerID
 WHERE
  Customer.CustomerID IN (11000, 11001)
 ORDER BY
  Customer.CustomerID
 FOR XML AUTO, ROOT('Orders'), XMLSCHEMA
```

The output looks like this:

```
<Orders>
  <xsd:schema targetNamespace="urn:schemas-microsoft-com:sql:SqlRowSet4"
xmlns:schema="urn:schemas-microsoft-com:sql:SqlRowSet4" xmlns:xsd="http://www.w3.org/2001/
XMLSchema" xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
elementFormDefault="qualified">
    <xsd:import namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
schemaLocation="http://schemas.microsoft.com/sqlserver/2004/sqltypes/sqltypes.xsd" />
    <xsd:element name="Customer">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="schema:OrderDetail" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="CustomerID" type="sqltypes:int" use="required" />
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="OrderDetail">
      <xsd:complexType>
        <xsd:attribute name="SalesOrderID" type="sqltypes:int" use="required" />
        <xsd:attribute name="OrderDate" type="sqltypes:datetime" use="required" />
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
  <Customer xmlns="urn:schemas-microsoft-com:sql:SqlRowSet4" CustomerID="11000">
    <OrderDetail SalesOrderID="43793" OrderDate="2005-07-22T00:00:00" />
    <OrderDetail SalesOrderID="51522" OrderDate="2007-07-22T00:00:00" />
    <OrderDetail SalesOrderID="57418" OrderDate="2007-11-04T00:00:00" />
  </Customer>
  <Customer xmlns="urn:schemas-microsoft-com:sql:SqlRowSet4" CustomerID="11001">
    <OrderDetail SalesOrderID="43767" OrderDate="2005-07-18T00:00:00" />
    <OrderDetail SalesOrderID="51493" OrderDate="2007-07-20T00:00:00" />
    <OrderDetail SalesOrderID="72773" OrderDate="2008-06-12T00:00:00" />
  </Customer>
</Orders>
```

SQL Server infers the schema based on the underlying data types of the result set. For example, the *SalesOrderID* field is set to an *int* and is a required field (as per the inline schema based on the properties of the field in the underlying SQL table).

Producing Element-Based XML

Element-based XML is more verbose than attribute-based XML but is usually easier to view and work with. Initially, in SQL Server 2000, *FOR XML RAW* and *FOR XML AUTO* could only generate attribute-based XML (as shown in Listings 6-13 and 6-14). As we've demonstrated in Listings 6-17 and 6-20, you can customize the generated XML and produce element-based XML using *FOR XML EXPLICIT* and (later) *FOR XML PATH*.

Both *FOR XML RAW* and *FOR XML AUTO* were later enhanced to support the *ELEMENTS* keyword, enabling them to alternatively produce element-based XML rather than attribute-based XML. When all you need is element-based XML, and you require no other customization over the shape of generated XML, you will find it much easier to use *FOR XML RAW/AUTO* with *ELEMENT* rather than *FOR XML EXPLICIT* (and even *FOR XML PATH*). Listing 6-25 demonstrates this.

LISTING 6-25 Using *FOR XML AUTO* with *ELEMENTS* to produce element-based hierarchical XML.

```
SELECT
  Customer.CustomerID,
  OrderDetail.SalesOrderID,
  OrderDetail.OrderDate
 FROM
  Sales.Customer AS Customer
  INNER JOIN Sales.SalesOrderHeader AS OrderDetail
   ON OrderDetail.CustomerID = Customer.CustomerID
WHERE
 Customer.CustomerID IN (11000, 11001)
ORDER BY
 Customer.CustomerID
FOR XML AUTO, ROOT('Orders'), ELEMENTS
```

Here are the query results:

```
<Orders>
  <Customer>
    <CustomerID>11000</CustomerID>
    <OrderDetail>
      <SalesOrderID>43793</SalesOrderID>
      <OrderDate>2005-07-22T00:00:00</OrderDate>
    </OrderDetail>
    <OrderDetail>
      <SalesOrderID>51522</SalesOrderID>
      <OrderDate>2007-07-22T00:00:00</OrderDate>
    </OrderDetail>
    <OrderDetail>
      <SalesOrderID>57418</SalesOrderID>
      <OrderDate>2007-11-04T00:00:00</OrderDate>
    </OrderDetail>
  </Customer>
  <Customer>
    <CustomerID>11001</CustomerID>
    <OrderDetail>
      <SalesOrderID>43767</SalesOrderID>
      <OrderDate>2005-07-18T00:00:00</OrderDate>
    </OrderDetail>
    <OrderDetail>
      <SalesOrderID>51493</SalesOrderID>
      <OrderDate>2007-07-20T00:00:00</OrderDate>
    </OrderDetail>
    <OrderDetail>
      <SalesOrderID>72773</SalesOrderID>
      <OrderDate>2008-06-12T00:00:00</OrderDate>
    </OrderDetail>
  </Customer>
</Orders>
```

You can see that each column of the query becomes a nested element in the resulting XML, as opposed to an attribute of one single element. The *ELEMENTS* keyword used in conjunction with *FOR XML RAW* or *FOR XML AUTO* converts each column from your result set to an individual XML element. *FOR XML AUTO* also converts each row from a joined table to a new XML element, as just demonstrated.

Shredding XML Using *OPENXML*

Up to this point, you have been using *FOR XML* to compose XML from rows of data, but what if you already have XML data and want to shred it back into relational data? SQL Server 2000 introduced a feature called *OPENXML* for this purpose. The *OPENXML* system function is designed for this purpose, and allows an XML document file to be shredded into T-SQL rows as we'll explain next. Since the introduction of the native *xml* data type in SQL Server 2005, XQuery (covered in the next section) offers even more choices for extracting data from XML input.

To shred data XML into relational rows using *OPENXML*, you first create a handle to the XML document using the system stored procedure *sp_xml_preparedocument*. This system-stored procedure takes an XML document and creates a representation that you can reference using a special handle, which it returns via an *OUTPUT* parameter. *OPENXML* uses this handle along with a specified path and behaves like a database view to the XML data, so you simply choose *SELECT* from the *OPENXML* function just as you would *SELECT* from a table or a view. The code in Listing 6-26 shows an example of *OPENXML* in action.

LISTING 6-26 Using *FOR XML AUTO* with *ELEMENTS* to produce element-based hierarchical XML.

```
DECLARE @handle int
DECLARE @OrdersXML varchar(max)
SET @OrdersXML = '
<Orders>
  <Customer CustomerID="HERBC" ContactName="Charlie Herb">
     <Order CustomerID="HERBC" EmployeeID="5" OrderDate="2011-11-04">
        <OrderDetail OrderID="10248" ProductID="16" Quantity="12"/>
        <OrderDetail OrderID="10248" ProductID="32" Quantity="10"/>
     </Order>
     <Order CustomerID="HERBC" EmployeeID="2" OrderDate="2011-11-16">
        <OrderDetail OrderID="10283" ProductID="99" Quantity="3"/>
     </Order>
  </Customer>
  <Customer CustomerID="HINKM" ContactName="Matt Hink">
     <Order CustomerID="HINKM" EmployeeID="3" OrderDate="2011-11-23">
        <OrderDetail OrderID="10283" ProductID="99" Quantity="3"/>
     </Order>
  </Customer>
</Orders>'

-- Get a handle onto the XML document
EXEC sp_xml_preparedocument @handle OUTPUT, @OrdersXML

-- Use the OPENXML rowset provider against the handle to parse/query the XML
SELECT *
 FROM OPENXML(@handle, '/Orders/Customer/Order')
 WITH (
  CustomerName varchar(max) '../@ContactName',
  OrderDate date)
```

This code allows you to query and work with the XML text as if it were relational data. The output looks like this:

```
CustomerName    OrderDate
--------------- --------------
Charlie Herb    2011-11-04
Charlie Herb    2011-11-16
Matt Hink       2011-11-23
```

This code first calls *sp_xml_preparedocument* to get a handle over the XML of customer orders. The handle is passed as the first parameter to *OPENXML*. The second parameter is an XPath expression that specifies the *row pattern*, and this identifies the nodes within the XML that are to be processed as rows. In this example, the XPath expression */Orders/Customer/Order* drills down to the order level for each customer. There are three orders in the XML, so the query produces three rows with order dates (one for each order). The customer name is not available at the order level; it must be retrieved by reaching "up" one level for the *Customer* element's *ContactName* attribute using a column pattern. This is achieved using the *WITH* clause. In this example, the *CustomerName* column is based on the column pattern *../@ContactName* to obtain the *ContactName* attribute (remember that in XPath an @-symbol means "attribute") from the parent *Customer* node (as denoted by the *../* path syntax).

Querying XML Data Using XQuery

Storing XML in the database is one thing; querying it efficiently is another. Prior to the *xml* data type in SQL Server 2005, you had to deconstruct the XML and move element and attribute data into relational columns to perform a query on the XML data residing in the text column. You could also resort to some other searching mechanism, such as character pattern matching or full-text search, neither of which provides completely reliable parsing capability. Today, XQuery provides a native and elegant way to parse and query XML data in SQL Server.

Understanding XQuery Expressions and XPath

XQuery is a language used to query and process XML data. XQuery is a W3C standard, and its specification is located at *http://www.w3.org/TR/xquery/*. The XQuery specification contains several descriptions of requirements, use cases, and data models. We encourage you to review the specification to get a full understanding of what XQuery is all about. For now, we will explain enough to cover the basics. After reading this section, you will be able to select, filter, and update XML data using XQuery.

Because XQuery is an XML language, all the rules of XML apply. XQuery uses lowercase element names ("keywords"), and because XML itself is case-sensitive, you must take this into account when writing queries. Although XQuery has some powerful formatting and processing commands, it is primarily a query language (as its name suggests), so we will focus here on writing queries. The body of a query consists of two parts: an XPath expression and a *FLWOR* (pronounced "flower") expression. (FLWOR is an acronym based on the primitive XQuery keywords *for, let, where, order by,* and *return*.)

XPath Expressions

XPath, another W3C standard (*http://www.w3.org/TR/xpath*), uses path expressions to identify specific nodes and attributes in an XML document. These path expressions are similar to the syntax you see when you work with a computer file system (for example, C:\folder\myfile.doc). Take a look at the following XML document:

```
<catalog>
  <book category="ITPro">
    <title>Windows Step By Step</title>
    <author>Jeff Hay</author>
    <price>49.99</price>
  </book>
  <book category="Developer">
    <title>Learning ADO .NET</title>
    <author>Holly Holt</author>
    <price>39.93</price>
  </book>
  <book category="ITPro">
    <title>Administering IIS</title>
    <author>Jed Brown</author>
    <price>59.99</price>
  </book>
</catalog>
```

The following XPath expression selects the root element catalog:

```
/catalog
```

This XPath expression selects all the book elements of the catalog root element:

```
/catalog/book
```

And this XPath expression selects all the author elements of all the book elements of the catalog root element:

```
/catalog/book/author
```

XPath enables you to specify a subset of data within the XML (via its location within the XML structure) that you want to work with. XQuery is more robust and allows you to perform more complex queries against the XML data using *FLWOR* expressions combined with XPath.

FLWOR Expressions

Just as *SELECT, FROM, WHERE, GROUP BY*, and *ORDER BY* form the basis of the SQL selection logic, the *for, let, where, order by,* and *return* (*FLWOR*) keywords form the basis of every XQuery query you write. You use the *for* and *let* keywords to assign variables and iterate through the data within the context of the XQuery query. The *where* keyword works as a restriction and outputs the value of the variable.

For example, the following basic XQuery query uses the XPath expression */catalog/book* to obtain a reference to all the *<book>* nodes, and the *for* keyword initiates a loop, but only of elements where the *category* attribute is equal to *"ITPro"*. This simple code snippet iterates through each */catalog/book* node using the *$b* variable with the *for* statement only where the category attribute is *"ITPro"* and then returns as output the resulting information in descending order by the author's name using the *order* keyword:

```
for $b in /catalog/book
 where $b/@category="ITPro"
 order by $b/author[1] descending
 return ($b)
```

Listing 6-27 shows a simple example that uses this XQuery expression on an *xml* data type variable. XML is assigned to the variable, and then the preceding XQuery expression is used in the *query* method (explained in the next section) of the *xml* data type.

LISTING 6-27 A simple XQuery example.

```
DECLARE @Books xml = '
<catalog>
  <book category="ITPro">
    <title>Windows Step By Step</title>
    <author>Jeff Hay</author>
    <price>49.99</price>
  </book>
  <book category="Developer">
    <title>Learning ADO .NET</title>
    <author>Holly Holt</author>
    <price>39.93</price>
  </book>
  <book category="ITPro">
    <title>Administering IIS</title>
    <author>Ted Bremer</author>
    <price>59.99</price>
  </book>
</catalog>'

SELECT @Books.query('
  <ITProBooks>
    {
      for $b in /catalog/book
      where $b/@category="ITPro"
      order by $b/author[1] descending
      return ($b)
    }
  </ITProBooks>')
```

The results are as follows:

```
<ITProBooks>
  <book category="ITPro">
    <title>Administering IIS</title>
    <author>Ted Bremer</author>
    <price>59.99</price>
  </book>
  <book category="ITPro">
    <title>Windows Step By Step</title>
    <author>Jeff Hay</author>
    <price>49.99</price>
  </book>
</ITProBooks>
```

Notice that Ted's record is first because the order is descending by the *author* element. Holly's record is not in the output because the *category* element is restricted to *"ITPro"*. There is a root element wrapped around the XQuery statement with *<ITProBooks>* and *</ITProBooks>*, so all the results for IT books extracted from source XML having a catalog root element are contained inside of an *ITProBooks* root element.

SQL Server XQuery in Action

SQL Server has a standards-based implementation of XQuery that directly supports XQuery functions on the *xml* data type by using five methods of the *xml* data type, as shown here:

- ***xml.exist*** Uses XQuery input to return 0, 1, or *NULL*, depending on the result of the query. This method returns 0 if no elements match, 1 if there is a match, and *NULL* if there is no XML data on which to query. The *xml.exist* method is often used for query predicates.
- ***xml.value*** Accepts an XQuery query that resolves to a single value as input and returns a SQL Server scalar type.
- ***xml.query*** Accepts an XQuery query that resolves to multiple values as input and returns an *xml* data type stream as output.
- ***xml.nodes*** Accepts an XQuery query as input and returns a single-column rowset from the XML document. In essence, this method shreds XML into multiple smaller XML results.
- ***xml.modify*** Allows you to insert, delete, or modify nodes or sequences of nodes in an *xml* data type instance using an XQuery data manipulation language (DML).

We will discuss all of these methods shortly. But first, you'll create some sample data in a simple table that contains speakers at a software developer conference and the corresponding classes they will teach. Traditionally, you would normalize such data and have a one-to-many relationship between a speakers table and a classes table. Taking an XML approach instead, you will model this as one table with the speakers' information and one XML column with the speakers' classes. In the real world, you might encounter this scenario when you have a speaker and his or her classes represented in a series of one-to-many tables in a back-office database. Then for the web database, you might "publish" a database on a frequent time interval (such as a reporting database) or transform normalized data and use the XML column for easy HTML display with extensible stylesheet transformation (XSLT).

First, create a schema for the XML data, as shown in Listing 6-28. The schema defines the data types and required properties for particular XML elements in the list of classes that will be maintained for each speaker.

LISTING 6-28 Creating an XML schema definition for speaker classes.

```
USE master
GO

IF EXISTS(SELECT name FROM sys.databases WHERE name = 'SampleDB')
 DROP DATABASE SampleDB
GO

CREATE DATABASE SampleDB
GO

USE SampleDB
GO

CREATE XML SCHEMA COLLECTION ClassesXSD AS '
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="class">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>
  <xs:element name="classes">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="class" minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="speakerBio" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>
</xs:schema>'
```

Next, create the *Speaker* table (and indexes), as shown in Listing 6-29. Notice that the *xml* column, *ClassesXML*, uses the *ClassesXSD* XSD schema we just created in Listing 6-28.

LISTING 6-29 Creating the *Speaker* table with the typed (XSD schema-based) indexed XML column *ClassesXML*.

```
CREATE TABLE Speaker(
 SpeakerId int IDENTITY PRIMARY KEY,
 SpeakerName varchar(50),
 Country varchar(25),
 ClassesXML xml(ClassesXSD) NOT NULL)

-- Create primary XML index
CREATE PRIMARY XML INDEX ix_speakers
        ON Speaker(ClassesXML)

-- Create secondary structural (path) XML index
CREATE XML INDEX ix_speakers_path ON Speaker(ClassesXML)
 USING XML INDEX ix_speakers FOR PATH
```

XQuery runs more efficiently when there is an XML index on the XML column. As you learned earlier, an XML index works only if there is a primary key constraint on the table (such as the *SpeakerId* primary key column in the *Speaker* table). The code in Listing 6-29 creates a primary and then a structural (*PATH*) index because our examples will apply a lot of *where* restrictions on the values of particular elements. It's also important to remember that XQuery works more efficiently if it is strongly typed, so you should always use a schema (XSD) on your XML column for the best performance. Without a schema, the SQL Server XQuery engine assumes that everything is untyped and simply treats it as string data.

You're now ready to get data into the table by using some *INSERT* statements, as shown in Listing 6-30. The final *INSERT* statement, *'Bad Speaker'*, will fail because it does not contain a *<classes>* element as required by the *ClassesXSD* schema. (Because XML is case sensitive, its *<CLASSES>* element is not a match for the *<classes>* element specified as required in the schema.)

LISTING 6-30 Populating the *Speaker* table with sample data.

```
INSERT INTO Speaker VALUES('Jeff Hay', 'USA', '
  <classes speakerBio="Jeff has solid security experience from years of hacking">
    <class name="Writing Secure Code for ASP .NET" />
    <class name="Using XQuery to Manipulate XML Data in SQL Server 2012" />
    <class name="SQL Server and Oracle Working Together" />
    <class name="Protecting against SQL Injection Attacks" />
  </classes>')

INSERT INTO Speaker VALUES('Holly Holt', 'Canada', '
  <classes speakerBio="Holly is a Canadian-born database professional">
    <class name="SQL Server Profiler" />
    <class name="Advanced SQL Querying Techniques" />
    <class name="SQL Server and Oracle Working Together" />
  </classes>')

INSERT INTO Speaker VALUES('Ted Bremer', 'USA', '
  <classes speakerBio="Ted specializes in client development">
    <class name="Smart Client Stuff" />
    <class name="More Smart Client Stuff" />
  </classes>')

INSERT INTO Speaker VALUES('Bad Speaker', 'France', '
  <CLASSES SPEAKERBIO="Jean has case-sensitivity issues">
          <class name="SQL Server Index" />
          <class name="SQL Precon" />
  </CLASSES>')
```

Now that you have some data, it's time to start writing some XQuery expressions in T-SQL. To do this, you will use the query-based methods of the *xml* data type inside a regular T-SQL query.

xml.exist

Having XML in the database is almost useless unless you can query the elements and attributes of the XML data natively. XQuery becomes very useful when you use it to search XML based on the values of a particular element or attribute. The *xml.exist* method accepts an XQuery query as input and returns 0, 1, or *NULL*, depending on the result of the query: 0 is returned if no elements match, 1 is returned if there is a match, and *NULL* is returned if there is no data to query on. For example, Listing 6-31 shows how to test whether a particular node exists within an XML document.

LISTING 6-31 A simple *xml.exist* example.

```
DECLARE @SomeData xml = '
<classes>
        <class name="SQL Server Index"/>
        <class name="SQL Precon"/>
</classes>'

SELECT
 @SomeData.exist('/classes') AS HasClasses,
 @SomeData.exist('/dogs') AS HasDogs
```

This query produces the following output:

```
HasClasses HasDogs
---------- -------
1          0
```

You will most likely use the return value of *xml.exist* (0, 1, or *NULL*) as part of a *WHERE* clause. This lets you run a T-SQL query and restrict the query on a value of a particular XML element. For example, here is an XQuery expression that finds every *<class>* element beneath *<classes>* with a *name* attribute containing the phrase *"SQL Server"*:

```
/classes/class/@name[contains(., "SQL Server ")]
```

Listing 6-32 shows how you put this expression to work.

LISTING 6-32 Using *xml.exist* to test for an attribute value.

```
SELECT * FROM Speaker
 WHERE
  ClassesXML.exist('/classes/class/@name[contains(., "SQL Server")]') = 1
```

The results look like this:

```
SpeakerId  SpeakerName  Country  ClassesXML
---------  -----------  -------  ----------
1          Jeff Hay     USA      <classes speakerBio="Jeff has solid security...
2          Holly Holt   Canada   <classes speakerBio="Holly is a Canadian-bor...
```

Jeff and Holly (but not Ted) each give one or more SQL Server classes. The XML returned in these results look like this for Jeff:

```
<classes speakerBio="Jeff has solid security experience based on years of hacking">
  <class name="Writing Secure Code for ASP .NET" />
  <class name="Using XQuery to Manipulate XML Data in SQL Server 2012" />
  <class name="SQL Server and Oracle Working Together" />
  <class name="Protecting against SQL Injection Attacks" />
</classes>
```

Listing 6-33 shows a query similar to the previous one. This version demonstrates how to seamlessly integrate XQuery with ordinary filtering of relational columns, by simply building out the *WHERE* clause to further restrict by *Country* for USA only.

LISTING 6-33 Combining XQuery with relational column filtering.

```
SELECT * FROM Speaker
 WHERE
  ClassesXML.exist('/classes/class/@name[contains(., "SQL Server")]') = 1
  AND Country = 'USA'
```

Executing this query returns only Jeff. SQL Server will filter out the other two rows because Ted does not have any SQL Server classes and Holly is from Canada.

xml.value

The *xml.value* method takes an XQuery expression *that resolves to a single value* and returns it, cast as the SQL Server data type you specify. You can leverage this very powerful method to completely shield the internal XML representation of your data, and expose ordinary scalar values with ordinary SQL Server data types instead. Consider the query in Listing 6-34.

LISTING 6-34 Using *xml.value* to represent XML data elements as scalar SQL Server data typed-columns

```
.SELECT
  SpeakerName,
  Country,
  ClassesXML.value('/classes[1]/@speakerBio','varchar(max)') AS SpeakerBio,
  ClassesXML.value('count(/classes/class)', 'int') AS SessionCount
 FROM
  Speaker
 ORDER BY
  ClassesXML.value('count(/classes/class)', 'int')
```

From the output generated by this query, there is no indication that—behind the scenes—the source for some of the output comes from an embedded XML document, stored in an *xml* data type column, and then shredded with XQuery:

```
SpeakerName Country SpeakerBio                                               SessionCount
----------- ------- -------------------------------------------------------- ------------
Ted Bremer  USA     Ted specializes in client development                    2
Holly Holt  Canada  Holly is a Canadian-born database professional           3
Jeff Hay    USA     Jeff has solid security experience from years of hacking 4
```

The *SpeakerName* and *Country* columns came right out of the *Speaker* table. However, the *SpeakerBio* and *SessionCount* columns were each extracted from the *ClassesXML* column using *xml.value* with an XQuery expression and a SQL Server data type that the expression's result was cast to. Because you are requesting a specific data type, the XQuery expression *must* resolve to a *single* value. That value can come from a node element's inner text, attribute, or XQuery function, but it must be a *single* value. For *SpeakerBio*, the XQuery drills into the *classes* element for the *speakerBio* attribute, extracts its value, and casts it as a *varchar(max)* type. The XQuery for *SessionCount* invokes the *count* function to return the number of *class* elements nested beneath the *classes* element cast as an *int*. The same XQuery is used again in the *ORDER BY* clause, so that the results of the query themselves are sorted by a value derived from data embedded in XML content.

You can build views and TVFs over queries such as this, and create an effective abstraction layer over the way XML is stored internally in your database. This means you can alter the XSD schemas and then adjust the XQuery expressions in your views and TVFs accordingly, such that consumers remain unaffected. Indeed, you could even transparently switch from XML storage to traditional column storage and back again, without disturbing any existing clients. SQL Server thus provides extremely flexible abstraction in both directions, because you've seen the myriad of ways to dynamically construct and serve XML from relational column data with the various *FOR XML* options earlier in the chapter. This flexibility means you can choose just the right degree of XML integration in your database that best suits your needs—whether that involves persisting XML data, constructing XML data, or both.

xml.query

The *xml.query* method accepts and executes an XQuery expression much like the *xml.value* method, but it always returns an *xml* data type result. So unlike *xml.value*, the XQuery expression doesn't need to resolve to a single value, and can easily return multiple values as a subset of the source XML. But furthermore, it can transform that source XML and produce entirely different XML—even injecting values from other non-*xml* columns living the in same row as the *xml* column being queried. Listing 6-35 demonstrates how this is achieved using FLWOR expressions and *sql:column* (a SQL Server XQuery extension).

LISTING 6-35 Using *xml.query* with FLWOR expressions and *sql:column* for XML transformations.

```
SELECT
  SpeakerId,
  ClassesXML.query('
    let $c := count(/classes/class)
    let $b := data(/classes[1]/@speakerBio)
    return
      <SpeakerInfo>
        <Name>{sql:column("SpeakerName")}</Name>
        <Country>{sql:column("Country")}</Country>
       <Bio>{$b}</Bio>
        <Sessions count="{$c}">
          {
            for $s in /classes/class
            let $n := data($s/@name)
            order by $n
            return
              <Session>{$n}</Session>
          }
        </Sessions>
      </SpeakerInfo>
           ') AS SpeakerInfo
 FROM
  Speaker
```

The XML returned in these results looks like this for Jeff:

```
<SpeakerInfo>
  <Name>Jeff Hay</Name>
  <Country>USA</Country>
  <Bio>Jeff has solid security experience from years of hacking</Bio>
  <Sessions count="4">
    <Session>Protecting against SQL Injection Attacks</Session>
    <Session>SQL Server and Oracle Working Together</Session>
    <Session>Using XQuery to Manipulate XML Data in SQL Server 2012</Session>
    <Session>Writing Secure Code for ASP .NET</Session>
  </Sessions>
</SpeakerInfo>
```

Let's explain the code in detail. The XQuery expression in the *xml.query* method on the *ClassesXML* column begins with a FLWOR expression. The two *let* statements use XPath expressions to capture the speaker's number of classes (using the *count* function) and bio text (using the *data* function), and stores the results into the variables *$c* and *$b* respectively. Then the *return* statement defines the shape of the XML to be constructed, starting with the root node's *<SpeakerInfo>* element. Inside the root node, the *<Name>* and *<Country>* elements are returned, with values extracted from the *SpeakerName* and *Country* columns. These are values that are not present in the XML being parsed by *xml.query*, but are available as ordinary columns elsewhere in the same row, and are exposed using the special *sql:column* SQL Server extension to XQuery.

Next, the *<Sessions>* element is returned with a *count* attribute that returns the number of class elements beneath the source XML's classes element. Within *<Sessions>*, a new (nested) FLWOR expression is used to iterate the speaker's classes and build a sequence of *<Session>* elements. The *for* statement loops through the source XML's *classes* element for each nested *class* element and stores it into the variable *$s*. The *let* statement then uses the *data* function to capture the string value inside the *name* attribute of the *class* element in *$s* and stores it into the variable *$n*. The inner FLWOR expression results (that is, the sequence of elements returned by the upcoming *return* statement) are sorted by name using the *order by* statement. Finally, the *return* statement generates a new *<Session>* element. The session name is rendered as the inner text of the *<Session>* element. This XQuery has essentially transformed the *<Classes>* and *<Class name="title">* structure of the source XML to a *<Sessions>* and *<Session>title</Session>* structure.

The *sql:variable* function is another very powerful SQL Server extension to XQuery. With it, you can easily parameterize your XQuery expressions using ordinary T-SQL parameters. This technique is demonstrated in Listing 6-36.

LISTING 6-36 Using *xml.query* with *sql:variable* for parameterized transformations.

```
DECLARE @Category varchar(max) = 'SQL Server'

SELECT
  SpeakerName,
  Country,
  ClassesXML.query('
    <classes
      category="{sql:variable("@Category")}"
      speakerBio="{data(/classes[1]/@speakerBio)}">
      {
        for $c in /classes/class
        where $c/@name[contains(., sql:variable("@Category"))]
        return $c
      }
    </classes>') AS ClassesXML
 FROM
  Speaker
 WHERE
  ClassesXML.exist
   ('/classes/class/@name[contains(., sql:variable("@Category"))]') = 1
```

The results look like this:

```
SpeakerName  Country  ClassesXML
-----------  -------  ----------
Jeff Hay     USA      <classes category="SQL Server" speakerBio="Jeff has solid e...
Holly Holt   Canada   <classes category="SQL Server" speakerBio="Holly is a Canad...
```

The XML returned in these results looks like this for Jeff:

```
<classes category="SQL Server"
   speakerBio="Jeff has solid security experience from years of hacking">
  <class name="Using XQuery to Manipulate XML Data in SQL Server 2012" />
  <class name="SQL Server and Oracle Working Together" />
</classes>
```

In this example, the T-SQL *@Category* parameter is assigned the value *SQL Server*, and the *sql:variable* is then used in several places to reference *@Category*. The first reference adds a *category* attribute to the *classes* element. The second reference applies filtering against the *name* attribute using *contains* in the inner FLWOR expression's *where* statement, and the last reference applies filtering at the resultset row level in the *SELECT* statement's *WHERE* clause. Thus, only rows having *SQL Server* in the name of at least one class are returned in the resultset, and within those rows, only classes having SQL Server in their name are returned as elements in *ClassesXML* (all other non-SQL Server classes are filtered out).

Our last *xml.query* example demonstrates how to combine child elements into a delimited string value, as shown in Listing 6-37.

LISTING 6-37 Using *xml.query* with *CONVERT* to combine child elements.

```
SELECT
  SpeakerName,
  Country,
  CONVERT(varchar(max), ClassesXML.query('
    for $s in /classes/class
    let $n := data($s/@name)
    let $p := concat($n, "|")
    return $p')) AS SessionList
 FROM
  Speaker
```

The *SessionList* column produced by this query contains a single pipe-delimited string containing the names of all the classes given by the speaker:

```
SpeakerName Country SessionList
----------- ------- ------------------------------------------------------------------
Jeff Hay    USA     Writing Secure Code for ASP .NET| Using XQuery to Manipulate XML Da...
Holly Holt  Canada  SQL Server Profiler| Advanced SQL Querying Techniques| SQL Server a...
Ted Bremer  USA     Smart Client Stuff| More Smart Client Stuff|
```

This XQuery expression in Listing 6-37 simply iterates each *class* element, extracts the *name* attribute, and concatenates it with a pipe symbol, appending each result to build a single string. Although the elements are ultimately combined to form a single value, they are still multiple values from an XPath perspective, and so *xml.value* cannot be used. Instead, *xml.query* produces the concatenated string, and *CONVERT* is used to cast the result as a *varchar(max)* data type.

XML DML

The W3C XQuery specification does not provide a way for you to modify XML data as you can modify relational table data using the *INSERT, UPDATE*, and *DELETE* keywords in T-SQL. So Microsoft has created its own XML data manipulation language, XML DML, which is included in its own XQuery implementation.

XML DML gives you three ways to manipulate the XML data of a column via the *xml.modify* method:

- ***xml.modify(insert)*** Allows you to insert a node or sequence of nodes into the *xml* data type instance you are working with.
- ***xml.modify(delete)*** Allows you to delete zero or more nodes that are the result of the output sequence of the XQuery expression you specify.
- ***xml.modify(replace)*** Modifies the value of a single node.

xml.modify(insert)

The *xml.modify(insert)* method allows you to insert a node or sequence of nodes into the *xml* data type instance you are working with. You use the *xml.modify* method in conjunction with a T-SQL *UPDATE* statement and, if necessary, a T-SQL or XQuery *where* clause (or both). For example, the code in Listing 6-38 adds another <*class*> element to Jeff's <*classes*> element in *ClassesXML*.

LISTING 6-38 Using *xml.modify* to insert a new element.

```
UPDATE Speaker
 SET ClassesXML.modify('
  insert
    <class name="Ranking and Windowing Functions in SQL Server" />
  into
    /classes[1]')
 WHERE SpeakerId = 1
```

xml.modify(delete)

The *xml.modify(delete)* method deletes zero or more nodes based on the criteria you specify. For example, the code in Listing 6-39 deletes the fourth <*class*> element from Jeff's <*classes*> element in *ClassesXML*.

LISTING 6-39 Using *xml.modify* to delete an element.

```
UPDATE Speaker
 SET ClassesXML.modify('delete /classes/class[4]')
 WHERE SpeakerId = 1
```

xml.modify(replace)

Finally, the *xml.modify(replace)* method allows you to replace XML data with new information. For example, the code in Listing 6-40 updates the *name* attribute in the third <*class*> element of Jeff's <*classes*> element in *ClassesXML*.

LISTING 6-40 Using *xml.modify* to update an element.

```
UPDATE Speaker
 SET ClassesXML.modify('
  replace value of /classes[1]/class[3]/@name[1]
         with "Getting SQL Server and Oracle to Work Together"')
 WHERE SpeakerId = 1
```

Summary

XML is ubiquitous nowadays, and in this chapter, we have taken a fairly extensive tour of the *FOR XML* clause and its various options, as well as the *xml* data type and its data manipulation mechanisms, XQuery and XML DML. As you have seen, SQL Server provides a rich feature set of XML technologies. At times, you will want to store XML in the database, other times you will want to serve XML from the database, and still other times you may want to do both. Whatever your XML needs are, the *xml* data type allows you to work with XML natively at the database level. Armed with this data type and the ability to query it using XQuery, you can fully exploit the power of XML inside your SQL Server databases and build smart, XML-aware applications.

Index

About the Authors

Leonard Lobel is a Microsoft MVP in SQL Server and a Principal Consultant at Tallan, Inc., a Microsoft National Systems Integrator and Gold Competency Partner. With over 30 years of experience, Lenni is one of the industry's leading .NET and SQL Server experts, having consulted for Tallan's clients in a variety of domains, including publishing, financial services, retail, health care, and e-commerce. Lenni has served as chief architect and lead developer on large scale projects, as well as advisor to many high-profile clients.

About Tallan

Tallan (*http://www.tallan.com*) is a, national technology consulting firm that provides web development, business intelligence, customer relationship management, custom development, and integration services to customers in the financial services, health care, government, retail, education, and manufacturing industries.

Tallan is one of 40 Microsoft National Systems Integrators (NSI) in the United States, and a member of Microsoft's Business Intelligence Partner Advisory Council. For more than 25 years, Tallan's hands-on, collaborative approach has enabled its clients to obtain real cost and time savings, increase revenues, and generate competitive advantage.

Lenni is also chief technology officer (CTO) and cofounder of Sleek Technologies, Inc., a New York-based development shop with an early adopter philosophy toward new technologies. He is a sought after and highly rated speaker at industry conferences such as Visual Studio Live!, SQL PASS, SQL Bits, and local technology user group meetings. He is also lead author of this book's previous edition, *Programming Microsoft SQL Server 2008*. Lenni can be reached at *lenni.lobel@tallan.com* or *lenni.lobel@sleektech.com*.

Andrew J. Brust is Founder and CEO of Blue Badge Insights (*http://www.bluebadgeinsights.com*), an analysis, strategy and advisory firm serving Microsoft customers and partners. Brust pens ZDNet's "Big on Data" blog (*http://bit.ly/bigondata*); is a Microsoft Regional Director and MVP; an advisor to the New York Technology Council; Co-Chair of Visual Studio Live!;

a frequent speaker at industry events and a columnist for Visual Studio Magazine. He has been a participant in the Microsoft ecosystem for 20 years; worked closely with both Microsoft's Redmond-based corporate team and its field organization for the last 10; has served on Microsoft’s Business Intelligence Partner Advisory Council; and is a member of several Microsoft "insiders" groups that supply him with insight around important technologies out of Redmond.

Paul Delcogliano is a technology director at Broadridge Financial Services, Inc. Paul has been working with the Microsoft .NET Framework since its first public introduction and has been developing Microsoft SQL Server applications even longer. He builds systems for a diverse range of platforms including Microsoft Windows, the Internet, and mobile devices. Paul has authored many articles and columns for various trade publications on a variety of topics. He can be reached by email at *pdelco@hotmail.com*.

Paul would like to thank his family for their patience and understanding while he was frantically trying to meet his deadlines. He would also like to thank Lenni for offering him another opportunity to contribute to the book. The second time around was better than the first.