Microsoft

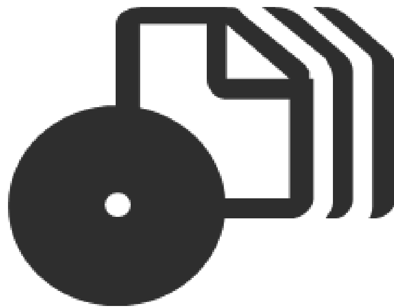
EXAM 70-461

Querying Microsoft® SQL Server® 2012

Itzik Ben-Gan
Dejan Sarka
Ron Talmage

Training Kit

How to access your CD files



The print edition of this book includes a CD. To access the CD files, go to http://aka.ms/666054/files, and look for the Downloads tab.

Note: Use a desktop web browser, as files may not be accessible from all ereader devices.

Questions? Please contact: mspinput@microsoft.com

Microsoft Press

Exam 70-461: Querying Microsoft SQL Server 2012

OBJECTIVE	CHAPTER	LESSON
1. CREATE DATABASE OBJECTS		
1.1 Create and alter tables using T-SQL syntax (simple statements).	8	1
1.2 Create and alter views (simple statements).	9 15	1 1
1.3 Design views.	9	1
1.4 Create and modify constraints (simple statements).	8	2
1.5 Create and alter DML triggers.	13	2
2. WORK WITH DATA		
2.1 Query data by using SELECT statements.	1 2 3 4 5 6 8 9 12	1 2 All lessons All lessons 3 Lessons 2 and 3 2 2 3
2.2 Implement sub-queries.	4 5 17	2 2 1
2.3 Implement data types.	2 3	2 1
2.4 Implement aggregate queries.	5	Lessons 1 and 3
2.5 Query and manage XML data.	7	All lessons
3. MODIFY DATA		
3.1 Create and alter stored procedures (simple statements).	13	All lessons
3.2 Modify data by using INSERT, UPDATE, and DELETE statements.	10 11	All lessons 3
3.3 Combine datasets.	2 4 11	2 3 2
3.4 Work with functions.	2 3 6 13	2 1 3 3
4. TROUBLESHOOT & OPTIMIZE		
4.1 Optimize queries.	12 14 15 17	Both lessons All lessons All lessons All lessons
4.2 Manage transactions.	12	1
4.3 Evaluate the use of row-based operations vs. set-based operations.	16	1
4.4 Implement error handling.	12 16	2 1

Exam Objectives The exam objectives listed here are current as of this book's publication date. Exam objectives are subject to change at any time without prior notice and at Microsoft's sole discretion. Please visit the Microsoft Learning website for the most current listing of exam objectives: *http://www.microsoft.com/learning/en/us/exam.aspx?ID=70-461&locale=en-us*.

Microsoft

Querying Microsoft® SQL Server® 2012

Exam 70-461 Training Kit

Itzik Ben-Gan
Dejan Sarka
Ron Talmage

Copyright © 2012 by SolidQuality Global SL.

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-6605-4

Ninth Printing: March 2015

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*.

Microsoft and the trademarks listed at *http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx* are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

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 & Developmental Editor: Ken Jones

Production Editor: Melanie Yarbrough

Editorial Production: Online Training Solutions, Inc.

Technical Reviewer: Herbert Albert

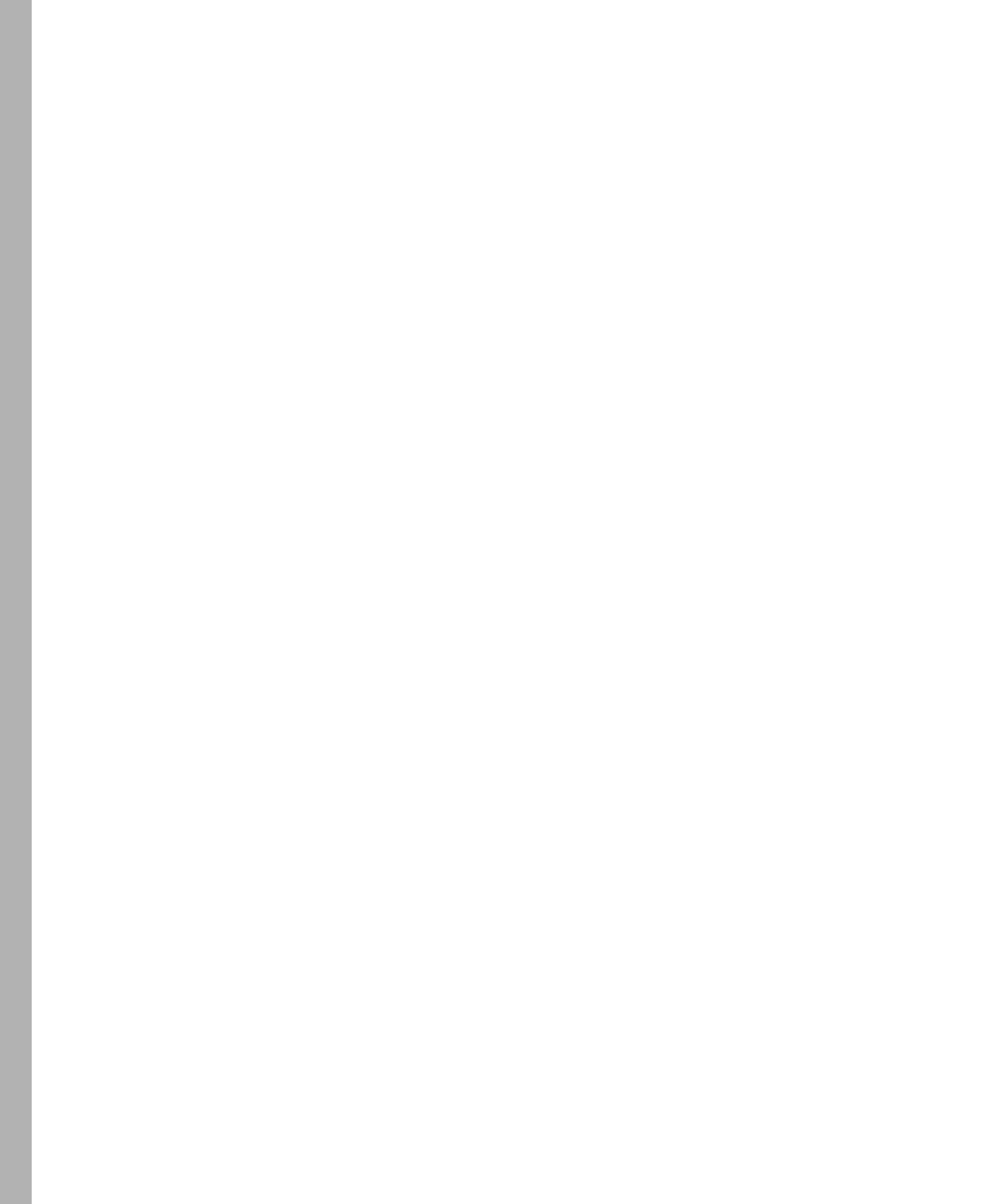
Indexer: WordCo Indexing Services

Cover Design: Twist Creative • Seattle

Cover Composition: Zyg Group, LLC

Contents at a Glance

	Introduction	*xxv*
CHAPTER 1	Foundations of Querying	1
CHAPTER 2	Getting Started with the SELECT Statement	29
CHAPTER 3	Filtering and Sorting Data	61
CHAPTER 4	Combining Sets	101
CHAPTER 5	Grouping and Windowing	149
CHAPTER 6	Querying Full-Text Data	191
CHAPTER 7	Querying and Managing XML Data	221
CHAPTER 8	Creating Tables and Enforcing Data Integrity	265
CHAPTER 9	Designing and Creating Views, Inline Functions, and Synonyms	299
CHAPTER 10	Inserting, Updating, and Deleting Data	329
CHAPTER 11	Other Data Modification Aspects	369
CHAPTER 12	Implementing Transactions, Error Handling, and Dynamic SQL	411
CHAPTER 13	Designing and Implementing T-SQL Routines	469
CHAPTER 14	Using Tools to Analyze Query Performance	517
CHAPTER 15	Implementing Indexes and Statistics	549
CHAPTER 16	Understanding Cursors, Sets, and Temporary Tables	599
CHAPTER 17	Understanding Further Optimization Aspects	631
	Index	*677*



Contents

Introduction **xxv**

Chapter 1 Foundations of Querying **1**

Before You Begin. 1

Lesson 1: Understanding the Foundations of T-SQL. 2

- Evolution of T-SQL 2
- Using T-SQL in a Relational Way 5
- Using Correct Terminology 10
- Lesson Summary 13
- Lesson Review 13

Lesson 2: Understanding Logical Query Processing. 14

- T-SQL As a Declarative English-Like Language 14
- Logical Query Processing Phases 15
- Lesson Summary 23
- Lesson Review 23

Case Scenarios. 24

- Case Scenario 1: Importance of Theory 24
- Case Scenario 2: Interviewing for a Code Reviewer Position 24

Suggested Practices 25

- Visit T-SQL Public Newsgroups and Review Code 25
- Describe Logical Query Processing 25

Answers. 26

- Lesson 1 26
- Lesson 2 27

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Case Scenario 1 28
Case Scenario 2 28

Chapter 2 Getting Started with the SELECT Statement 29

Before You Begin. 29

Lesson 1: Using the FROM and SELECT Clauses. 30
The FROM Clause 30
The SELECT Clause 31
Delimiting Identifiers 34
Lesson Summary 36
Lesson Review 36

Lesson 2: Working with Data Types and
Built-in Functions. 37
Choosing the Appropriate Data Type 37
Choosing a Data Type for Keys 41
Date and Time Functions 44
Character Functions 46
CASE Expression and Related Functions 49
Lesson Summary 55
Lesson Review 55

Case Scenarios 56
Case Scenario 1: Reviewing the Use of Types 56
Case Scenario 2: Reviewing the Use of Functions 57

Suggested Practices 57
Analyze the Data Types in the Sample Database 57
Analyze Code Samples in Books Online for SQL Server 2012 57

Answers. 58
Lesson 1 58
Lesson 2 58
Case Scenario 1 59
Case Scenario 2 60

Chapter 3 Filtering and Sorting Data **61**

Before You Begin. 61

Lesson 1: Filtering Data with Predicates 62

Predicates, Three-Valued Logic, and Search Arguments 62

Combining Predicates 66

Filtering Character Data 68

Filtering Date and Time Data 70

Lesson Summary 73

Lesson Review 74

Lesson 2: Sorting Data 74

Understanding When Order Is Guaranteed 75

Using the ORDER BY Clause to Sort Data 76

Lesson Summary 83

Lesson Review 83

Lesson 3: Filtering Data with TOP and OFFSET-FETCH 84

Filtering Data with TOP 84

Filtering Data with OFFSET-FETCH 88

Lesson Summary 93

Lesson Review 94

Case Scenarios 95

Case Scenario 1: Filtering and Sorting Performance Recommendations 95

Case Scenario 2: Tutoring a Junior Developer 95

Suggested Practices 96

Identify Logical Query Processing Phases and Compare Filters 96

Understand Determinism 96

Answers. 97

Lesson 1 97

Lesson 2 98

Lesson 3 98

Case Scenario 1 99

Case Scenario 2 100

Chapter 4 Combining Sets 101

Before You Begin. 101

Lesson 1: Using Joins 102

Cross Joins 102

Inner Joins 105

Outer Joins 108

Multi-Join Queries 112

Lesson Summary 116

Lesson Review 117

Lesson 2: Using Subqueries, Table Expressions, and the APPLY Operator 117

Subqueries 118

Table Expressions 121

APPLY 128

Lesson Summary 135

Lesson Review 136

Lesson 3: Using Set Operators 136

UNION and UNION ALL 137

INTERSECT 139

EXCEPT 140

Lesson Summary 142

Lesson Review 142

Case Scenarios 143

Case Scenario 1: Code Review 143

Case Scenario 2: Explaining Set Operators 144

Suggested Practices 144

Combine Sets 144

Answers. 145

Lesson 1 145

Lesson 2 145

Lesson 3 146

Case Scenario 1 147

Case Scenario 2 147

Chapter 5 Grouping and Windowing 149

Before You Begin. 149

Lesson 1: Writing Grouped Queries. 150

Working with a Single Grouping Set 150

Working with Multiple Grouping Sets 155

Lesson Summary 161

Lesson Review 162

Lesson 2: Pivoting and Unpivoting Data. 163

Pivoting Data 163

Unpivoting Data 166

Lesson Summary 171

Lesson Review 171

Lesson 3: Using Window Functions 172

Window Aggregate Functions 172

Window Ranking Functions 176

Window Offset Functions 178

Lesson Summary 183

Lesson Review 183

Case Scenarios 184

Case Scenario 1: Improving Data Analysis Operations 184

Case Scenario 2: Interviewing for a Developer Position 185

Suggested Practices 185

Logical Query Processing 185

Answers. 186

Lesson 1 186

Lesson 2 187

Lesson 3 187

Case Scenario 1 188

Case Scenario 2 188

Chapter 6 Querying Full-Text Data 191

Before You Begin. 191

Lesson 1: Creating Full-Text Catalogs and Indexes 192

Full-Text Search Components 192

Creating and Managing Full-Text Catalogs and Indexes 194

Lesson Summary 201

Lesson Review 201

Lesson 2: Using the CONTAINS and FREETEXT Predicates 202

The CONTAINS Predicate 202

The FREETEXT Predicate 204

Lesson Summary 208

Lesson Review 208

Lesson 3: Using the Full-Text and Semantic Search Table-Valued Functions 209

Using the Full-Text Search Functions 209

Using the Semantic Search Functions 210

Lesson Summary 214

Lesson Review 214

Case Scenarios. 215

Case Scenario 1: Enhancing the Searches 215

Case Scenario 2: Using the Semantic Search 215

Suggested Practices 215

Check the FTS Dynamic Management Views and Backup and Restore of a Full-Text Catalog and Indexes 215

Answers. 217

Lesson 1 217

Lesson 2 217

Lesson 3 218

Case Scenario 1 219

Case Scenario 2 219

Chapter 7 Querying and Managing XML Data 221

Before You Begin. 221

Lesson 1: Returning Results As XML with FOR XML 222

- Introduction to XML 222
- Producing XML from Relational Data 226
- Shredding XML to Tables 230
- Lesson Summary 234
- Lesson Review 234

Lesson 2: Querying XML Data with XQuery. 235

- XQuery Basics 236
- Navigation 240
- FLWOR Expressions 243
- Lesson Summary 248
- Lesson Review 248

Lesson 3: Using the XML Data Type. 249

- When to Use the XML Data Type 250
- XML Data Type Methods 250
- Using the XML Data Type for Dynamic Schema 252
- Lesson Summary 259
- Lesson Review 259

Case Scenarios. 260

- Case Scenario 1: Reports from XML Data 260
- Case Scenario 2: Dynamic Schema 261

Suggested Practices 261

- Query XML Data 261

Answers. 262

- Lesson 1 262
- Lesson 2 262
- Lesson 3 263
- Case Scenario 1 264
- Case Scenario 2 264

Chapter 8 Creating Tables and Enforcing Data Integrity 265

Before You Begin. 265

Lesson 1: Creating and Altering Tables 265

Introduction 266

Creating a Table 267

Altering a Table 276

Choosing Table Indexes 276

Lesson Summary 280

Lesson Review 280

Lesson 2: Enforcing Data Integrity 281

Using Constraints 281

Primary Key Constraints 282

Unique Constraints 283

Foreign Key Constraints 285

Check Constraints 286

Default Constraints 288

Lesson Summary 292

Lesson Review 292

Case Scenarios 293

Case Scenario 1: Working with Table Constraints 293

Case Scenario 2: Working with Unique and Default Constraints 293

Suggested Practices 294

Create Tables and Enforce Data Integrity 294

Answers. 295

Lesson 1 295

Lesson 2 295

Case Scenario 1 296

Case Scenario 2 297

Chapter 9 Designing and Creating Views, Inline Functions, and Synonyms 299

Before You Begin. 299

Lesson 1: Designing and Implementing Views and Inline Functions. . . .300

Introduction 300

Views 300

Inline Functions 307

Lesson Summary 313

Lesson Review 314

Lesson 2: Using Synonyms. 315

Creating a Synonym 315

Comparing Synonyms with Other Database Objects 318

Lesson Summary 322

Lesson Review 322

Case Scenarios 323

Case Scenario 1: Comparing Views, Inline Functions, and Synonyms 323

Case Scenario 2: Converting Synonyms to Other Objects 323

Suggested Practices 324

Design and Create Views, Inline Functions, and Synonyms 324

Answers. 325

Lesson 1 325

Lesson 2 326

Case Scenario 1 326

Case Scenario 2 327

Chapter 10 Inserting, Updating, and Deleting Data 329

Before You Begin. 329

Lesson 1: Inserting Data. 330

Sample Data 330

INSERT VALUES 331

INSERT SELECT 333

INSERT EXEC 334

SELECT INTO 335

Lesson Summary 340

Lesson Review 340

Lesson 2: Updating Data 341

Sample Data 341

UPDATE Statement 342

UPDATE Based on Join 344

Nondeterministic UPDATE 346

UPDATE and Table Expressions 348

UPDATE Based on a Variable 350

UPDATE All-at-Once 351

Lesson Summary 354

Lesson Review 355

Lesson 3: Deleting Data 356

Sample Data 356

DELETE Statement 357

TRUNCATE Statement 358

DELETE Based on a Join 359

DELETE Using Table Expressions 360

Lesson Summary 362

Lesson Review 363

Case Scenarios. 363

Case Scenario 1: Using Modifications That Support Optimized Logging 364

Case Scenario 2: Improving a Process That Updates Data 364

Suggested Practices 364

DELETE vs. TRUNCATE 364

Answers. 366
Lesson 1 366
Lesson 2 367
Lesson 3 367
Case Scenario 1 368
Case Scenario 2 368

Chapter 11 Other Data Modification Aspects 369

Before You Begin. 369
Lesson 1: Using the Sequence Object and IDENTITY Column Property . 370
Using the IDENTITY Column Property 370
Using the Sequence Object 374
Lesson Summary 381
Lesson Review 381
Lesson 2: Merging Data 382
Using the MERGE Statement 383
Lesson Summary 392
Lesson Review 393
Lesson 3: Using the OUTPUT Option. 394
Working with the OUTPUT Clause 394
INSERT with OUTPUT 395
DELETE with OUTPUT 396
UPDATE with OUTPUT 397
MERGE with OUTPUT 397
Composable DML 399
Lesson Summary 403
Lesson Review 404
Case Scenarios 405
Case Scenario 1: Providing an Improved Solution for Generating Keys 405
Case Scenario 2: Improving Modifications 405
Suggested Practices 406
Compare Old and New Features 406

Answers. 407
Lesson 1 407
Lesson 2 408
Lesson 3 408
Case Scenario 1 409
Case Scenario 2 409

Chapter 12 Implementing Transactions, Error Handling, and Dynamic SQL 411

Before You Begin. 411
Lesson 1: Managing Transactions and Concurrency. 412
Understanding Transactions 412
Types of Transactions 415
Basic Locking 422
Transaction Isolation Levels 426
Lesson Summary 434
Lesson Review 434
Lesson 2: Implementing Error Handling 435
Detecting and Raising Errors 435
Handling Errors After Detection 440
Lesson Summary 449
Lesson Review 450
Lesson 3: Using Dynamic SQL. 450
Dynamic SQL Overview 451
SQL Injection 456
Using sp_executesql 457
Lesson Summary 462
Lesson Review 462
Case Scenarios. 463
Case Scenario 1: Implementing Error Handling 463
Case Scenario 2: Implementing Transactions 463
Suggested Practices 464
Implement Error Handling 464

Answers. 465
Lesson 1 465
Lesson 2 466
Lesson 3 467
Case Scenario 1 468
Case Scenario 2 468

Chapter 13 Designing and Implementing T-SQL Routines 469

Before You Begin. 469
Lesson 1: Designing and Implementing Stored Procedures 470
Understanding Stored Procedures 470
Executing Stored Procedures 475
Branching Logic 477
Developing Stored Procedures 481
Lesson Summary 489
Lesson Review 490
Lesson 2: Implementing Triggers 490
DML Triggers 491
AFTER Triggers 492
INSTEAD OF Triggers 495
DML Trigger Functions 496
Lesson Summary 499
Lesson Review 500
Lesson 3: Implementing User-Defined Functions 501
Understanding User-Defined Functions 501
Scalar UDFs 502
Table-Valued UDFs 503
Limitations on UDFs 505
UDF Options 506
UDF Performance Considerations 506
Lesson Summary 509
Lesson Review 510

Case Scenarios 511
Case Scenario 1: Implementing Stored Procedures and UDFs 511
Case Scenario 2: Implementing Triggers 511

Suggested Practices 512
Use Stored Procedures, Triggers, and UDFs 512

Answers. 513
Lesson 1 513
Lesson 2 514
Lesson 3 514
Case Scenario 1 515
Case Scenario 2 516

Chapter 14 Using Tools to Analyze Query Performance 517

Before You Begin. 517

Lesson 1: Getting Started with Query Optimization. 518
Query Optimization Problems and the Query Optimizer 518
SQL Server Extended Events, SQL Trace, and SQL Server Profiler 523
Lesson Summary 528
Lesson Review 528

Lesson 2: Using SET Session Options and
Analyzing Query Plans 529
SET Session Options 529
Execution Plans 532
Lesson Summary 538
Lesson Review 538

Lesson 3: Using Dynamic Management Objects 539
Introduction to Dynamic Management Objects 539
The Most Important DMOs for Query Tuning 540
Lesson Summary 544
Lesson Review 544

Case Scenarios 544
Case Scenario 1: Analysis of Queries 545
Case Scenario 2: Constant Monitoring 545

Suggested Practices 545
Learn More About Extended Events, Execution Plans, and Dynamic Management Objects 545
Answers. 546
Lesson 1 546
Lesson 2 546
Lesson 3 547
Case Scenario 1 548
Case Scenario 2 548

Chapter 15 Implementing Indexes and Statistics 549

Before You Begin. 550
Lesson 1: Implementing Indexes 550
Heaps and Balanced Trees 550
Implementing Nonclustered Indexes 564
Implementing Indexed Views 568
Lesson Summary 573
Lesson Review 573
Lesson 2: Using Search Arguments 573
Supporting Queries with Indexes 574
Search Arguments 578
Lesson Summary 584
Lesson Review 584
Lesson 3: Understanding Statistics. 585
Auto-Created Statistics 585
Manually Maintaining Statistics 589
Lesson Summary 592
Lesson Review 592
Case Scenarios. 593
Case Scenario 1: Table Scans 593
Case Scenario 2: Slow Updates 594
Suggested Practices 594
Learn More About Indexes and How Statistics Influence Query Execution 594

Answers. 595
Lesson 1 595
Lesson 2 595
Lesson 3 596
Case Scenario 1 597
Case Scenario 2 597

Chapter 16 Understanding Cursors, Sets, and Temporary Tables 599

Before You Begin. 599
Lesson 1: Evaluating the Use of Cursor/Iterative Solutions vs. Set-Based Solutions. 600
The Meaning of "Set-Based" 600
Iterations for Operations That Must Be Done Per Row 601
Cursor vs. Set-Based Solutions for Data Manipulation Tasks 604
Lesson Summary 610
Lesson Review 610
Lesson 2: Using Temporary Tables vs. Table Variables 611
Scope 612
DDL and Indexes 613
Physical Representation in tempdb 616
Transactions 617
Statistics 618
Lesson Summary 623
Lesson Review 624
Case Scenarios 624
Case Scenario 1: Performance Improvement Recommendations for Cursors and Temporary Objects 625
Case Scenario 2: Identifying Inaccuracies in Answers 625
Suggested Practices 626
Identify Differences 626

Answers. 627
Lesson 1 627
Lesson 2 628
Case Scenario 1 628
Case Scenario 2 629

Chapter 17 Understanding Further Optimization Aspects 631

Before You Begin. 632
Lesson 1: Understanding Plan Iterators 632
Access Methods 632
Join Algorithms 638
Other Plan Iterators 641
Lesson Summary 647
Lesson Review 647
Lesson 2: Using Parameterized Queries and Batch Operations 647
Parameterized Queries 648
Batch Processing 653
Lesson Summary 660
Lesson Review 660
Lesson 3: Using Optimizer Hints and Plan Guides. 661
Optimizer Hints 661
Plan Guides 666
Lesson Summary 670
Lesson Review 670
Case Scenarios 671
Case Scenario 1: Query Optimization 671
Case Scenario 2: Table Hint 671
Suggested Practices 672
Analyze Execution Plans and Force Plans 672

Answers. 673
Lesson 1 673
Lesson 2 674
Lesson 3 674
Case Scenario 1 675
Case Scenario 2 675

Index 677

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Introduction

This Training Kit is designed for information technology (IT) professionals who need to query data in Microsoft SQL Server 2012 and who also plan to take Exam 70-461, "Querying Microsoft SQL Server 2012." It is assumed that before you begin using this Training Kit, you have a foundation-level understanding of using Transact-SQL (T-SQL) to query data in SQL Server 2012 and have some experience using the product. Although this book helps prepare you for the 70-461 exam, you should consider it as one part of your exam preparation plan. Meaningful, real-world experience with SQL Server 2012 is required to pass this exam.

The material covered in this Training Kit and on Exam 70-461 relates to the technologies in SQL Server 2012. The topics in this Training Kit cover what you need to know for the exam as described on the Skills Measured tab for the exam, which is available at *http://www.microsoft.com/learning/en/us/exam.aspx?ID=70-461&locale=en-us#tab2*.

By using this Training Kit, you will learn how to do the following:

- Create database objects
- Work with data
- Modify data
- Troubleshoot and optimize T-SQL code

Refer to the objective mapping page in the front of this book to see where in the book each exam objective is covered.

System Requirements

The following are the minimum system requirements your computer needs to meet to complete the practice exercises in this book and to run the companion CD.

SQL Server Software and Data Requirements

You can find the minimum SQL Server software and data requirements here:

- **SQL Server 2012** You need access to a SQL Server 2012 instance with a logon that has permissions to create new databases—preferably one that is a member of the sysadmin role. For the purposes of this Training Kit, you can use almost any edition of on-premises SQL Server (Standard, Enterprise, Business Intelligence, or Developer), both 32-bit and 64-bit editions. If you don't have access to an existing SQL Server instance, you can install a trial copy that you can use for 180 days. You can download a trial copy from *http://www.microsoft.com/sqlserver/en/us/get-sql-server/try-it.aspx*.

- **SQL Server 2012 Setup Feature Selection** In the Feature Selection dialog box of the SQL Server 2012 setup program, choose at minimum the following components:
  - Database Engine Services
  - Full-Text And Semantic Extractions For Search
  - Documentation Components
  - Management Tools—Basic (required)
  - Management Tools—Complete (recommended)
- **TSQL2012 sample database and source code** Most exercises in this Training Kit use a sample database called TSQL2012. The companion content for the Training Kit includes a compressed file called TK70461-YYYYMMDD.zip (where YYYYMMDD reflects the date of the last revision) that contains the book's source code, exercises, and a script file called TSQL2012.sql that you use to create the sample database. You can download the compressed file from the website at *http://go.microsoft.com/FWLink/?Linkid=263548* and from the authors' website at *http://tsql.solidq.com/books/tk70461/*.

Hardware and Operating System Requirements

You can find the minimum hardware and operating system requirements for installing and running SQL Server 2012 at *http://msdn.microsoft.com/en-us/library/ms143506(v=sql.110).aspx*.

Using the Companion CD

A companion CD is included with this Training Kit. The companion CD contains the following:

- **Practice tests** You can reinforce your understanding of the topics covered in this Training Kit by using electronic practice tests that you customize to meet your needs. You can practice for the 70-461 certification exam by using tests created from a pool of 200 practice exam questions, which give you many practice exams to help you prepare for the certification exam. These questions are not from the exam; they are for practice and preparation.
- **An eBook** An electronic version (eBook) of this book is included for when you do not want to carry the printed book with you.

How to Install the Practice Tests

To install the practice test software from the companion CD to your hard disk, perform the following steps:

1. Insert the companion CD into your CD drive and accept the license agreement. A CD menu appears.

> ***NOTE*** **IF THE CD MENU DOES NOT APPEAR**
>
> **If the CD menu or the license agreement does not appear, AutoRun might be disabled on your computer. Refer to the Readme.txt file on the CD for alternate installation instructions.**

2. Click Practice Tests and follow the instructions on the screen.

How to Use the Practice Tests

To start the practice test software, follow these steps:

1. Click Start, All Programs, and then select Microsoft Press Training Kit Exam Prep.

   A window appears that shows all the Microsoft Press Training Kit exam prep suites installed on your computer.

2. Double-click the practice test you want to use.

When you start a practice test, you choose whether to take the test in Certification Mode, Study Mode, or Custom Mode:

- **Certification Mode** Closely resembles the experience of taking a certification exam. The test has a set number of questions. It is timed, and you cannot pause and restart the timer.
- **Study Mode** Creates an untimed test during which you can review the correct answers and the explanations after you answer each question.
- **Custom Mode** Gives you full control over the test options so that you can customize them as you like.

In all modes, the user interface when you are taking the test is basically the same but with different options enabled or disabled, depending on the mode.

When you review your answer to an individual practice test question, a "References" section is provided that lists where in the Training Kit you can find the information that relates to that question and provides links to other sources of information. After you click Test Results to score your entire practice test, you can click the Learning Plan tab to see a list of references for every objective.

How to Uninstall the Practice Tests

To uninstall the practice test software for a Training Kit, use the Program And Features option in Windows Control Panel.

Acknowledgments

A book is put together by many more people than the authors whose names are listed on the cover page. We'd like to express our gratitude to the following people for all the work they have done in getting this book into your hands: Herbert Albert (technical editor), Lilach Ben-Gan (project manager), Ken Jones (acquisitions and developmental editor), Melanie Yarbrough (production editor), Jaime Odell (copyeditor), Marlene Lambert (PTQ project manager), Jeanne Craver (graphics), Jean Trenary (desktop publisher), Kathy Krause (proofreader), and Kerin Forsyth (PTQ copyeditor).

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/9780735666054.

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*.

Preparing for the Exam

Microsoft certification exams are a great way to build your resume and let the world know about your level of expertise. Certification exams validate your on-the-job experience and product knowledge. While there is no substitution for on-the-job experience, preparation through study and hands-on practice can help you prepare for the exam. We recommend that you round out your exam preparation plan by using a combination of available study materials and courses. For example, you might use the Training Kit and another study guide for your "at home" preparation, and take a Microsoft Official Curriculum course for the classroom experience. Choose the combination that you think works best for you.

> ***NOTE*** **PASSING THE EXAM**
>
> **Take a minute (well, one minute and two seconds) to look at the "Passing a Microsoft Exam" video at *http://www.youtube.com/watch?v=Jp5qg2NhgZ0&feature=youtu.be*. It's true. Really!**

CHAPTER 1

Foundations of Querying

Exam objectives in this chapter:

- Work with Data
  - Query data by using SELECT statements.

Transact-SQL (T-SQL) is the main language used to manage and manipulate data in Microsoft SQL Server. This chapter lays the foundations for querying data by using T-SQL. The chapter describes the roots of this language, terminology, and the mindset you need to adopt when writing T-SQL code. It then moves on to describe one of the most important concepts you need to know about the language—logical query processing.

> IMPORTANT
> ***Have you read page xxx?***
> **It contains valuable information regarding the skills you need to pass the exam.**

Although this chapter doesn't directly target specific exam objectives other than discussing the design of the SELECT statement, which is the main T-SQL statement used to query data, the rest of the chapters in this Training Kit do. However, the information in this chapter is critical in order to correctly understand the rest of the book.

Lessons in this chapter:

- Lesson 1: Understanding the Foundations of T-SQL
- Lesson 2: Understanding Logical Query Processing

Before You Begin

To complete the lessons in this chapter, you must have:

- An understanding of basic database concepts.
- Experience working with SQL Server Management Studio (SSMS).
- Some experience writing T-SQL code.
- Access to a SQL Server 2012 instance with the sample database TSQL2012 installed. (Please see the book's introduction for details on how to create the sample database.)

Lesson 1: Understanding the Foundations of T-SQL

Many aspects of computing, like programming languages, evolve based on intuition and the current trend. Without strong foundations, their lifespan can be very short, and if they do survive, often the changes are very rapid due to changes in trends. T-SQL is different, mainly because it has strong foundations—mathematics. You don't need to be a mathematician to write good SQL (though it certainly doesn't hurt), but as long as you understand what those foundations are, and some of their key principles, you will better understand the language you are dealing with. Without those foundations, you can still write T-SQL code—even code that runs successfully—but it will be like eating soup with a fork!

After this lesson, you will be able to:

- Describe the foundations that T-SQL is based on.
- Describe the importance of using T-SQL in a relational way.
- Use correct terminology when describing T-SQL–related elements.

Estimated lesson time: 40 minutes

Evolution of T-SQL

As mentioned, unlike many other aspects of computing, T-SQL is based on strong mathematical foundations. Understanding some of the key principals from those foundations can help you better understand the language you are dealing with. Then you will think in T-SQL terms when coding in T-SQL, as opposed to coding with T-SQL while thinking in procedural terms.

Figure 1-1 illustrates the evolution of T-SQL from its core mathematical foundations.

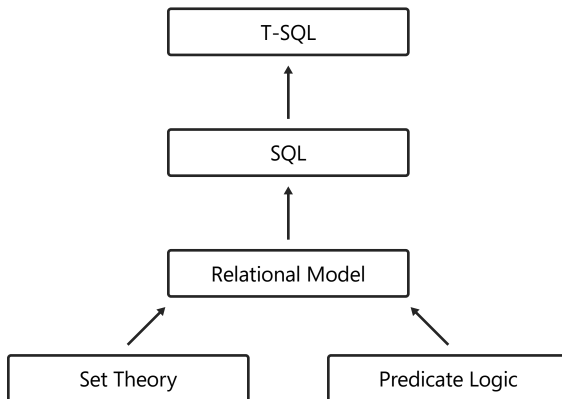


FIGURE 1-1 Evolution of T-SQL.

T-SQL is the main language used to manage and manipulate data in Microsoft's main relational database management system (RDBMS), SQL Server—whether on premises or in the cloud (Microsoft Windows Azure SQL Database). SQL Server also supports other languages, like Microsoft Visual C# and Microsoft Visual Basic, but T-SQL is usually the preferred language for data management and manipulation.

T-SQL is a dialect of standard SQL. SQL is a standard of both the International Organization for Standards (ISO) and the American National Standards Institute (ANSI). The two standards for SQL are basically the same. The SQL standard keeps evolving with time. Following is a list of the major revisions of the standard so far:

- SQL-86
- SQL-89
- SQL-92
- SQL:1999
- SQL:2003
- SQL:2006
- SQL:2008
- SQL:2011

All leading database vendors, including Microsoft, implement a dialect of SQL as the main language to manage and manipulate data in their database platforms. Therefore, the core language elements look the same. However, each vendor decides which features to implement and which not to. Also, the standard sometimes leaves some aspects as an implementation choice. Each vendor also usually implements extensions to the standard in cases where the vendor feels that an important feature isn't covered by the standard.

Writing in a standard way is considered a best practice. When you do so, your code is more portable. Your knowledge is more portable, too, because it is easy for you to start working with new platforms. When the dialect you're working with supports both a standard and a nonstandard way to do something, you should always prefer the standard form as your default choice. You should consider a nonstandard option only when it has some important benefit to you that is not covered by the standard alternative.

As an example of when to choose the standard form, T-SQL supports two "not equal to" operators: <> and !=. The former is standard and the latter is not. This case should be a no-brainer: go for the standard one!

As an example of when the choice of standard or nonstandard depends on the circumstances, consider the following: T-SQL supports multiple functions that convert a source value to a target type. Among them are the CAST and CONVERT functions. The former is standard and the latter isn't. The nonstandard CONVERT function has a style argument that CAST doesn't support. Because CAST is standard, you should consider it your default choice for conversions. You should consider using CONVERT only when you need to rely on the style argument.

Yet another example of choosing the standard form is in the termination of T-SQL statements. According to standard SQL, you should terminate your statements with a semicolon. T-SQL currently doesn't make this a requirement for all statements, only in cases where there would otherwise be ambiguity of code elements, such as in the WITH clause of a common table expression (CTE). You should still follow the standard and terminate all of your statements even where it is currently not required.

Standard SQL is based on the *relational model*, which is a mathematical model for data management and manipulation. The relational model was initially created and proposed by Edgar F. Codd in 1969. Since then, it has been explained and developed by Chris Date, Hugh Darwen, and others.

A common misconception is that the name "relational" has to do with relationships between tables (that is, foreign keys). Actually, the true source for the model's name is the mathematical concept *relation*.

A relation in the relational model is what SQL calls a *table*. The two are not synonymous. You could say that a table is an attempt by SQL to represent a relation (in addition to a relation variable, but that's not necessary to get into here). Some might say that it is not a very successful attempt. Even though SQL is based on the relational model, it deviates from it in a number of ways. But it's important to note that as you understand the model's principles, you can use SQL—or more precisely, the dialect you are using—in a relational way. More on this, including a further reading recommendation, is in the next section, "Using T-SQL in a Relational Way."

Getting back to a relation, which is what SQL attempts to represent with a table: a relation has a heading and a body. The heading is a set of attributes (what SQL attempts to represent with columns), each of a given type. An attribute is identified by name and type name. The body is a set of tuples (what SQL attempts to represent with rows). Each tuple's heading is the heading of the relation. Each value of each tuple's attribute is of its respective type.

Some of the most important principals to understand about T-SQL stem from the relational model's core foundations—set theory and predicate logic.

Remember that the heading of a relation is a set of attributes, and the body a set of tuples. So what is a set? According to the creator of mathematical set theory, Georg Cantor, a *set* is described as follows:

> *By a "set" we mean any collection M into a whole of definite, distinct objects m (which are called the "elements" of M) of our perception or of our thought.*
>
> —George Cantor, in "Georg Cantor" by Joseph W. Dauben (Princeton University Press, 1990)

There are a number of very important principles in this definition that, if understood, should have direct implications on your T-SQL coding practices. For one, notice the term *whole*. A set should be considered as a whole. This means that you do not interact with the individual elements of the set, rather with the set as a whole.

Notice the term *distinct*—a set has no duplicates. Codd once remarked on the no duplicates aspect: "If something is true, then saying it twice won't make it any truer." For example, the set {a, b, c} is considered equal to the set {a, a, b, c, c, c}.

Another critical aspect of a set doesn't explicitly appear in the aforementioned definition by Cantor, but rather is implied—there's no relevance to the order of elements in a set. In contrast, a sequence (which is an *ordered* set), for example, does have an order to its elements. Combining the no duplicates and no relevance to order aspects means that the set {a, b, c} is equal to the set {b, a, c, c, a, c}.

The other branch of mathematics that the relational model is based on is called predicate logic. A *predicate* is an expression that when attributed to some object, makes a proposition either true or false. For example, "salary greater than $50,000" is a predicate. You can evaluate this predicate for a specific employee, in which case you have a proposition. For example, suppose that for a particular employee, the salary is $60,000. When you evaluate the proposition for that employee, you get a true proposition. In other words, a predicate is a parameterized proposition.

The relational model uses predicates as one of its core elements. You can enforce data integrity by using predicates. You can filter data by using predicates. You can even use predicates to define the data model itself. You first identify propositions that need to be stored in the database. Here's an example proposition: an order with order ID 10248 was placed on February 12, 2012 by the customer with ID 7, and handled by the employee with ID 3. You then create predicates from the propositions by removing the data and keeping the heading. Remember, the heading is a set of attributes, each identified by name and type name. In this example, you have orderid INT, orderdate DATE, custid INT, and empid INT.

Quick Check

1. **What are the mathematical branches that the relational model is based on?**
2. **What is the difference between T-SQL and SQL?**

Quick Check Answer

1. **Set theory and predicate logic.**
2. **SQL is standard; T-SQL is the dialect of and extension to SQL that Microsoft implements in its RDBMS—SQL Server.**

Using T-SQL in a Relational Way

As mentioned in the previous section, T-SQL is based on SQL, which in turn is based on the relational model. However, there are a number of ways in which SQL—and therefore, T-SQL—deviates from the relational model. But the language gives you enough tools so that if you understand the relational model, you can use the language in a relational manner, and thus write more-correct code.

MORE INFO SQL AND RELATIONAL THEORY

For detailed information about the differences between SQL and the relational model and how to use SQL in a relational way, see *SQL and Relational Theory,* Second Edition by C. J. Date (O'Reilly Media, 2011). It's an excellent book that all database practitioners should read.

Remember that a relation has a heading and a body. The heading is a set of attributes and the body is a set of tuples. Remember from the definition of a set that a set is supposed to be considered as a whole. What this translates to in T-SQL is that you're supposed to write queries that interact with the tables as a whole. You should try to avoid using iterative constructs like cursors and loops that iterate through the rows one at a time. You should also try to avoid thinking in iterative terms because this kind of thinking is what leads to iterative solutions.

For people with a procedural programming background, the natural way to interact with data (in a file, record set, or data reader) is with iterations. So using cursors and other iterative constructs in T-SQL is, in a way, an extension to what they already know. However, the correct way from the relational model's perspective is not to interact with the rows one at a time; rather, use relational operations and return a relational result. This, in T-SQL, translates to writing queries.

Remember also that a set has no duplicates. T-SQL doesn't always enforce this rule. For example, you can create a table without a key. In such a case, you are allowed to have duplicate rows in the table. To follow relational theory, you need to enforce uniqueness in your tables—for example, by using a primary key or a unique constraint.

Even when the table doesn't allow duplicate rows, a query against the table can still return duplicate rows in its result. You'll find further discussion about duplicates in subsequent chapters, but here is an example for illustration purposes. Consider the following query.

```
USE TSQL2012;

SELECT country
FROM HR.Employees;
```

The query is issued against the TSQL2012 sample database. It returns the country attribute of the employees stored in the HR.Employees table. According to the relational model, a relational operation against a relation is supposed to return a relation. In this case, this should translate to returning the set of countries where there are employees, with an emphasis on set, as in no duplicates. However, T-SQL doesn't attempt to remove duplicates by default.

Here's the output of this query.

```
Country
---------------
USA
USA
USA
USA
UK
UK
UK
USA
UK
```

In fact, T-SQL is based more on multiset theory than on set theory. A *multiset* (also known as a bag or a superset) in many respects is similar to a set, but can have duplicates. As mentioned, the T-SQL language does give you enough tools so that if you want to follow relational theory, you can do so. For example, the language provides you with a DISTINCT clause to remove duplicates. Here's the revised query.

```
SELECT DISTINCT country
FROM HR.Employees;
```

Here's the revised query's output.

```
Country
---------------
UK
USA
```

Another fundamental aspect of a set is that there's no relevance to the order of the elements. For this reason, rows in a table have no particular order, conceptually. So when you issue a query against a table and don't indicate explicitly that you want to return the rows in particular presentation order, the result is supposed to be relational. Therefore, you shouldn't assume any specific order to the rows in the result, never mind what you know about the physical representation of the data, for example, when the data is indexed.

As an example, consider the following query.

```
SELECT empid, lastname
FROM HR.Employees;
```

When this query was run on one system, it returned the following output, which looks like it is sorted by the column lastname.

```
empid  lastname
------ -------------
5      Buck
8      Cameron
1      Davis
9      Dolgopyatova
2      Funk
7      King
3      Lew
4      Peled
6      Suurs
```

Even if the rows were returned in a different order, the result would have still been considered correct. SQL Server can choose between different physical access methods to process the query, knowing that it doesn't need to guarantee the order in the result. For example, SQL Server could decide to parallelize the query or scan the data in file order (as opposed to index order).

If you do need to guarantee a specific presentation order to the rows in the result, you need to add an ORDER BY clause to the query, as follows.

```
SELECT empid, lastname
FROM HR.Employees
ORDER BY empid;
```

This time, the result isn't relational—it's what standard SQL calls a *cursor*. The order of the rows in the output is guaranteed based on the empid attribute. Here's the output of this query.

```
empid  lastname
------ -------------
1      Davis
2      Funk
3      Lew
4      Peled
5      Buck
6      Suurs
7      King
8      Cameron
9      Dolgopyatova
```

The heading of a relation is a set of attributes that are supposed to be identified by name and type name. There's no order to the attributes. Conversely, T-SQL does keep track of ordinal positions of columns based on their order of appearance in the table definition. When you issue a query with SELECT *, you are guaranteed to get the columns in the result based on definition order. Also, T-SQL allows referring to ordinal positions of columns from the result in the ORDER BY clause, as follows.

```
SELECT empid, lastname
FROM HR.Employees
ORDER BY 1;
```

Beyond the fact that this practice is not relational, think about the potential for error if at some point you change the SELECT list and forget to change the ORDER BY list accordingly. Therefore, the recommendation is to always indicate the names of the attributes that you need to order by.

T-SQL has another deviation from the relational model in that it allows defining result columns based on an expression without assigning a name to the target column. For example, the following query is valid in T-SQL.

```
SELECT empid, firstname + ' ' + lastname
FROM HR.Employees;
```

This query generates the following output.

```
empid
------ ------------------
1      Sara Davis
2      Don Funk
3      Judy Lew
4      Yael Peled
5      Sven Buck
6      Paul Suurs
7      Russell King
8      Maria Cameron
9      Zoya Dolgopyatova
```

But according to the relational model, all attributes must have names. In order for the query to be relational, you need to assign an alias to the target attribute. You can do so by using the AS clause, as follows.

```
SELECT empid, firstname + ' ' + lastname AS fullname
FROM HR.Employees;
```

Also, T-SQL allows a query to return multiple result columns with the same name. For example, consider a join between two tables, T1 and T2, both with a column called keycol. T-SQL allows a SELECT list that looks like the following.

```
SELECT T1.keycol, T2.keycol ...
```

For the result to be relational, all attributes must have unique names, so you would need to use different aliases for the result attributes, as in the following.

```
SELECT T1.keycol AS key1, T2.keycol AS key2 ...
```

As for predicates, following the *law of excluded middle* in mathematical logic, a predicate can evaluate to true or false. In other words, predicates are supposed to use two-valued logic. However, Codd wanted to reflect the possibility for values to be missing in his model. He referred to two kinds of missing values: missing but applicable and missing but inapplicable. Take a mobilephone attribute of an employee as an example. A missing but applicable value would be if an employee has a mobile phone but did not want to provide this information, for example, for privacy reasons. A missing but inapplicable value would be when the employee simply doesn't have a mobile phone. According to Codd, a language based on his model

should provide two different marks for the two cases. T-SQL—again, based on standard SQL—implements only one general purpose mark called NULL for any kind of missing value. This leads to three-valued predicate logic. Namely, when a predicate compares two values, for example, mobilephone = '(425) 555-0136', if both are present, the result evaluates to either true or false. But if one of them is NULL, the result evaluates to a third logical value—unknown.

Note that some believe that a valid relational model should follow two-valued logic, and strongly object to the concept of NULLs in SQL. But as mentioned, the creator of the relational model believed in the idea of supporting missing values, and predicates that extend beyond two-valued logic. What's important from a perspective of coding with T-SQL is to realize that if the database you are querying supports NULLs, their treatment is far from being trivial. That is, you need to carefully understand what happens when NULLs are involved in the data you're manipulating with various query constructs, like filtering, sorting, grouping, joining, or intersecting. Hence, with every piece of code you write with T-SQL, you want to ask yourself whether NULLs are possible in the data you're interacting with. If the answer is yes, you want to make sure that you understand the treatment of NULLs in your query, and ensure that your tests address treatment of NULLs specifically.

Quick Check

1. **Name two aspects in which T-SQL deviates from the relational model.**
2. **Explain how you can address the two items in question 1 and use T-SQL in a relational way.**

Quick Check Answer

1. **A relation has a body with a distinct set of tuples. A table doesn't have to have a key. T-SQL allows referring to ordinal positions of columns in the ORDER BY clause.**
2. **Define a key in every table. Refer to attribute names—not their ordinal positions—in the ORDER BY clause.**

Using Correct Terminology

Your use of terminology reflects on your knowledge. Therefore, you should make an effort to understand and use correct terminology. When discussing T-SQL–related topics, people often use incorrect terms. And if that's not enough, even when you do realize what the correct terms are, you also need to understand the differences between the terms in T-SQL and those in the relational model.

As an example of incorrect terms in T-SQL, people often use the terms "field" and "record" to refer to what T-SQL calls "column" and "row," respectively. Fields and records are physical. Fields are what you have in user interfaces in client applications, and records are what you have in files and cursors. Tables are logical, and they have logical rows and columns.

Another example of an incorrect term is referring to "NULL values." A NULL is a mark for a missing value—not a value itself. Hence, the correct usage of the term is either "NULL mark" or just "NULL."

Besides using correct T-SQL terminology, it's also important to understand the differences between T-SQL terms and their relational counterparts. Remember from the previous section that T-SQL attempts to represent a relation with a table, a tuple with a row, and an attribute with a column; but the T-SQL concepts and their relational counterparts differ in a number of ways. As long as you are conscious of those differences, you can, and should, strive to use T-SQL in a relational way.

Quick Check

1. **Why are the terms "field" and "record" incorrect when referring to column and row?**
2. **Why is the term "NULL value" incorrect?**

Quick Check Answer

1. **Because "field" and "record" describe physical things, whereas columns and rows are logical elements of a table.**
2. **Because NULL isn't a value; rather, it's a mark for a missing value.**

PRACTICE Using T-SQL in a Relational Way

In this practice, you exercise your knowledge of using T-SQL in a relational way.

If you encounter a problem completing an exercise, you can install the completed projects from the companion content for this chapter and lesson.

EXERCISE 1 Identify Nonrelational Elements in a Query

In this exercise, you are given a query. Your task is to identify the nonrelational elements in the query.

1. Open SQL Server management Studio (SSMS) and connect to the sample database TSQL2012. (See the book's introduction for instructions on how to create the sample database and how to work with SSMS.)
2. Type the following query in the query window and execute it.

```
SELECT custid, YEAR(orderdate)
FROM Sales.Orders
ORDER BY 1, 2;
```

You get the following output shown here in abbreviated form.

```
custid
----------- -----------
1           2007
1           2007
1           2007
1           2008
1           2008
1           2008
2           2006
2           2007
2           2007
2           2008
...
```

3. Review the code and its output. The query is supposed to return for each customer and order year the customer ID (custid) and order year (YEAR(orderdate)). Note that there's no presentation ordering requirement from the query. Can you identify what the nonrelational aspects of the query are?

   Answer: The query doesn't alias the expression YEAR(orderdate), so there's no name for the result attribute. The query can return duplicates. The query forces certain presentation ordering to the result and uses ordinal positions in the ORDER BY clause.

EXERCISE 2 Make the Nonrelational Query Relational

In this exercise, you work with the query provided in Exercise 1 as your starting point. After you identify the nonrelational elements in the query, you need to apply the appropriate revisions to make it relational.

- In step 3 of Exercise 1, you identified the nonrelational elements in the last query. Apply revisions to the query to make it relational.

  A number of revisions are required to make the query relational.

  - Define an attribute name by assigning an alias to the expression YEAR(orderdate).
  - Add a DISTINCT clause to remove duplicates.
  - Also, remove the ORDER BY clause to return a relational result.
  - Even if there was a presentation ordering requirement (not in this case), you should not use ordinal positions; instead, use attribute names. Your code should look like the following.

    ```
    SELECT DISTINCT custid, YEAR(orderdate) AS orderyear
    FROM Sales.Orders;
    ```

Lesson Summary

- T-SQL is based on strong mathematical foundations. It is based on standard SQL, which in turn is based on the relational model, which in turn is based on set theory and predicate logic.
- It is important to understand the relational model and apply its principals when writing T-SQL code.
- When describing concepts in T-SQL, you should use correct terminology because it reflects on your knowledge.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. Why is it important to use standard SQL code when possible and know what is standard and what isn't? (Choose all that apply.)
   - **A.** It is not important to code using standard SQL.
   - **B.** Standard SQL code is more portable between platforms.
   - **C.** Standard SQL code is more efficient.
   - **D.** Knowing what standard SQL code is makes your knowledge more portable.
2. Which of the following is not a violation of the relational model?
   - **A.** Using ordinal positions for columns
   - **B.** Returning duplicate rows
   - **C.** Not defining a key in a table
   - **D.** Ensuring that all attributes in the result of a query have names
3. What is the relationship between SQL and T-SQL?
   - **A.** T-SQL is the standard language and SQL is the dialect in Microsoft SQL Server.
   - **B.** SQL is the standard language and T-SQL is the dialect in Microsoft SQL Server.
   - **C.** Both SQL and T-SQL are standard languages.
   - **D.** Both SQL and T-SQL are dialects in Microsoft SQL Server.

Lesson 2: Understanding Logical Query Processing

T-SQL has both logical and physical sides to it. The logical side is the conceptual interpretation of the query that explains what the correct result of the query is. The physical side is the processing of the query by the database engine. Physical processing must produce the result defined by logical query processing. To achieve this goal, the database engine can apply optimization. Optimization can rearrange steps from logical query processing or remove steps altogether—but only as long as the result remains the one defined by logical query processing. The focus of this lesson is *logical query processing*—the conceptual interpretation of the query that defines the correct result.

After this lesson, you will be able to:

- Understand the reasoning for the design of T-SQL.
- Describe the main logical query processing phases.
- Explain the reasons for some of the restrictions in T-SQL.

Estimated lesson time: 40 minutes

T-SQL As a Declarative English-Like Language

T-SQL, being based on standard SQL, is a declarative English-like language. In this language, declarative means you define *what* you want, as opposed to *imperative* languages that define also *how* to achieve what you want. Standard SQL describes the logical interpretation of the declarative request (the “what” part), but it’s the database engine’s responsibility to figure out how to physically process the request (the “how” part).

For this reason, it is important not to draw any performance-related conclusions from what you learn about logical query processing. That’s because logical query processing only defines the correctness of the query. When addressing performance aspects of the query, you need to understand how optimization works. As mentioned, optimization can be quite different from logical query processing because it’s allowed to change things as long as the result achieved is the one defined by logical query processing.

It’s interesting to note that the standard language SQL wasn’t originally called so; rather, it was called SEQUEL; an acronym for “structured English query language.” But then due to a trademark dispute with an airline company, the language was renamed to SQL, for “structured query language.” Still, the point is that you provide your instructions in an English-like manner. For example, consider the instruction, “Bring me a soda from the refrigerator.” Observe that in the instruction in English, the object comes before the location. Consider the following request in T-SQL.

```
SELECT shipperid, phone, companyname
FROM Sales.Shippers;
```

Observe the similarity of the query's keyed-in order to English. The query first indicates the SELECT list with the attributes you want to return and then the FROM clause with the table you want to query.

Now try to think of the order in which the request needs to be logically interpreted. For example, how would you define the instructions to a robot instead of a human? The original English instruction to get a soda from the refrigerator would probably need to be revised to something like, "Go to the refrigerator; open the door; get a soda; bring it to me."

Similarly, the logical processing of a query must first know which table is being queried before it can know which attributes can be returned from that table. Therefore, contrary to the keyed-in order of the previous query, the logical query processing has to be as follows.

```
FROM Sales.Shippers
SELECT shipperid, phone, companyname
```

This is a basic example with just two query clauses. Of course, things can get more complex. If you understand the concept of logical query processing well, you will be able to explain many things about the way the language behaves—things that are very hard to explain otherwise.

Logical Query Processing Phases

This section covers logical query processing and the phases involved. Don't worry if some of the concepts discussed here aren't clear yet. Subsequent chapters in this Training Kit provide more detail, and after you go over those, this topic should make more sense. To make sure you really understand these concepts, make a first pass over the topic now and then revisit it later after going over Chapters 2 through 5.

The main statement used to retrieve data in T-SQL is the SELECT statement. Following are the main query clauses specified in the order that you are supposed to type them (known as "keyed-in order"):

1. SELECT
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY

But as mentioned, the logical query processing order, which is the conceptual interpretation order, is different. It starts with the FROM clause. Here is the logical query processing order of the six main query clauses:

1. FROM
2. WHERE

3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

Each phase operates on one or more tables as inputs and returns a virtual table as output. The output table of one phase is considered the input to the next phase. This is in accord with operations on relations that yield a relation. Note that if an ORDER BY is specified, the result isn't relational. This fact has implications that are discussed later in this Training Kit, in Chapter 3, "Filtering and Sorting Data," and Chapter 4, "Combining Sets."

Consider the following query as an example.

```
SELECT country, YEAR(hiredate) AS yearhired, COUNT(*) AS numemployees
FROM HR.Employees
WHERE hiredate >= '20030101'
GROUP BY country, YEAR(hiredate)
HAVING COUNT(*) > 1
ORDER BY country , yearhired DESC;
```

This query is issued against the HR.Employees table. It filters only employees that were hired in or after the year 2003. It groups the remaining employees by country and the hire year. It keeps only groups with more than one employee. For each qualifying group, the query returns the hire year and count of employees, sorted by country and hire year, in descending order.

The following sections provide a brief description of what happens in each phase according to logical query processing.

1. Evaluate the FROM Clause

In the first phase, the FROM clause is evaluated. That's where you indicate the tables you want to query and table operators like joins if applicable. If you need to query just one table, you indicate the table name as the input table in this clause. Then, the output of this phase is a table result with all rows from the input table. That's the case in the following query: the input is the HR.Employees table (nine rows), and the output is a table result with all nine rows (only a subset of the attributes are shown).

```
empid  hiredate    country
------ ----------- --------
1      2002-05-01  USA
2      2002-08-14  USA
3      2002-04-01  USA
4      2003-05-03  USA
5      2003-10-17  UK
6      2003-10-17  UK
7      2004-01-02  UK
8      2004-03-05  USA
9      2004-11-15  UK
```

2. Filter Rows Based on the WHERE Clause

The second phase filters rows based on the predicate in the WHERE clause. Only rows for which the predicate evaluates to true are returned.

EXAM TIP

Rows for which the predicate evaluates to false, or evaluates to an unknown state, are not returned.

In this query, the WHERE filtering phase filters only rows for employees hired on or after January 1, 2003. Six rows are returned from this phase and are provided as input to the next one. Here's the result of this phase.

```
empid  hiredate    country
------ ----------- --------
4      2003-05-03  USA
5      2003-10-17  UK
6      2003-10-17  UK
7      2004-01-02  UK
8      2004-03-05  USA
9      2004-11-15  UK
```

A typical mistake made by people who don't understand logical query processing is attempting to refer in the WHERE clause to a column alias defined in the SELECT clause. This isn't allowed because the WHERE clause is evaluated before the SELECT clause. As an example, consider the following query.

```
SELECT country, YEAR(hiredate) AS yearhired
FROM HR.Employees
WHERE yearhired >= 2003;
```

This query fails with the following error.

```
Msg 207, Level 16, State 1, Line 3
Invalid column name 'yearhired'.
```

If you understand that the WHERE clause is evaluated before the SELECT clause, you realize that this attempt is wrong because at this phase, the attribute yearhired doesn't yet exist. You can indicate the expression YEAR(hiredate) >= 2003 in the WHERE clause. Better yet, for optimization reasons that are discussed in Chapter 3 and Chapter 15, "Implementing Indexes and Statistics," use the form hiredate >= '20030101' as done in the original query.

3. Group Rows Based on the GROUP BY Clause

This phase defines a group for each distinct combination of values in the grouped elements from the input table. It then associates each input row to its respective group. The query you've been working with groups the rows by country and YEAR(hiredate). Within the six rows in the input table, this step identifies four groups. Here are the groups and the detail rows that are associated with them (redundant information removed for purposes of illustration).

```
group    group          detail detail  detail
country  YEAR(hiredate) empid  country hiredate
-------- -------------- ------ ------- ----------
UK       2003           5      UK      2003-10-17
                        6      UK      2003-10-17
UK       2004           7      UK      2004-01-02
                        9      UK      2004-11-15
USA      2003           4      USA     2003-05-03
USA      2004           8      USA     2004-03-05
```

As you can see, the group UK, 2003 has two associated detail rows with employees 5 and 6; the group for UK, 2004 also has two associated detail rows with employees 7 and 9; the group for USA, 2003 has one associated detail row with employee 4; the group for USA, 2004 also has one associated detail row with employee 8.

The final result of this query has one row representing each group (unless filtered out). Therefore, expressions in all phases that take place after the current grouping phase are somewhat limited. All expressions processed in subsequent phases must guarantee a single value per group. If you refer to an element from the GROUP BY list (for example, country), you already have such a guarantee, so such a reference is allowed. However, if you want to refer to an element that is not part of your GROUP BY list (for example, empid), it must be contained within an aggregate function like MAX or SUM. That's because multiple values are possible in the element within a single group, and the only way to guarantee that just one will be returned is to aggregate the values. For more details on grouped queries, see Chapter 5, "Grouping and Windowing."

4. Filter Rows Based on the HAVING Clause

This phase is also responsible for filtering data based on a predicate, but it is evaluated after the data has been grouped; hence, it is evaluated per group and filters groups as a whole. As is usual in T-SQL, the filtering predicate can evaluate to true, false, or unknown. Only groups for which the predicate evaluates to true are returned from this phase. In this case, the HAVING clause uses the predicate COUNT(*) > 1, meaning filter only country and hire year groups that have more than one employee. If you look at the number of rows that were associated with each group in the previous step, you will notice that only the groups UK, 2003 and UK, 2004 qualify. Hence, the result of this phase has the following remaining groups, shown here with their associated detail rows.

```
group    group          detail detail  detail
country  YEAR(hiredate) empid  country hiredate
-------- -------------- ------ ------- ----------
UK       2003           5      UK      2003-10-17
                        6      UK      2003-10-17
UK       2004           7      UK      2004-01-02
                        9      UK      2004-11-15
```

Quick Check

- **What is the difference between the WHERE and HAVING clauses?**

Quick Check Answer

- **The WHERE clause is evaluated before rows are grouped, and therefore is evaluated per row. The HAVING clause is evaluated after rows are grouped, and therefore is evaluated per group.**

5. Process the SELECT Clause

The fifth phase is the one responsible for processing the SELECT clause. What's interesting about it is the point in logical query processing where it gets evaluated—almost last. That's interesting considering the fact that the SELECT clause appears first in the query.

This phase includes two main steps. The first step is evaluating the expressions in the SELECT list and producing the result attributes. This includes assigning attributes with names if they are derived from expressions. Remember that if a query is a grouped query, each group is represented by a single row in the result. In the query, two groups remain after the processing of the HAVING filter. Therefore, this step generates two rows. In this case, the SELECT list returns for each country and hire year group a row with the following attributes: country, YEAR(hiredate) aliased as yearhired, and COUNT(*) aliased as numemployees.

The second step in this phase is applicable if you indicate the DISTINCT clause, in which case this step removes duplicates. Remember that T-SQL is based on multiset theory more than it is on set theory, and therefore, if duplicates are possible in the result, it's your responsibility to remove those with the DISTINCT clause. In this query's case, this step is inapplicable. Here's the result of this phase in the query.

```
country  yearhired  numemployees
-------- ---------- ------------
UK       2003       2
UK       2004       2
```

If you need a reminder of what the query looks like, here it is again.

```
SELECT country, YEAR(hiredate) AS yearhired, COUNT(*) AS numemployees
FROM HR.Employees
WHERE hiredate >= '20030101'
GROUP BY country, YEAR(hiredate)
HAVING COUNT(*) > 1
ORDER BY country , yearhired DESC;
```

The fifth phase returns a relational result. Therefore, the order of the rows isn't guaranteed. In this query's case, there is an ORDER BY clause that guarantees the order in the result, but this will be discussed when the next phase is described. What's important to note is that the outcome of the phase that processes the SELECT clause is still relational.

Also, remember that this phase assigns column aliases, like yearhired and numemployees. This means that newly created column aliases are not visible to clauses processed in previous phases, like FROM, WHERE, GROUP BY, and HAVING.

Note that an alias created by the SELECT phase isn't even visible to other expressions that appear in the same SELECT list. For example, the following query isn't valid.

```
SELECT empid, country, YEAR(hiredate) AS yearhired, yearhired - 1 AS prevyear
FROM HR.Employees;
```

This query generates the following error.

```
Msg 207, Level 16, State 1, Line 1
Invalid column name 'yearhired'.
```

The reason that this isn't allowed is that, conceptually, T-SQL evaluates all expressions that appear in the same logical query processing phase in an all-at-once manner. Note the use of the word *conceptually*. SQL Server won't necessarily physically process all expressions at the same point in time, but it has to produce a result as if it did. This behavior is different than many other programming languages where expressions usually get evaluated in a left-to-right order, making a result produced in one expression visible to the one that appears to its right. But T-SQL is different.

Quick Check

1. **Why are you not allowed to refer to a column alias defined by the SELECT clause in the WHERE clause?**
2. **Why are you not allowed to refer to a column alias defined by the SELECT clause in the same SELECT clause?**

Quick Check Answer

1. **Because the WHERE clause is logically evaluated in a phase earlier to the one that evaluates the SELECT clause.**
2. **Because all expressions that appear in the same logical query processing phase are evaluated conceptually at the same point in time.**

6. Handle Presentation Ordering

The sixth phase is applicable if the query has an ORDER BY clause. This phase is responsible for returning the result in a specific presentation order according to the expressions that appear in the ORDER BY list. The query indicates that the result rows should be ordered first by country (in ascending order by default), and then by yearhired, descending, yielding the following output.

```
country  yearhired  numemployees
-------- ---------- ------------
UK       2004       2
UK       2003       2
```

Notice that the ORDER BY clause is the first and only clause that is allowed to refer to column aliases defined in the SELECT clause. That's because the ORDER BY clause is the only one to be evaluated after the SELECT clause.

Unlike in previous phases where the result was relational, the output of this phase isn't relational because it has a guaranteed order. The result of this phase is what standard SQL calls a cursor. Note that the use of the term cursor here is conceptual. T-SQL also supports an object called a cursor that is defined based on a result of a query, and that allows fetching rows one at a time in a specified order.

You might care about returning the result of a query in a specific order for presentation purposes or if the caller needs to consume the result in that manner through some cursor mechanism that fetches the rows one at a time. But remember that such processing isn't relational. If you need to process the query result in a relational manner—for example, define a table expression like a view based on the query (details later in Chapter 4)—the result will need to be relational. Also, sorting data can add cost to the query processing. If you don't care about the order in which the result rows are returned, you can avoid this unnecessary cost by not adding an ORDER BY clause.

A query may specify the TOP or OFFSET-FETCH filtering options. If it does, the same ORDER BY clause that is normally used to define presentation ordering also defines which rows to filter for these options. It's important to note that such a filter is processed after the SELECT phase evaluates all expressions and removes duplicates (in case a DISTINCT clause was specified). You might even consider the TOP and OFFSET-FETCH filters as being processed in their own phase number 7. The query doesn't indicate such a filter, and therefore, this phase is inapplicable in this case.

PRACTICE Logical Query Processing

In this practice, you exercise your knowledge of logical query processing.

If you encounter a problem completing an exercise, you can install the completed projects from the companion content for this chapter and lesson.

EXERCISE 1 Fix a Problem with Grouping

In this exercise, you are presented with a grouped query that fails when you try to execute it. You are provided with instructions on how to fix the query.

1. Open SSMS and connect to the sample database TSQL2012.
2. Type the following query in the query window and execute it.

```
SELECT custid, orderid
FROM Sales.Orders
GROUP BY custid;
```

The query was supposed to return for each customer the customer ID and the maximum order ID for that customer, but instead it fails. Try to figure out why the query failed and what needs to be revised so that it would return the desired result.

3. The query failed because orderid neither appears in the GROUP BY list nor within an aggregate function. There are multiple possible orderid values per customer. To fix the query, you need to apply an aggregate function to the orderid attribute. The task is to return the maximum orderid value per customer. Therefore, the aggregate function should be MAX. Your query should look like the following.

```
SELECT custid, MAX(orderid) AS maxorderid
FROM Sales.Orders
GROUP BY custid;
```

EXERCISE 2 Fix a Problem with Aliasing

In this exercise, you are presented with another grouped query that fails, this time because of an aliasing problem. As in the first exercise, you are provided with instructions on how to fix the query.

1. Clear the query window, type the following query, and execute it.

```
SELECT shipperid, SUM(freight) AS totalfreight
FROM Sales.Orders
WHERE freight > 20000.00
GROUP BY shipperid;
```

The query was supposed to return only shippers for whom the total freight value is greater than 20,000, but instead it returns an empty set. Try to identify the problem in the query.

2. Remember that the WHERE filtering clause is evaluated per row—not per group. The query filters individual orders with a freight value greater than 20,000, and there are none. To correct the query, you need to apply the filter per each shipper group—not per each order. You need to filter the total of all freight values per shipper. This can be achieved by using the HAVING filter. You try to fix the problem by using the following query.

```
SELECT shipperid, SUM(freight) AS totalfreight
FROM Sales.Orders
GROUP BY shipperid
HAVING totalfreight > 20000.00;
```

But this query also fails. Try to identify why it fails and what needs to be revised to achieve the desired result.

3. The problem now is that the query attempts to refer in the HAVING clause to the alias totalfreight, which is defined in the SELECT clause. The HAVING clause is evaluated before the SELECT clause, and therefore, the column alias isn't visible to it. To fix the problem, you need to refer to the expression SUM(freight) in the HAVING clause, as follows.

```
SELECT shipperid, SUM(freight) AS totalfreight
FROM Sales.Orders
GROUP BY shipperid
HAVING SUM(freight) > 20000.00;
```

Lesson Summary

- T-SQL was designed as a declarative language where the instructions are provided in an English-like manner. Therefore, the keyed-in order of the query clauses starts with the SELECT clause.
- Logical query processing is the conceptual interpretation of the query that defines the correct result, and unlike the keyed-in order of the query clauses, it starts by evaluating the FROM clause.
- Understanding logical query processing is crucial for correct understanding of T-SQL.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. Which of the following correctly represents the logical query processing order of the various query clauses?
   - **A.** SELECT > FROM > WHERE > GROUP BY > HAVING > ORDER BY
   - **B.** FROM > WHERE > GROUP BY > HAVING > SELECT > ORDER BY
   - **C.** FROM > WHERE > GROUP BY > HAVING > ORDER BY > SELECT
   - **D.** SELECT > ORDER BY > FROM > WHERE > GROUP BY > HAVING
2. Which of the following is invalid? (Choose all that apply.)
   - **A.** Referring to an attribute that you group by in the WHERE clause
   - **B.** Referring to an expression in the GROUP BY clause; for example, GROUP BY YEAR(orderdate)
   - **C.** In a grouped query, referring in the SELECT list to an attribute that is not part of the GROUP BY list and not within an aggregate function
   - **D.** Referring to an alias defined in the SELECT clause in the HAVING clause

3. What is true about the result of a query without an ORDER BY clause?
   A. It is relational as long as other relational requirements are met.
   B. It cannot have duplicates.
   C. The order of the rows in the output is guaranteed to be the same as the insertion order.
   D. The order of the rows in the output is guaranteed to be the same as that of the clustered index.

Case Scenarios

In the following case scenarios, you apply what you've learned about T-SQL querying. You can find the answers to these questions in the "Answers" section at the end of this chapter.

Case Scenario 1: Importance of Theory

You and a colleague on your team get into a discussion about the importance of understanding the theoretical foundations of T-SQL. Your colleague argues that there's no point in understanding the foundations, and that it's enough to just learn the technical aspects of T-SQL to be a good developer and to write correct code. Answer the following questions posed to you by your colleague:

1. Can you give an example for an element from set theory that can improve your understanding of T-SQL?
2. Can you explain why understanding the relational model is important for people who write T-SQL code?

Case Scenario 2: Interviewing for a Code Reviewer Position

You are interviewed for a position as a code reviewer to help improve code quality. The organization's application has queries written by untrained people. The queries have numerous problems, including logical bugs. Your interviewer poses a number of questions and asks for a concise answer of a few sentences to each question. Answer the following questions addressed to you by your interviewer:

1. Is it important to use standard code when possible, and why?
2. We have many queries that use ordinal positions in the ORDER BY clause. Is that a bad practice, and if so why?
3. If a query doesn't have an ORDER BY clause, what is the order in which the records are returned?
4. Would you recommend putting a DISTINCT clause in every query?

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks.

Visit T-SQL Public Newsgroups and Review Code

To practice your knowledge of using T-SQL in a relational way, you should review code samples written by others.

- **Practice 1** List as many examples as you can for aspects of T-SQL coding that are not relational.
- **Practice 2** After creating the list in Practice 1, visit the Microsoft public forum for T-SQL at *http://social.msdn.microsoft.com/Forums/en/transactsql/threads*. Review code samples in the T-SQL threads. Try to identify cases where nonrelational elements are used; if you find such cases, identify what needs to be revised to make them relational.

Describe Logical Query Processing

To better understand logical query processing, we recommend that you complete the following tasks:

- **Practice 1** Create a document with a numbered list of the phases involved with logical query processing in the correct order. Provide a brief paragraph summarizing what happens in each step.
- **Practice 2** Create a graphical flow diagram representing the flow of the logical query processing phases by using a tool such as Microsoft Visio, Microsoft PowerPoint, or Microsoft Word.

Answers

This section contains the answers to the lesson review questions and solutions to the case scenarios in this chapter.

Lesson 1

1. **Correct Answers: B and D**
   - **A. Incorrect:** It is important to use standard code.
   - **B. Correct:** Use of standard code makes it easier to port code between platforms because fewer revisions are required.
   - **C. Incorrect:** There's no assurance that standard code will be more efficient.
   - **D. Correct:** When using standard code, you can adapt to a new environment more easily because standard code elements look similar in the different platforms.
2. **Correct Answer: D**
   - **A. Incorrect:** A relation has a header with a set of attributes, and tuples of the relation have the same heading. A set has no order, so ordinal positions do not have meaning and constitute a violation of the relational model. You should refer to attributes by their name.
   - **B. Incorrect:** A query is supposed to return a relation. A relation has a body with a set of tuples. A set has no duplicates. Returning duplicate rows is a violation of the relational model.
   - **C. Incorrect:** Not defining a key in the table allows duplicate rows in the table, and like the answer to B, that's a violation of the relational model.
   - **D. Correct:** Because attributes are supposed to be identified by name, ensuring that all attributes have names is relational, and hence not a violation of the relational model.
3. **Correct Answer: B**
   - **A. Incorrect:** T-SQL isn't standard and SQL isn't a dialect in Microsoft SQL Server.
   - **B. Correct:** SQL is standard and T-SQL is a dialect in Microsoft SQL Server.
   - **C. Incorrect:** T-SQL isn't standard.
   - **D. Incorrect:** SQL isn't a dialect in Microsoft SQL Server.

Lesson 2

1. **Correct Answer: B**

   A. **Incorrect:** Logical query processing doesn't start with the SELECT clause.

   B. **Correct:** Logical query processing starts with the FROM clause, and then moves on to WHERE, GROUP BY, HAVING, SELECT, and ORDER BY.

   C. **Incorrect:** The ORDER BY clause isn't evaluated before the SELECT clause.

   D. **Incorrect:** Logical query processing doesn't start with the SELECT clause.

2. **Correct Answer: C and D**

   A. **Incorrect:** T-SQL allows you to refer to an attribute that you group by in the WHERE clause.

   B. **Incorrect:** T-SQL allows grouping by an expression.

   C. **Correct:** If the query is a grouped query, in phases processed after the GROUP BY phase, each attribute that you refer to must appear either in the GROUP BY list or within an aggregate function.

   D. **Correct:** Because the HAVING clause is evaluated before the SELECT clause, referring to an alias defined in the SELECT clause within the HAVING clause is invalid.

3. **Correct Answer: A**

   A. **Correct:** A query with an ORDER BY clause doesn't return a relational result. For the result to be relational, the query must satisfy a number of requirements, including the following: the query must not have an ORDER BY clause, all attributes must have names, all attribute names must be unique, and duplicates must not appear in the result.

   B. **Incorrect:** A query without a DISTINCT clause in the SELECT clause can return duplicates.

   C. **Incorrect:** A query without an ORDER BY clause does not guarantee the order of rows in the output.

   D. **Incorrect:** A query without an ORDER BY clause does not guarantee the order of rows in the output.

Case Scenario 1

1. One of the most typical mistakes that T-SQL developers make is to assume that a query without an ORDER BY clause always returns the data in a certain order—for example, clustered index order. But if you understand that in set theory, a set has no particular order to its elements, you know that you shouldn't make such assumptions. The only way in SQL to guarantee that the rows will be returned in a certain order is to add an ORDER BY clause. That's just one of many examples for aspects of T-SQL that can be better understood if you understand the foundations of the language.
2. Even though T-SQL is based on the relational model, it deviates from it in a number of ways. But it gives you enough tools that if you understand the relational model, you can write in a relational way. Following the relational model helps you write code more correctly. Here are some examples:
   - You shouldn't rely on order of columns or rows.
   - You should always name result columns.
   - You should eliminate duplicates if they are possible in the result of your query.

Case Scenario 2

1. It is important to use standard SQL code. This way, both the code and people's knowledge is more portable. Especially in cases where there are both standard and nonstandard forms for a language element, it's recommended to use the standard form.
2. Using ordinal positions in the ORDER BY clause is a bad practice. From a relational perspective, you are supposed to refer to attributes by name, and not by ordinal position. Also, what if the SELECT list is revised in the future and the developer forgets to revise the ORDER BY list accordingly?
3. When the query doesn't have an ORDER BY clause, there are no assurances for any particular order in the result. The order should be considered arbitrary. You also notice that the interviewer used the incorrect term *record* instead of *row*. You might want to mention something about this, because the interviewer may have done so on purpose to test you.
4. From a pure relational perspective, this actually could be valid, and perhaps even recommended. But from a practical perspective, there is the chance that SQL Server will try to remove duplicates even when there are none, and this will incur extra cost. Therefore, it is recommended that you add the DISTINCT clause only when duplicates are possible in the result and you're not supposed to return the duplicates.

CHAPTER 7

Querying and Managing XML Data

Exam objectives in this chapter:

- Work with Data
  - Query and manage XML data.

Microsoft SQL Server 2012 includes extensive support for XML. This support includes creating XML from relational data with a query and shredding XML into relational tabular format. Additionally, SQL Server has a native XML data type. You can store XML data, constrain it with XML schemas, index it with specialized XML indexes, and manipulate it using XML data type methods. All of the T-SQL XML data type methods accept an XQuery string as a parameter. XQuery (short for XML Query Language) is the standard language used to query and manipulate XML data.

In this chapter, you learn how to use all of the XML features mentioned. In addition, you get a couple of ideas about why you would use XML in a relational database.

> ***IMPORTANT*** **USE OF THE TERM XML IN THIS CHAPTER**
>
> **XML is used in this chapter to refer to both the open standard and T-SQL data type.**

Lessons in this chapter:

- Lesson 1: Returning Results As XML with FOR XML
- Lesson 2: Querying XML Data with XQuery
- Lesson 3: Using the XML Data Type

Before You Begin

To complete the lessons in this chapter, you must have:

- An understanding of relational database concepts.
- Experience working with SQL Server Management Studio (SSMS).
- Some experience writing T-SQL code.
- Access to a SQL Server 2012 instance with the sample database TSQL2012 installed.

Lesson 1: Returning Results As XML with FOR XML

XML is a widely used standard for data exchange, calling web services methods, configuration files, and more. This lesson starts with a short introduction to XML. After that, you learn how you can create XML as the result of a query by using different flavors of the FOR XML clause. The lesson finishes with information on shredding XML to relational tables by using the OPENXML rowset function.

After this lesson, you will be able to:

- Describe XML documents.
- Convert relational data to XML.
- Shred XML to tables.

Estimated lesson time: 40 minutes

Introduction to XML

This lesson introduces XML through samples. The following is an example of an XML document, created with a FOR XML clause of the SELECT statement.

```
<CustomersOrders>
  <Customer custid="1" companyname="Customer NRZBB">
    <Order orderid="10692" orderdate="2007-10-03T00:00:00" />
    <Order orderid="10702" orderdate="2007-10-13T00:00:00" />
    <Order orderid="10952" orderdate="2008-03-16T00:00:00" />
  </Customer>
  <Customer custid="2" companyname="Customer MLTDN">
    <Order orderid="10308" orderdate="2006-09-18T00:00:00" />
    <Order orderid="10926" orderdate="2008-03-04T00:00:00" />
  </Customer>
</CustomersOrders>
```

NOTE **COMPANION CODE**

The query that produces the XML output from the previous example and other queries for other examples are provided in the companion code files.

As you can see, XML uses tags to name parts of an *XML document*. These parts are called *elements*. Every begin *tag*, such as <Customer>, must have a corresponding end tag, in this case </Customer>. If an element has no nested elements, the notation can be abbreviated to a single tag that denotes the beginning and end of an element, such as <Order … />. Elements can be nested. Tags cannot be interleaved; the end tag of a parent element must be after the end tag of the last nested element. If every begin tag has a corresponding end tag, and if tags are nested properly, the XML document is *well-formed*.

XML documents are *ordered*. This does not mean they are ordered by any specific element value; it means that the position of elements matters. For example, the element with orderid equal to 10702 in the preceding example is the second Order element under the first Customer element.

XML is *case-sensitive Unicode text*. You should never forget that XML is case sensitive. In addition, some characters in XML, such as <, which introduces a tag, are processed as *markup* and have special meanings. If you want to include these characters in the values of your XML document, they must be escaped by using an ampersand (&), followed by a special code, followed by a semicolon (;), as shown in Table 7-1.

TABLE 7-1 Characters with special values in XML documents

Character	Replacement text
& (ampersand)	&
" (quotation mark)	"
< (less than)	<
> (greater than)	>
' (apostrophe)	'

Alternatively, you can use the special XML CDATA section, written as <![CDATA[...]]>. You can replace the three dots with any character string that does not include the string literal "]]>"; this will prevent special characters in the string from being parsed as XML markup.

Processing instructions, which are information for applications that process XML, are written similarly to elements, between less than (<) and greater than (>) characters, and they start and end with a question mark (?), like <?PItarget data?>. The engine that processes XML—for example, the SQL Server Database Engine — receives those instructions.

In addition to elements and processing instructions, XML can include comments in the format <!-- This is a comment -->.

Finally, XML can have a prolog at the beginning of the document, denoting the XML version and encoding of the document, such as <?xml version="1.0" encoding="ISO-8859-15"?>.

In addition to XML documents, you can also have *XML fragments*. The only difference between a document and a fragment is that a document has a single *root node*, like <CustomersOrders> in the preceding example. If you delete this node, you get the following XML fragment.

```
<Customer custid="1" companyname="Customer NRZBB">
  <Order orderid="10692" orderdate="2007-10-03T00:00:00" />
  <Order orderid="10702" orderdate="2007-10-13T00:00:00" />
  <Order orderid="10952" orderdate="2008-03-16T00:00:00" />
</Customer>
```

```
  <Customer custid="2" companyname="Customer MLTDN">
    <Order orderid="10308" orderdate="2006-09-18T00:00:00" />
    <Order orderid="10926" orderdate="2008-03-04T00:00:00" />
  </Customer>
```

If you delete the second customer, you get an XML document because it will have a single root node again.

As you can see from the examples so far, elements can have *attributes*. Attributes have their own names, and their values are enclosed in quotation marks. This is *attribute-centric* presentation. However, you can write XML differently; every attribute can be a nested element of the original element. This is *element-centric* presentation.

Finally, element names do not have to be unique, because they can be referred to by their position; however, to distinguish between elements from different business areas, different departments, or different companies, you can add *namespaces*. You declare namespaces used in the root element of an XML document. You can also use an *alias* for every single namespace. Then you prefix element names with a namespace alias. The following code is an example of element-centric XML that uses a namespace; the data is the same as in the first example of this lesson.

```
<CustomersOrders xmlns:co="TK461-CustomersOrders">
  <co:Customer>
    <co:custid>1</co:custid>
    <co:companyname>Customer NRZBB</co:companyname>
    <co:Order>
      <co:orderid>10692</co:orderid>
      <co:orderdate>2007-10-03T00:00:00</co:orderdate>
    </co:Order>
    <co:Order>
      <co:orderid>10702</co:orderid>
      <co:orderdate>2007-10-13T00:00:00</co:orderdate>
    </co:Order>
    <co:Order>
      <co:orderid>10952</co:orderid>
      <co:orderdate>2008-03-16T00:00:00</co:orderdate>
    </co:Order>
  </co:Customer>
  <co:Customer>
    <co:custid>2</co:custid>
    <co:companyname>Customer MLTDN</co:companyname>
    <co:Order>
      <co:orderid>10308</co:orderid>
      <co:orderdate>2006-09-18T00:00:00</co:orderdate>
    </co:Order>
    <co:Order>
      <co:orderid>10926</co:orderid>
      <co:orderdate>2008-03-04T00:00:00</co:orderdate>
    </co:Order>
  </co:Customer>
</CustomersOrders>
```

XML is very flexible. As you've seen so far, there are very few rules for creating a well-formed XML document. In an XML document, the actual data is mixed with *metadata*, such as

element and attribute names. Because XML is text, it is very convenient for exchanging data between different systems and even between different platforms. However, when exchanging data, it becomes important to have metadata fixed. If you had to import a document with customers' orders, as in the preceding examples, every couple of minutes, you'd definitely want to automate the import process. Imagine how hard you'd have to work if the metadata changed with every new import. For example, imagine that the Customer element gets renamed to Client, and the Order element gets renamed to Purchase. Or imagine that the orderdate attribute (or element) suddenly changes its data type from timestamp to integer. You'd quickly conclude that you should have more fixed *schema* for the XML documents you are importing.

Many different standards have evolved to describe the metadata of XML documents. Currently, the most widely used metadata description is with *XML Schema Description* (XSD) documents. XSD documents are XML documents that describe the metadata of other XML documents. The schema of an XSD document is predefined. With the XSD standard, you can specify element names, data types, and number of occurrences of an element, constraints, and more. The following example shows an XSD schema describing the element-centric version of customers and their orders.

```
<xsd:schema targetNamespace="TK461-CustomersOrders" xmlns:schema="TK461-CustomersOrders"
  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 name="custid" type="sqltypes:int" />
        <xsd:element name="companyname">
          <xsd:simpleType>
            <xsd:restriction base="sqltypes:nvarchar" sqltypes:localeId="1033"
               sqltypes:sqlCompareOptions="IgnoreCase IgnoreKanaType IgnoreWidth"
               sqltypes:sqlSortId="52">
              <xsd:maxLength value="40" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element ref="schema:Order" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="orderid" type="sqltypes:int" />
        <xsd:element name="orderdate" type="sqltypes:datetime" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

When you check whether an XML document complies with a schema, you *validate* the document. A document with a predefined schema is said to be a *typed* XML document.

Producing XML from Relational Data

With the T-SQL SELECT statement, you can create all XML shown in this lesson. This section explains how you can convert a query result set to XML by using the FOR XML clause of the SELECT T-SQL statement. You learn about the most useful options and directives of this clause; for a detailed description of the complete syntax, see the Books Online for SQL Server 2012 article “FOR XML (SQL Server)” at *http://msdn.microsoft.com/en-us/library/ms178107.aspx*.

FOR XML RAW

The first option for creating XML from a query result is the RAW option. The XML created is quite close to the relational (tabular) presentation of the data. In RAW mode, every row from returned rowsets converts to a single element named row, and columns translate to the attributes of this element. Here is an example of an XML document created with the FOR XML RAW option.

```
<row custid="1" companyname="Customer NRZBB" orderid="10692"
 orderdate="2007-10-03T00:00:00" />
<row custid="1" companyname="Customer NRZBB" orderid="10702"
 orderdate="2007-10-13T00:00:00" />
<row custid="1" companyname="Customer NRZBB" orderid="10952"
 orderdate="2008-03-16T00:00:00" />
<row custid="2" companyname="Customer MLTDN" orderid="10308"
 orderdate="2006-09-18T00:00:00" />
<row custid="2" companyname="Customer MLTDN" orderid="10926"
 orderdate="2008-03-04T00:00:00" />
```

You can enhance the RAW mode by renaming the row element, adding a root element, including namespaces, and making the XML returned element-centric. The following is an example of enhanced XML created with the FOR XML RAW option.

```
<CustomersOrders>
  <Order custid="1" companyname="Customer NRZBB" orderid="10692"
   orderdate="2007-10-03T00:00:00" />
  <Order custid="1" companyname="Customer NRZBB" orderid="10702"
   orderdate="2007-10-13T00:00:00" />
  <Order custid="1" companyname="Customer NRZBB" orderid="10952"
   orderdate="2008-03-16T00:00:00" />
  <Order custid="2" companyname="Customer MLTDN" orderid="10308"
   orderdate="2006-09-18T00:00:00" />
  <Order custid="2" companyname="Customer MLTDN" orderid="10926"
   orderdate="2008-03-04T00:00:00" />
</CustomersOrders>
```

As you can see, this is a document instead of a fragment. It looks more like “real” XML; however, it does not include any additional level of nesting. The customer with custid equal to 1 is repeated three times, once for each order; it would be nicer if it appeared once only and included orders as nested elements. You can produce XML that is easier to read with the FOR XML AUTO option, described in the following section.

FOR XML AUTO

The FOR XML AUTO option gives you nice XML documents with nested elements, and it is not complicated to use. In AUTO and RAW modes, you can use the keyword ELEMENTS to produce element-centric XML. The WITH NAMESPACES clause, preceding the SELECT part of the query, defines namespaces and aliases in the returned XML. So far, you have seen XML results only. In the practice for this lesson, you create queries that produce similar results. However, in order to give you a better presentation of how SELECT with the FOR XML clause looks, here is an example.

```
WITH XMLNAMESPACES('TK461-CustomersOrders' AS co)
SELECT [co:Customer].custid AS [co:custid],
 [co:Customer].companyname AS [co:companyname],
 [co:Order].orderid AS [co:orderid],
 [co:Order].orderdate AS [co:orderdate]
FROM Sales.Customers AS [co:Customer]
 INNER JOIN Sales.Orders AS [co:Order]
  ON [co:Customer].custid = [co:Order].custid
WHERE [co:Customer].custid <= 2
  AND [co:Order].orderid %2 = 0
ORDER BY [co:Customer].custid, [co:Order].orderid
FOR XML AUTO, ELEMENTS, ROOT('CustomersOrders');
```

The T-SQL table and column aliases in the query are used to produce element names, prefixed with a namespace. A colon is used in XML to separate the namespace from the element name. The WHERE clause of the query limits the output to two customers, with only even orders for each customer retrieved. The output is a quite nice element-centric XML document.

```
<CustomersOrders xmlns:co="TK461-CustomersOrders">
  <co:Customer>
    <co:custid>1</co:custid>
    <co:companyname>Customer NRZBB</co:companyname>
    <co:Order>
      <co:orderid>10692</co:orderid>
      <co:orderdate>2007-10-03T00:00:00</co:orderdate>
    </co:Order>
    <co:Order>
      <co:orderid>10702</co:orderid>
      <co:orderdate>2007-10-13T00:00:00</co:orderdate>
    </co:Order>
    <co:Order>
      <co:orderid>10952</co:orderid>
      <co:orderdate>2008-03-16T00:00:00</co:orderdate>
    </co:Order>
  </co:Customer>
  <co:Customer>
```

```
  <co:Customer>
    <co:custid>2</co:custid>
    <co:companyname>Customer MLTDN</co:companyname>
    <co:Order>
      <co:orderid>10308</co:orderid>
      <co:orderdate>2006-09-18T00:00:00</co:orderdate>
    </co:Order>
    <co:Order>
      <co:orderid>10926</co:orderid>
      <co:orderdate>2008-03-04T00:00:00</co:orderdate>
    </co:Order>
  </co:Customer>
</CustomersOrders>
```

Note that a proper ORDER BY clause is very important. With T-SQL SELECT, you are actually formatting the returned XML. Without the ORDER BY clause, the order of rows returned is unpredictable, and you can get a weird XML document with an element repeated multiple times with just part of nested elements every time.

EXAM TIP

The FOR XML clause comes after the ORDER BY clause in a query.

It is not only the ORDER BY clause that is important; the order of columns in the SELECT clause also influences the XML returned. SQL Server uses column order to determine the nesting of elements. The order of the columns should follow one-to-many relationships. A customer can have many orders; therefore, you should have customer columns before order columns in your query.

You might be vexed by the fact that you have to take care of column order; in a relation, the order of columns and rows is not important. Nevertheless, you have to realize that the result of your query is not a relation; it is text in XML format, and parts of your query are used for formatting the text.

In RAW and AUTO mode, you can also return the *XSD schema* of the document you are creating. This schema is included inside the XML that is returned, before the actual XML data; therefore, it is called *inline* schema. You return XSD with the XMLSCHEMA directive. This directive accepts a parameter that defines a target namespace. If you need schema only, without data, simply include a WHERE condition in your query with a predicate that no row can satisfy. The following query returns the schema of the XML generated in the previous query.

```
SELECT [Customer].custid AS [custid],
 [Customer].companyname AS [companyname],
 [Order].orderid AS [orderid],
 [Order].orderdate AS [orderdate]
FROM Sales.Customers AS [Customer]
 INNER JOIN Sales.Orders AS [Order]
  ON [Customer].custid = [Order].custid
WHERE 1 = 2
FOR XML AUTO, ELEMENTS,
    XMLSCHEMA('TK461-CustomersOrders');
```

Here is the output, the XSD document.

```
<xsd:schema targetNamespace="TK461-CustomersOrders" xmlns:schema="TK461-CustomersOrders"
  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 name="custid" type="sqltypes:int" />
        <xsd:element name="companyname">
          <xsd:simpleType>
            <xsd:restriction base="sqltypes:nvarchar" sqltypes:localeId="1033"
              sqltypes:sqlCompareOptions="IgnoreCase IgnoreKanaType IgnoreWidth"
              sqltypes:sqlSortId="52">
              <xsd:maxLength value="40" />
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element ref="schema:Order" minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="orderid" type="sqltypes:int" />
        <xsd:element name="orderdate" type="sqltypes:datetime" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

FOR XML PATH

With the last two flavors of the FOR XML clause —the EXPLICIT and PATH options—you can manually define the XML returned. With these two options, you have total control of the XML document returned. The EXPLICIT mode is included for backward compatibility only; it uses proprietary T-SQL syntax for formatting XML. The PATH mode uses standard XML XPath expressions to define the elements and attributes of the XML you are creating. This section focuses on the PATH mode; if you want to learn more about the EXPLICIT mode, see the Books Online for SQL Server 2012 article "Use EXPLICIT Mode with FOR XML" at *http://msdn.microsoft.com/en-us/library/ms189068.aspx*.

In PATH mode, column names and aliases serve as XPath expressions. XPath expressions define the path to the element in the XML generated. Path is expressed in a hierarchical way; levels are delimited with the slash (/) character. By default, every column becomes an element; if you want to generate attribute-centric XML, prefix the alias name with the "at" (@) character.

Here is an example of a simple XPATH query.

```
SELECT Customer.custid AS [@custid],
 Customer.companyname AS [companyname]
FROM Sales.Customers AS Customer
WHERE Customer.custid <= 2
ORDER BY Customer.custid
FOR XML PATH ('Customer'), ROOT('Customers');
```

The query returns the following output.

```
<Customers>
  <Customer custid="1">
    <companyname>Customer NRZBB</companyname>
  </Customer>
  <Customer custid="2">
    <companyname>Customer MLTDN</companyname>
  </Customer>
</Customers>
```

If you want to create XML with nested elements for child tables, you have to use subqueries in the SELECT part of the query in the PATH mode. Subqueries have to return a scalar value in a SELECT clause. However, you know that a parent row can have multiple child rows; a customer can have multiple orders. You return a scalar value by returning XML from the subquery. Then the result is returned as a single scalar XML value. You format nested XML from the subquery with the FOR XML clause, like you format XML in an outer query. Additionally, you have to use the TYPE directive of the FOR XML clause to produce a value of the XML data type, and not XML as text, which cannot be consumed by the outer query.

You create XML with nested elements by using the FOR XML PATH clause in the practice for this lesson.

Quick Check

- **How can you get an XSD schema together with an XML document from your SELECT statement?**

Quick Check Answer

- **You should use the XMLSCHEMA directive in the FOR XML clause.**

Shredding XML to Tables

You just learned how to create XML from relational data. Of course, you can also do the opposite process: convert XML to tables. Converting XML to relational tables is known as *shredding* XML. You can do this by using the nodes method of the XML data type; you learn about this method in Lesson 3, "Using the XML Data Type." Starting with SQL Server 2000, you can do the shredding also with the OPENXML rowset function.

The OPENXML function provides a rowset over in-memory XML documents by using *Document Object Model* (DOM) presentation. Before parsing the DOM, you need to prepare it. To prepare the DOM presentation of XML, you need to call the system stored procedure sys.sp_xml_preparedocument. After you shred the document, you must remove the DOM presentation by using the system procedure sys.sp_xml_removedocument.

The OPENXML function uses the following parameters:

- An XML DOM document handle, returned by sp_xml_preparedocument
- An XPath expression to find the nodes you want to map to rows of a rowset returned
- A description of the rowset returned
- Mapping between XML nodes and rowset columns

The document handle is an integer. This is the simplest parameter. The XPath expression is specified as rowpattern, which defines how XML nodes translate to rows. The path to a node is used as a pattern; nodes below the selected node define rows of the returned rowset.

You can map XML elements or attributes to rows and columns by using the WITH clause of the OPENXML function. In this clause, you can specify an existing table, which is used as a template for the rowset returned, or you can define a table with syntax similar to that in the CREATE TABLE T-SQL statement.

The OPENXML function accepts an optional third parameter, called flags, which allows you to specify the mapping used between the XML data and the relational rowset. A value of 1 means attribute-centric mapping, 2 means element-centric, and 3 means both. However, flag value 3 is undocumented, and it is a best practice not to use it. Flag value 8 can be combined with values 1 and 2 with a bitwise logical OR operator to get both attribute and element-centric mapping. The XML used for the following OPENXML examples uses attributes and elements; for example, custid is the attribute and companyname is the element. The intention of this slightly overcomplicated XML is to show you the difference between attribute-centric and element-centric mappings. The following code shreds the same XML three times to show you the difference between different mappings by using the following values for the flags parameter: 1, 2, and 11 (8+1+2); all three queries use the same rowset description in the WITH clause.

```
DECLARE @DocHandle AS INT;
DECLARE @XmlDocument AS NVARCHAR(1000);
SET @XmlDocument = N'
<CustomersOrders>
  <Customer custid="1">
    <companyname>Customer NRZBB</companyname>
    <Order orderid="10692">
      <orderdate>2007-10-03T00:00:00</orderdate>
    </Order>
    <Order orderid="10702">
      <orderdate>2007-10-13T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-16T00:00:00</orderdate>
    </Order>
  </Customer>
```

```
  <Customer custid="2">
    <companyname>Customer MLTDN</companyname>
    <Order orderid="10308">
      <orderdate>2006-09-18T00:00:00</orderdate>
    </Order>
    <Order orderid="10926">
      <orderdate>2008-03-04T00:00:00</orderdate>
    </Order>
  </Customer>
</CustomersOrders>';
-- Create an internal representation
EXEC sys.sp_xml_preparedocument @DocHandle OUTPUT, @XmlDocument;
-- Attribute-centric mapping
SELECT *
FROM OPENXML (@DocHandle, '/CustomersOrders/Customer',1)
     WITH (custid INT,
           companyname NVARCHAR(40));
-- Element-centric mapping
SELECT *
FROM OPENXML (@DocHandle, '/CustomersOrders/Customer',2)
     WITH (custid INT,
           companyname NVARCHAR(40));
-- Attribute- and element-centric mapping
-- Combining flag 8 with flags 1 and 2
SELECT *
FROM OPENXML (@DocHandle, '/CustomersOrders/Customer',11)
     WITH (custid INT,
           companyname NVARCHAR(40));
-- Remove the DOM
EXEC sys.sp_xml_removedocument @DocHandle;
GO
```

Results of the preceding three queries are as follows.

```
custid      companyname
----------- ----------------------------------------
1           NULL
2           NULL
custid      companyname
----------- ----------------------------------------
NULL        Customer NRZBB
NULL        Customer MLTDN
custid      companyname
----------- ----------------------------------------
1           Customer NRZBB
2           Customer MLTDN
```

As you can see, you get attributes with attribute-centric mapping, elements with element-centric mapping, and both if you combine the two mappings. The nodes method of the XML data type is more efficient for shredding an XML document only once and is therefore the preferred way of shredding XML documents in such a case. However, if you need to shred the same document multiple times, like shown in the three-query example for the OPENXML function, then preparing the DOM presentation once, using OPENXML multiple times, and removing the DOM presentation might be faster.

PRACTICE Using the FOR XML Clause

In this practice, you create XML from relational data. You return XML data as a document and as a fragment.

If you encounter a problem completing an exercise, you can install the completed projects from the companion content for this chapter and lesson.

EXERCISE 1 Return an XML Document

In this exercise, you return XML formatted as a document from relational data.

1. Start SSMS and connect to your SQL Server instance.
2. Open a new query window by clicking the New Query button.
3. Change the current database context to the TSQL2012 database.
4. Return customers with their orders as XML in RAW mode. Return the custid and companyname columns from the Sales.Customers table, and orderid and orderdate columns from the Sales.Orders table. You can use the following query.

```
SELECT Customer.custid, Customer.companyname,
 [Order].orderid, [Order].orderdate
FROM Sales.Customers AS Customer
 INNER JOIN Sales.Orders AS [Order]
  ON Customer.custid = [Order].custid
ORDER BY Customer.custid, [Order].orderid
FOR XML RAW;
```

5. Observe the results.
6. Improve the XML created with the previous query by changing from RAW to AUTO mode. Make the result element-centric by using TK461-CustomersOrders as the namespace and CustomersOrders as the root element. You can use the following code.

```
WITH XMLNAMESPACES('TK461-CustomersOrders' AS co)
SELECT [co:Customer].custid AS [co:custid],
 [co:Customer].companyname AS [co:companyname],
 [co:Order].orderid AS [co:orderid],
 [co:Order].orderdate AS [co:orderdate]
FROM Sales.Customers AS [co:Customer]
 INNER JOIN Sales.Orders AS [co:Order]
  ON [co:Customer].custid = [co:Order].custid
ORDER BY [co:Customer].custid, [co:Order].orderid
FOR XML AUTO, ELEMENTS, ROOT('CustomersOrders');
```

7. Observe the results.

EXERCISE 2 **Return an XML Fragment**

In this exercise, you return XML formatted as a fragment from relational data.

1. Return the third XML as a fragment, not as a document. Return the top element Customer with custid and companyname attributes. Return the Order nested element with orderid and orderdate attributes. Use the FOR XML PATH clause for explicit formatting of XML. You can use the following code.

```
SELECT Customer.custid AS [@custid],
 Customer.companyname AS [@companyname],
 (SELECT [Order].orderid AS [@orderid],
   [Order].orderdate AS [@orderdate]
  FROM Sales.Orders AS [Order]
  WHERE Customer.custid = [Order].custid
    AND [Order].orderid %2 = 0
  ORDER BY [Order].orderid
  FOR XML PATH('Order'), TYPE)
FROM Sales.Customers AS Customer
WHERE Customer.custid <= 2
ORDER BY Customer.custid
FOR XML PATH('Customer');
```

2. Observe the results.

Lesson Summary

- You can use the FOR XML clause of the SELECT T-SQL statement to produce XML.
- Use the OPENXML function to shred XML to tables.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. Which FOR XML options are valid? (Choose all that apply.)
   - **A.** FOR XML AUTO
   - **B.** FOR XML MANUAL
   - **C.** FOR XML DOCUMENT
   - **D.** FOR XML PATH
2. Which directive of the FOR XML clause should you use to produce element-centric XML?
   - **A.** ATTRIBUTES
   - **B.** ROOT
   - **C.** ELEMENTS
   - **D.** XMLSCHEMA

3. Which FOR XML options can you use to manually format the XML returned? (Choose all that apply.)
   - **A.** FOR XML AUTO
   - **B.** FOR XML EXPLICIT
   - **C.** FOR XML RAW
   - **D.** FOR XML PATH

Lesson 2: Querying XML Data with XQuery

XQuery is a standard language for browsing XML instances and returning XML. It is much richer than *XPath* expressions, an older standard, which you can use for simple navigation only. With XQuery, you can navigate as with XPath; however, you can also loop over nodes, shape the returned XML instance, and much more.

For a query language, you need a query-processing engine. The SQL Server database engine processes XQuery inside T-SQL statements through XML data type methods. Not all XQuery features are supported in SQL Server. For example, XQuery user-defined functions are not supported in SQL Server because you already have T-SQL and CLR functions available. Additionally, T-SQL supports nonstandard extensions to XQuery, called *XML DML*, that you can use to modify elements and attributes in XML data. Because an XML data type is a large object, it could be a huge performance bottleneck if the only way to modify an XML value were to replace the entire value.

This lesson introduces XQuery for data retrieval purposes only; you learn more about the XML data type in Lesson 3. In this lesson, you use variables of the XML data type and the query method of the XML data type only. The query method accepts an XQuery string as its parameter, and it returns the XML you shape in XQuery.

The implementation of XQuery in SQL Server follows the World Wide Web Consortium (W3C) standard, and it is supplemented with extensions to support data modifications. You can find more about W3C on the web at *http://www.w3.org/*, and news and additional resources about XQuery at *http://www.w3.org/XML/Query/*.

After this lesson, you will be able to:

- Use XPath expressions to navigate through nodes of an XML instance.
- Use XQuery predicates.
- Use XQuery FLWOR expressions.

Estimated lesson time: 60 minutes

XQuery Basics

XQuery is, like XML, case sensitive. Therefore, if you want to check the examples manually, you have to write the queries exactly as they are written in this chapter. For example, if you write Data() instead of data(), you will get an error stating that there is no Data() function.

XQuery returns sequences. Sequences can include *atomic* values or *complex* values (XML nodes). Any node, such as an element, attribute, text, processing instruction, comment, or document, can be included in the sequence. Of course, you can format the sequences to get well-formed XML. The following code shows different sequences returned from a simple XML instance by three XML queries.

```
DECLARE @x AS XML;
SET @x=N'
<root>
 <a>1<c>3</c><d>4</d></a>
 <b>2</b>
</root>';
SELECT
 @x.query('*') AS Complete_Sequence,
 @x.query('data(*)') AS Complete_Data,
 @x.query('data(root/a/c)') AS Element_c_Data;
```

Here are the sequences returned.

```
Complete_Sequence                             Complete_Data Element_c_Data
--------------------------------------------- ------------- --------------
<root><a>1<c>3</c><d>4</d></a><b>2</b></root>          1342              3
```

The first XQuery expression uses the simplest possible path expression, which selects everything from the XML instance; the second uses the data() function to extract all atomic data values from the complete document; the third uses the data() function to extract atomic data from the element c only.

Every identifier in XQuery is a qualified name, or a *QName*. A QName consists of a local name and, optionally, a namespace prefix. In the preceding example, root, a, b, c, and d are QNames; however, they are without namespace prefixes. The following standard namespaces are predefined in SQL Server:

- **xs** The namespace for an XML schema (the uniform resource identifier, or URI, is *http://www.w3.org/2001/XMLSchema*)
- **xsi** The XML schema instance namespace, used to associate XML schemas with instance documents (*http://www.w3.org/2001/XMLSchema-instance*)
- **xdt** The namespace for XPath and XQuery data types (*http://www.w3.org/2004/07/xpath-datatypes*)
- **fn** The functions namespace (*http://www.w3.org/2004/07/xpath-functions*)
- **sqltypes** The namespace that provides mapping for SQL Server data types (*http://schemas.microsoft.com/sqlserver/2004/sqltypes*)
- **xml** The default XML namespace (*http://www.w3.org/XML/1998/namespace*)

You can use these namespaces in your queries without defining them again. You define your own data types in the *prolog*, which belongs at the beginning of your XQuery. You separate the prolog from the query body with a semicolon. In addition, in T-SQL, you can declare namespaces used in XQuery expressions in advance in the WITH clause of the T-SQL SELECT command. If your XML uses a single namespace, you can also declare it as the default namespace for all elements in the XQuery prolog.

You can also include comments in your XQuery expressions. The syntax for a comment is text between parentheses and colons: (: this is a comment :). Do not mix this with comment nodes in your XML document; this is the comment of your XQuery and has no influence on the XML returned. The following code shows all three methods of namespace declaration and uses XQuery comments. It extracts orders for the first customer from an XML instance.

```
DECLARE @x AS XML;
SET @x='
<CustomersOrders xmlns:co="TK461-CustomersOrders">
  <co:Customer co:custid="1" co:companyname="Customer NRZBB">
    <co:Order co:orderid="10692" co:orderdate="2007-10-03T00:00:00" />
    <co:Order co:orderid="10702" co:orderdate="2007-10-13T00:00:00" />
    <co:Order co:orderid="10952" co:orderdate="2008-03-16T00:00:00" />
  </co:Customer>
  <co:Customer co:custid="2" co:companyname="Customer MLTDN">
    <co:Order co:orderid="10308" co:orderdate="2006-09-18T00:00:00" />
    <co:Order co:orderid="10926" co:orderdate="2008-03-04T00:00:00" />
  </co:Customer>
</CustomersOrders>';
-- Namespace in prolog of XQuery
SELECT @x.query('
(: explicit namespace :)
declare namespace co="TK461-CustomersOrders";
//co:Customer[1]/*') AS [Explicit namespace];
-- Default namespace for all elements in prolog of XQuery
SELECT @x.query('
(: default namespace :)
declare default element namespace "TK461-CustomersOrders";
//Customer[1]/*') AS [Default element namespace];
-- Namespace defined in WITH clause of T-SQL SELECT
WITH XMLNAMESPACES('TK461-CustomersOrders' AS co)
SELECT @x.query('
(: namespace declared in T-SQL :)
//co:Customer[1]/*') AS [Namespace in WITH clause];
```

Here is the abbreviated output.

```
Explicit namespace
--------------------------------------------------------------------------------
<co:Order xmlns:co="TK461-CustomersOrders" co:orderid="10692" co:orderd

Default element namespace
--------------------------------------------------------------------------------
<Order xmlns="TK461-CustomersOrders" xmlns:p1="TK461-Customers

Namespace in WITH clause
--------------------------------------------------------------------------------
<co:Order xmlns:co="TK461-CustomersOrders" co:orderid="10692" co:orderd
```

NOTE THE DEFAULT NAMESPACE

If you use a default element namespace, the namespace is not included for the elements in the resulting XML; it is included for the attributes. Therefore, only the first and third queries are completely equivalent. In addition, when you use the default element namespace, you can't define your own namespace abbreviation. You should prefer an explicit namespace definition to using the default element namespace.

The queries used a relative path to find the Customer element. Before looking at all the different ways of navigation in XQuery, you should first read through the most important XQuery data types and functions, described in the following two sections.

XQuery Data Types

XQuery uses about 50 predefined data types. Additionally, in the SQL Server implementation you also have the sqltypes namespace, which defines SQL Server types. You already know about SQL Server types. Do not worry too much about XQuery types; you'll never use most of them. This section lists only the most important ones, without going into details about them.

XQuery data types are divided into node types and atomic types. The node types include attribute, comment, element, namespace, text, processing-instruction, and document-node. The most important atomic types you might use in queries are xs:boolean, xs:string, xs:QName, xs:date, xs:time, xs:datetime, xs:float, xs:double, xs:decimal, and xs:integer.

You should just do a quick review of this much-shortened list. The important thing to understand is that XQuery has its own type system, that it has all of the commonly used types you would expect, and that you can use specific functions on specific types only. Therefore, it is time to introduce a couple of important XQuery functions.

XQuery Functions

Just as there are many data types, there are dozens of functions in XQuery as well. They are organized into multiple categories. The *data()* function, used earlier in the chapter, is a data accessor function. Some of the most useful XQuery functions supported by SQL Server are:

- **Numeric functions** ceiling(), floor(), and round()
- **String functions** concat(), contains(), substring(), string-length(), lower-case(), and upper-case()
- **Boolean and Boolean constructor functions** not(), true(), and false()
- **Nodes functions** local-name() and namespace-uri()
- **Aggregate functions** count(), min(), max(), avg(), and sum()
- **Data accessor functions** data() and string()
- **SQL Server extension functions** sql:column() and sql:variable()

You can easily conclude what a function does and what data types it supports from the function and category names. For a complete list of functions with detailed descriptions, see the Books Online for SQL Server 2012 article "XQuery Functions against the xml Data Type" at *http://msdn.microsoft.com/en-us/library/ms189254.aspx*.

The following query uses the aggregate functions count() and max() to retrieve information about orders for each customer in an XML document.

```
DECLARE @x AS XML;
SET @x='
<CustomersOrders>
  <Customer custid="1" companyname="Customer NRZBB">
    <Order orderid="10692" orderdate="2007-10-03T00:00:00" />
    <Order orderid="10702" orderdate="2007-10-13T00:00:00" />
    <Order orderid="10952" orderdate="2008-03-16T00:00:00" />
  </Customer>
  <Customer custid="2" companyname="Customer MLTDN">
    <Order orderid="10308" orderdate="2006-09-18T00:00:00" />
    <Order orderid="10926" orderdate="2008-03-04T00:00:00" />
  </Customer>
</CustomersOrders>';
SELECT @x.query('
for $i in //Customer
return
   <OrdersInfo>
      { $i/@companyname }
      <NumberOfOrders>
        { count($i/Order) }
      </NumberOfOrders>
      <LastOrder>
        { max($i/Order/@orderid) }
      </LastOrder>
   </OrdersInfo>
');
```

As you can see, this XQuery is more complicated than previous examples. The query uses iterations, known as XQuery FLWOR expressions, and formats the XML returned in the return part of the query. The FLWOR expressions are discussed later in this lesson. For now, treat this query as an example of how you can use aggregate functions in XQuery. The result of this query is as follows.

```
<OrdersInfo companyname="Customer NRZBB">
  <NumberOfOrders>3</NumberOfOrders>
  <LastOrder>10952</LastOrder>
</OrdersInfo>
<OrdersInfo companyname="Customer MLTDN">
  <NumberOfOrders>2</NumberOfOrders>
  <LastOrder>10926</LastOrder>
</OrdersInfo>
```

Navigation

You have plenty of ways to navigate through an XML document with XQuery. Actually, there is not enough space in this book to fully describe all possibilities of XQuery navigation; you have to realize this is far from a complete treatment of the topic. The basic approach is to use XPath expressions. With XQuery, you can specify a path absolutely or relatively from the current node. XQuery takes care of the current position in the document; this means that you can refer to a path relatively, starting from the current node, to which you navigated through a previous path expression. Every path consists of a sequence of steps, listed from left to right. A complete path might take the following form.

```
Node-name/child::element-name[@attribute-name=value]
```

Steps are separated with slashes; therefore, the path example described here has two steps. In the second step you can see in detail from which parts a step can be constructed. A step may consist of three parts:

- **Axis** Specifies the direction of travel. In the example, the axis is child::, which specifies child nodes of the node from the previous step.
- **Node test** Specifies the criterion for selecting nodes. In the example, element-name is the node test; it selects only nodes named element-name.
- **Predicate** Further narrows down the search. In the example, there is one predicate: [@attribute-name=value], which selects only nodes that have an attribute named attribute-name with value value, such as [@orderid=10952].

Note that in the predicate example, there is a reference to the attribute:: axis; the at sign (@) is an abbreviation for the axis attribute::. This looks a bit confusing; it might help if you think of navigation in an XML document in four directions: up (in the hierarchy), down (in the hierarchy), here (in current node), and right (in the current context level, to find attributes). Table 7-2 describes the axes supported in SQL Server.

TABLE 7-2 Axes supported in SQL Server

Axis	Abbrevation	Description
child::		Returns children of the current context node. This is the default axis; you can omit it. Direction is down.
descendant::		Retrieves all descendants of the context node. Direction is down.
self::		Retrieves the context node. Direction is here.
descendant-or-self::	//	Retrieves the context node and all its descendants. Direction is here and then down.
attribute::	@	Retrieves the specified attribute of the context node. Direction is right.
parent::	..	Retrieves the parent of the context node. Direction is up.

A *node test* follows the axis you specify. A node test can be as simple as a name test. Specifying a name means that you want nodes with that name. You can also use wildcards. An asterisk (*) means that you want any *principal node*, with any name. A principal node is the default node kind for an axis. The principal node is an attribute if the axis is attribute::, and it is an element for all other axes. You can also narrow down wildcard searches. If you want all principal nodes in the namespace prefix, use prefix:*. If you want all principal nodes named local-name, no matter which namespace they belong to, use *:local-name.

You can also perform node kind tests, which help you query nodes that are not principal nodes. You can use the following node type tests:

- **comment()** Allows you to select comment nodes.
- **node()** True for any kind of node. Do not mix this with the asterisk (*) wildcard; * means any principal node, whereas node() means any node at all.
- **processing-instruction()** Allows you to retrieve a processing instruction node.
- **text()** Allows you to retrieve text nodes, or nodes without tags.

EXAM TIP

Navigation through XML can be quite tricky; make sure you understand the complete path.

Predicates

Basic predicates include *numeric* and *Boolean* predicates. Numeric predicates simply select nodes by position. You include them in brackets. For example, /x/y[1] means the first y child element of each x element. You can also use parentheses to apply a numeric predicate to the entire result of a path. For example, (/x/y)[1] means the first element out of all nodes selected by x/y.

Boolean predicates select all nodes for which the predicate evaluates to true. XQuery supports logical and and or operators. However, you might be surprised by how comparison operators work. They work on both atomic values and sequences. For sequences, if one atomic value in a sequence leads to a true exit of the expression, the whole expression is evaluated to true. Look at the following example.

```
DECLARE @x AS XML = N'';
SELECT @x.query('(1, 2, 3) = (2, 4)');     -- true
SELECT @x.query('(5, 6) < (2, 4)');        -- false
SELECT @x.query('(1, 2, 3) = 1');          -- true
SELECT @x.query('(1, 2, 3) != 1');         -- true
```

The first expression evaluates to true because the number 2 is in both sequences. The second evaluates to false because none of the atomic values from the first sequence is less than any of the values from the second sequence. The third expression is true because there is an atomic value in the sequence on the left that is equal to the atomic value on the right. The fourth expression is true because there is an atomic value in the sequence on the left that is not equal to the atomic value on the right. Interesting result, right? Sequence (1, 2, 3) is both

equal and not equal to atomic value 1. If this confuses you, use the *value comparison operators*. (The familiar symbolic operators in the preceding example are called *general comparison operators* in XQuery.) Value comparison operators do not work on sequences, they work on singletons. The following example shows usage of value comparison operators.

```
DECLARE @x AS XML = N'';
SELECT @x.query('(5) lt (2)');              -- false
SELECT @x.query('(1) eq 1');                -- true
SELECT @x.query('(1) ne 1');                -- false
GO
DECLARE @x AS XML = N'';
SELECT @x.query('(2, 2) eq (2, 2)');     -- error
GO
```

Note that the last query, which is in a separate batch, produces an error because it is trying to use a value comparison operator on sequences. Table 7-3 lists the general comparison operators and their value comparison operator counterparts.

TABLE 7-3 General and value comparison operators

General	Value	Description
=	eq	equal
!=	ne	not equal
<	lt	less than
<=	le	less than or equal to
>	gt	greater than
>=	ge	greater than or equal to

XQuery also supports conditional if..then..else expressions with the following syntax.

```
if (<expression1>)
then
  <expression2>
else
  <expression3>
```

Note that the if..then..else expression is not used to change the program flow of the XQuery query. It is more like a function that evaluates a logical expression parameter and returns one expression or another depending on the value of the logical expression. It is more like the T-SQL CASE expression than the T-SQL IF statement.

The following code shows usage of a conditional expression.

```
DECLARE @x AS XML = N'
<Employee empid="2">
  <FirstName>fname</FirstName>
  <LastName>lname</LastName>
</Employee>
';
DECLARE @v AS NVARCHAR(20) = N'FirstName';
SELECT @x.query('
 if (sql:variable("@v")="FirstName") then
  /Employee/FirstName
 else
   /Employee/LastName
') AS FirstOrLastName;
GO
```

In this case, the result would be the first name of the employee with ID equal to 2. If you change the value of the variable *@v*, the result of the query would be the employee's last name.

FLWOR Expressions

The real power of XQuery lies in its so-called *FLWOR* expressions. FLWOR is the acronym for for, let, where, order by, and return. A FLWOR expression is actually a for each loop. You can use it to iterate through a sequence returned by an XPath expression. Although you typically iterate through a sequence of nodes, you can use FLWOR expressions to iterate through any sequence. You can limit the nodes to be processed with a predicate, sort the nodes, and format the returned XML. The parts of a FLWOR statement are:

- **For** With a for clause, you bind iterator variables to input sequences. Input sequences are either sequences of nodes or sequences of atomic values. You create atomic value sequences by using literals or functions.
- **Let** With the optional let clause, you assign a value to a variable for a specific iteration. The expression used for an assignment can return a sequence of nodes or a sequence of atomic values.
- **Where** With the optional where clause, you filter the iteration.
- **Order by** Using the order by clause, you can control the order in which the elements of the input sequence are processed. You control the order based on atomic values.
- **Return** The return clause is evaluated once per iteration, and the results are returned to the client in the iteration order. With this clause, you format the resulting XML.

Here is an example of usage of all FLWOR clauses.

```
DECLARE @x AS XML;
SET @x = N'
<CustomersOrders>
  <Customer custid="1">
    <!-- Comment 111 -->
    <companyname>Customer NRZBB</companyname>
    <Order orderid="10692">
      <orderdate>2007-10-03T00:00:00</orderdate>
    </Order>
    <Order orderid="10702">
      <orderdate>2007-10-13T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-16T00:00:00</orderdate>
    </Order>
  </Customer>
  <Customer custid="2">
    <!-- Comment 222 -->
    <companyname>Customer MLTDN</companyname>
    <Order orderid="10308">
      <orderdate>2006-09-18T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-04T00:00:00</orderdate>
    </Order>
  </Customer>
</CustomersOrders>';
SELECT @x.query('for $i in CustomersOrders/Customer/Order
                 let $j := $i/orderdate
                 where $i/@orderid < 10900
                 order by ($j)[1]
                 return
                 <Order-orderid-element>
                  <orderid>{data($i/@orderid)}</orderid>
                  {$j}
                 </Order-orderid-element>')
       AS [Filtered, sorted and reformatted orders with let clause];
```

The query iterates, as you can see from the for clause, through all Order nodes using an iterator variable and returns those nodes. The name of the iterator variable must start with a dollar sign ($) in XQuery. The where clause limits the Order nodes processed to those with an orderid attribute smaller than 10900.

The expression passed to the order by clause must return values of a type compatible with the gt XQuery operator. As you'll recall, the gt operator expects atomic values. The query orders the XML returned by the orderdate element. Although there is a single orderdate element per order, XQuery does not know this, and it considers orderdate to be a sequence, not an atomic value. The numeric predicate specifies the first orderdate element of an order as the value to order by. Without this numeric predicate, you would get an error.

The return clause shapes the XML returned. It converts the orderid attribute to an element by creating the element manually and extracting only the value of the attribute with the data() function. It returns the orderdate element as well, and wraps both in the Order-orderid-element element. Note the braces around the expressions that extract the value of the orderid element and the orderdate element. XQuery evaluates expressions in braces; without braces, everything would be treated as a string literal and returned as such.

The let clause assigns a name to the $i/orderdate expression. This expression repeats twice in the query, in the order by and the return clauses. To name the expression, you have to use a variable different from $i. XQuery inserts the expression every time the new variable is referenced. Here is the result of the query.

```
<Order-orderid-element>
  <orderid>10308</orderid>
  <orderdate>2006-09-18T00:00:00</orderdate>
</Order-orderid-element>
<Order-orderid-element>
  <orderid>10692</orderid>
  <orderdate>2007-10-03T00:00:00</orderdate>
</Order-orderid-element>
<Order-orderid-element>
  <orderid>10702</orderid>
  <orderdate>2007-10-13T00:00:00</orderdate>
</Order-orderid-element>
```

Quick Check

1. **What do you do in the return clause of the FLWOR expressions?**
2. **What would be the result of the expression (12, 4, 7) != 7?**

Quick Check Answers

1. **In the return clause, you format the resulting XML of a query.**
2. **The result would be true.**

PRACTICE Using XQuery/XPath Navigation

In this practice, you use XPath expressions for navigation inside XQuery. You start with simple path expressions, and then use more complex path expressions with predicates.

If you encounter a problem completing an exercise, you can install the completed projects from the companion content for this chapter and lesson.

EXERCISE 1 Use Simple XPath Expressions

In this exercise, you use simple XPath expressions to return subsets of XML data.

1. If you closed SSMS, start it and connect to your SQL Server instance. Open a new query window by clicking the New Query button.
2. Connect to your TSQL2012 database.
3. Use the following XML instance for testing the navigation.

```
DECLARE @x AS XML;
SET @x = N'
<CustomersOrders>
  <Customer custid="1">
    <!-- Comment 111 -->
    <companyname>Customer NRZBB</companyname>
    <Order orderid="10692">
      <orderdate>2007-10-03T00:00:00</orderdate>
    </Order>
    <Order orderid="10702">
      <orderdate>2007-10-13T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-16T00:00:00</orderdate>
    </Order>
  </Customer>
  <Customer custid="2">
    <!-- Comment 222 -->
    <companyname>Customer MLTDN</companyname>
    <Order orderid="10308">
      <orderdate>2006-09-18T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-04T00:00:00</orderdate>
    </Order>
  </Customer>
</CustomersOrders>';
```

4. Write a query that selects Customer nodes with child nodes. Select principal nodes (elements in this context) only. The result should be similar to the abbreviated result here.

```
1. Principal nodes
--------------------------------------------------------------------------------
<companyname>Customer NRZBB</companyname><Order orderid="10692"><orderdate>2007-
```

Use the following query to get the desired result.

```
SELECT @x.query('CustomersOrders/Customer/*')
       AS [1. Principal nodes];
```

5. Now return all nodes, not just the principal ones. The result should be similar to the abbreviated result here.

```
2. All nodes
--------------------------------------------------------------------------------
<!-- Comment 111 --><companyname>Customer NRZBB</companyname><Order orderid="106
```

Use the following query to get the desired result.

```
SELECT @x.query('CustomersOrders/Customer/node()')
       AS [2. All nodes];
```

6. Return comment nodes only. The result should be similar to the result here.

```
3. Comment nodes
--------------------------------------------------------------------------------
<!-- Comment 111 --><!-- Comment 222 -->
```

Use the following query to get the desired result.

```
SELECT @x.query('CustomersOrders/Customer/comment()')
       AS [3. Comment nodes];
```

EXERCISE 2 Use XPath Expressions with Predicates

In this exercise, you use XPath expressions with predicates to return filtered subsets of XML data.

1. Use the following XML instance (the same as in the previous exercise) for testing the navigation.

```
DECLARE @x AS XML;
SET @x = N'
<CustomersOrders>
  <Customer custid="1">
    <!-- Comment 111 -->
    <companyname>Customer NRZBB</companyname>
    <Order orderid="10692">
      <orderdate>2007-10-03T00:00:00</orderdate>
    </Order>
    <Order orderid="10702">
      <orderdate>2007-10-13T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-16T00:00:00</orderdate>
    </Order>
  </Customer>
  <Customer custid="2">
    <!-- Comment 222 -->
    <companyname>Customer MLTDN</companyname>
    <Order orderid="10308">
      <orderdate>2006-09-18T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-04T00:00:00</orderdate>
    </Order>
  </Customer>
</CustomersOrders>';
```

2. Return all orders for customer 2. The result should be similar to the abbreviated result here.

```
4. Customer 2 orders
--------------------------------------------------------------------------------
<Order orderid="10308"><orderdate>2006-09-18T00:00:00</orderdate></Order><Order
```

Use the following query to get the desired result.

```
SELECT @x.query('//Customer[@custid=2]/Order')
       AS [4. Customer 2 orders];
```

3. Return all orders with order number 10952, no matter who the customer is. The result should be similar to the abbreviated result here.

```
5. Orders with orderid=10952
--------------------------------------------------------------------------------
<Order orderid="10952"><orderdate>2008-03-16T00:00:00</orderdate></Order><Order
```

Use the following query to get the desired result.

```
SELECT @x.query('//Order[@orderid=10952]')
       AS [5. Orders with orderid=10952];
```

4. Return the second customer who has at least one order. The result should be similar to the abbreviated result here.

```
6. 2nd Customer with at least one Order
--------------------------------------------------------------------------------
<Customer custid="2"><!-- Comment 222 --><companyname>Customer MLTDN</companyname
```

Use the following query to get the desired result.

```
SELECT @x.query('(/CustomersOrders/Customer/
                  Order/parent::Customer)[2]')
       AS [6. 2nd Customer with at least one Order];
```

Lesson Summary

- You can use the XQuery language inside T-SQL queries to query XML data.
- XQuery supports its own data types and functions.
- You use XPath expressions to navigate through an XML instance.
- The real power of XQuery is in the FLWOR expressions.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. Which of the following is not a FLWOR clause?
   - **A.** for
   - **B.** let
   - **C.** where
   - **D.** over
   - **E.** return
2. Which node type test can be used to retrieve all nodes of an XML instance?
   - **A.** Asterisk (*)
   - **B.** comment()
   - **C.** node()
   - **D.** text()
3. Which conditional expression is supported in XQuery?
   - **A.** IIF
   - **B.** if..then..else
   - **C.** CASE
   - **D.** switch

Lesson 3: Using the XML Data Type

XML is the standard format for exchanging data among different applications and platforms. It is widely used, and almost all modern technologies support it. Databases simply have to deal with XML. Although XML could be stored as simple text, plain text representation means having no knowledge of the structure built into an XML document. You could decompose the text, store it in multiple relational tables, and use relational technologies to manipulate the data. Relational structures are quite static and not so easy to change. Think of dynamic or volatile XML structures. Storing XML data in a native XML data type solves these problems, enabling functionality attached to the type that can accommodate support for a wide variety of XML technologies.

After this lesson, you will be able to:

- Use the XML data type and its methods.
- Index XML data.

Estimated lesson time: 45 minutes

When to Use the XML Data Type

A database schema is sometimes volatile. Think about situations in which you have to support many different schemas for the same kind of event. SQL Server has many such cases within it. Data definition language (DDL) triggers and extended events are good examples. There are dozens of different DDL events. Each event returns different event information; each event returns data with a different schema. A conscious design choice was that DDL triggers return event information in XML format via the eventdata() function. Event information in XML format is quite easy to manipulate. Furthermore, with this architecture, SQL Server will be able to extend support for new DDL events in future versions more easily.

Another interesting example of internal XML support is XML showplan. You can generate execution plan information in XML format by using the SET SHOWPLAN_XML and SET STATISTICS XML statements. Think of the value for applications and tools that need execution plan information—it's easy to request and parse now. You can even force the optimizer to use a specified execution plan by providing the XML plan in a USE PLAN query hint.

Another place to use XML is to represent data that is sparse. Your data is sparse and you have a lot of NULLs if some columns are not applicable to all rows. Standard solutions for such a problem introduce subtypes or implement an open schema model in a relational environment. However, a solution based on XML could be the easiest to implement. A solution that introduces subtypes can lead to many new tables. SQL Server 2008 introduced sparse columns and filtered indexes. Sparse columns could be another solution for having attributes that are not applicable for all rows in a table. Sparse columns have optimized storage for NULLs. If you have to index them, you can efficiently use filtered indexes to index known values only; this way, you optimize table and index storage. In addition, you can have access to all sparse columns at once through a column set. A column set is an XML representation of all the sparse columns that is even updateable. However, with sparse columns and a column set, the schema is more complicated than a schema with an explicit XML column.

You could have other reasons to use an XML model. XML inherently supports hierarchical and sorted data. If ordering is inherent in your data, you might decide to store it as XML. You could receive XML documents from your business partner, and you might not need to shred the document to tables. It might be more practical to just store the complete XML documents in your database, without shredding.

XML Data Type Methods

In the XQuery introduction in this chapter, you already saw the XML data type. XQuery was a parameter for the query() method of this type. An XML data type includes five methods that accept XQuery as a parameter. The methods support querying (the query() method), retrieving atomic values (the value() method), checking existence (the exist() method), modifying sections within the XML data (the modify() method) as opposed to overwriting the whole thing, and shredding XML data into multiple rows in a result set (the nodes() method). You use the XML data type methods in the practice for this lesson.

The value() method of the XML data type returns a scalar value, so it can be specified anywhere where scalar values are allowed; for example, in the SELECT list of a query. Note that the value() method accepts an XQuery expression as the first input parameter. The second parameter is the SQL Server data type returned. The value() method must return a scalar value; therefore, you have to specify the position of the element in the sequence you are browsing, even if you know that there is only one.

You can use the exist() method to test if a specific node exists in an XML instance. Typical usage of this clause is in the WHERE clause of T-SQL queries. The exist() method returns a bit, a flag that represents true or false. It can return the following:

- 1, representing true, if the XQuery expression in a query returns a nonempty result. That means that the node searched for exists in the XML instance.
- 0, representing false, if the XQuery expression returns an empty result.
- NULL, if the XML instance is NULL.

The query() method, as the name implies, is used to query XML data. You already know this method from the previous lesson of this chapter. It returns an instance of an untyped XML value.

The XML data type is a large object type. The amount of data stored in a column of this type can be very large. It would not be very practical to replace the complete value when all you need is just to change a small portion of it; for example, a scalar value of some subelement. The SQL Server XML data type provides you with the modify() method, similar in concept to the WRITE method that can be used in a T-SQL UPDATE statement for VARCHAR(MAX) and the other MAX types. You invoke the modify() method in an UPDATE T-SQL statement.

The W3C standard doesn't support data modification with XQuery. However, SQL Server provides its own language extensions to support data modification with XQuery. SQL Server XQuery supports three data manipulation language (DML) keywords for data modification: insert, delete, and replace value of.

The nodes() method is useful when you want to shred an XML value into relational data. Its purpose is therefore the same as the purpose of the OPENXML rowset function introduced in Lesson 1 of this chapter. However, using the nodes() method is usually much faster than preparing the DOM with a call to sp_xml_preparedocument, executing a SELECT..FROM OPENXML statement, and calling sp_xml_removedocument. The nodes() method prepares DOM internally, during the execution of the T-SQL SELECT. The OPENXML approach could be faster if you prepared the DOM once and then shredded it multiple times in the same batch.

The result of the nodes() method is a result set that contains logical copies of the original XML instances. In those logical copies, the context node of every row instance is set to one of the nodes identified by the XQuery expression, meaning that you get a row for every single node from the starting point defined by the XQuery expression. The nodes() method returns copies of the XML values, so you have to use additional methods to extract the scalar values

out of them. The nodes() method has to be invoked for every row in the table. With the T-SQL APPLY operator, you can invoke a right table expression for every row of a left table expression in the FROM part.

Using the XML Data Type for Dynamic Schema

In this lesson, you learn how to use an XML data type inside your database through an example. This example shows how you can make a relational database schema dynamic. The example extends the Products table from the TSQL2012 database.

Suppose that you need to store some specific attributes only for beverages and other attributes only for condiments. For example, you need to store the percentage of recommended daily allowance (RDA) of vitamins only for beverages, and a short description only for condiments to indicate the condiment's general character (such as sweet, spicy, or salty). You could add an XML data type column to the Production.Products table of the TSQL2012 database; for this example, call it additionalattributes. Because the other product categories have no additional attributes, this column has to be nullable. The following code alters the Production.Products table to add this column.

```
ALTER TABLE Production.Products
 ADD additionalattributes XML NULL;
```

Before inserting data in the new column, you might want to constrain the values of this column. You should use a typed XML, an XML validated against a schema. With an XML schema, you constrain the possible nodes, the data type of those nodes, and more. In SQL Server, you can validate XML data against an XML schema collection. This is exactly what you need for a dynamic schema; if you could validate XML data against a single schema only, you could not use an XML data type for a dynamic schema solution, because XML instances would be limited to a single schema. Validation against a collection of schemas enables support of different schemas for beverages and condiments. If you wanted to validate XML values only against a single schema, you would define only a single schema in the collection.

You create the schema collection by using the CREATE XML SCHEMA COLLECTION T-SQL statement. You have to supply the XML schema, an XSD document, as input. Creating the schema is a task that should not be taken lightly. If you make an error in the schema, some invalid data might be accepted and some valid data might be rejected.

The easiest way to create XML schemas is to create relational tables first, and then use the XMLSCHEMA option of the FOR XML clause. Store the resulting XML value (the schema) in a variable, and provide the variable as input to the CREATE XML SCHEMA COLLECTION statement. The following code creates two auxiliary empty tables for beverages and condiments, and then uses SELECT with the FOR XML clause to create an XML schema from those tables. Then it stores the schemas in a variable, and creates a schema collection from that variable. Finally, after the schema collection is created, the code drops the auxiliary tables.

```
-- Auxiliary tables
CREATE TABLE dbo.Beverages
(
  percentvitaminsRDA INT
);
CREATE TABLE dbo.Condiments
(
  shortdescription NVARCHAR(50)
);
GO
-- Store the Schemas in a Variable and Create the Collection
DECLARE @mySchema NVARCHAR(MAX);
SET @mySchema = N'';
SET @mySchema = @mySchema +
  (SELECT *
   FROM Beverages
   FOR XML AUTO, ELEMENTS, XMLSCHEMA('Beverages'));
SET @mySchema = @mySchema +
  (SELECT *
   FROM Condiments
   FOR XML AUTO, ELEMENTS, XMLSCHEMA('Condiments'));
SELECT CAST(@mySchema AS XML);
CREATE XML SCHEMA COLLECTION dbo.ProductsAdditionalAttributes AS @mySchema;
GO
-- Drop Auxiliary Tables
DROP TABLE dbo.Beverages, dbo.Condiments;
GO
```

The next step is to alter the XML column from a well-formed state to a schema-validated one.

```
ALTER TABLE Production.Products
  ALTER COLUMN additionalattributes
   XML(dbo.ProductsAdditionalAttributes);
```

You can get information about schema collections by querying the catalog views sys.xml_schema_collections, sys.xml_schema_namespaces, sys.xml_schema_components, and some others views in the sys schema with names that start with xml_schema_. However, a schema collection is stored in SQL Server in tabular format, not in XML format. It would make sense to perform the same schema validation on the client side as well. Why would you send data to the server side if the relational database management system (RDBMS) will reject it? You can perform schema collection validation in Microsoft .NET code as well, as long as you have the schemas. Therefore, it makes sense to save the schemas you create with T-SQL in files in a file system as well. If you forgot to save the schemas in files, you can still retrieve them from SQL Server schema collections with the xml_schema_namespace system function. Note that the schema returned by this function might not be lexically the same as the original schema used when you created your schema collection. Comments, annotations, and white spaces are lost. However, the aspects of the schema used for validation are preserved.

Before using the new data type, you have to take care of one more issue. How do you avoid binding the wrong schema to a product of a specific category? For example, how do you prevent binding a condiments schema to a beverage? You could solve this issue with a trigger; however, having a declarative constraint, a check constraint, is preferable. This is why the code added namespaces to the schemas. You need to check whether the namespace is the same as the product category name. You cannot use XML data type methods inside constraints. You have to create two additional functions: one retrieves the XML namespace of the additionalattributes XML column, and the other retrieves the category name of a product. In the check constraint, you can check whether the return values of both functions are equal. Here is the code that creates both functions and adds a check constraint to the Production. Products table.

```
-- Function to Retrieve the Namespace
CREATE FUNCTION dbo.GetNamespace(@chkcol XML)
 RETURNS NVARCHAR(15)
AS
BEGIN
 RETURN @chkcol.value('namespace-uri((/*)[1])','NVARCHAR(15)')
END;
GO
-- Function to Retrieve the Category Name
CREATE FUNCTION dbo.GetCategoryName(@catid INT)
 RETURNS NVARCHAR(15)
AS
BEGIN
 RETURN
  (SELECT categoryname
   FROM Production.Categories
   WHERE categoryid = @catid)
END;
GO
-- Add the Constraint
ALTER TABLE Production.Products ADD CONSTRAINT ck_Namespace
 CHECK (dbo.GetNamespace(additionalattributes) =
        dbo.GetCategoryName(categoryid));
GO
```

The infrastructure is prepared. You can try to insert some valid XML data in your new column.

```
-- Beverage
UPDATE Production.Products
   SET additionalattributes = N'
<Beverages xmlns="Beverages">
  <percentvitaminsRDA>27</percentvitaminsRDA>
</Beverages>'
WHERE productid = 1;
-- Condiment
UPDATE Production.Products
   SET additionalattributes = N'
<Condiments xmlns="Condiments">
  <shortdescription>very sweet</shortdescription>
</Condiments>'
WHERE productid = 3;
```

To test whether the schema validation and check constraint work, you should try to insert some invalid data as well.

```
-- String instead of int
UPDATE Production.Products
   SET additionalattributes = N'
<Beverages xmlns="Beverages">
  <percentvitaminsRDA>twenty seven</percentvitaminsRDA>
</Beverages>'
WHERE productid = 1;
-- Wrong namespace
UPDATE Production.Products
   SET additionalattributes = N'
<Condiments xmlns="Condiments">
  <shortdescription>very sweet</shortdescription>
</Condiments>'
WHERE productid = 2;
-- Wrong element
UPDATE Production.Products
   SET additionalattributes = N'
<Condiments xmlns="Condiments">
  <unknownelement>very sweet</unknownelement>
</Condiments>'
WHERE productid = 3;
```

You should get errors for all three UPDATE statements. You can check the data with the SELECT statement. When you are done, you could clean up the TSQL2012 database with the following code.

```
ALTER TABLE Production.Products
 DROP CONSTRAINT ck_Namespace;
ALTER TABLE Production.Products
 DROP COLUMN additionalattributes;
DROP XML SCHEMA COLLECTION dbo.ProductsAdditionalAttributes;
DROP FUNCTION dbo.GetNamespace;
DROP FUNCTION dbo.GetCategoryName;
GO
```

Quick Check

- **Which XML data type method would you use to retrieve scalar values from an XML instance?**

Quick Check Answer

- **The value() XML data type method retrieves scalar values from an XML instance.**

XML Indexes

The XML data type is actually a large object type. There can be up to 2 gigabytes (GB) of data in every single column value. Scanning through the XML data sequentially is not a very efficient way of retrieving a simple scalar value. With relational data, you can create an index on a filtered column, allowing an index seek operation instead of a table scan. Similarly, you can index XML columns with specialized *XML indexes*. The first index you create on an XML column is the *primary XML index*. This index contains a shredded persisted representation of the XML values. For each XML value in the column, the index creates several rows of data. The number of rows in the index is approximately the number of nodes in the XML value. Such an index alone can speed up searches for a specific element by using the exist() method. After creating the primary XML index, you can create up to three other types of *secondary XML indexes*:

- **PATH** This secondary XML index is especially useful if your queries specify path expressions. It speeds up the exist() method better than the Primary XML index. Such an index also speeds up queries that use value() for a fully specified path.
- **VALUE** This secondary XML index is useful if queries are value-based and the path is not fully specified or it includes a wildcard.
- **PROPERTY** This secondary XML index is very useful for queries that retrieve one or more values from individual XML instances by using the value() method.

The primary XML index has to be created first. It can be created only on tables with a clustered primary key.

PRACTICE Using XML Data Type Methods

In this practice, you use XML data type methods.

If you encounter a problem completing an exercise, you can install the completed projects from the companion content for this chapter and lesson.

EXERCISE 1 Use the value() and exist() Methods

In this exercise, you use the value() and exist() XML data type methods.

1. If you closed SSMS, start it and connect to your SQL Server instance. Open a new query window by clicking the New Query button.
2. Connect to your TSQL2012 database.

3. Use the following XML instance for testing the XML data type methods.

```
DECLARE @x AS XML;
SET @x = N'
<CustomersOrders>
  <Customer custid="1">
    <!-- Comment 111 -->
    <companyname>Customer NRZBB</companyname>
    <Order orderid="10692">
      <orderdate>2007-10-03T00:00:00</orderdate>
    </Order>
    <Order orderid="10702">
      <orderdate>2007-10-13T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-16T00:00:00</orderdate>
    </Order>
  </Customer>
  <Customer custid="2">
    <!-- Comment 222 -->
    <companyname>Customer MLTDN</companyname>
    <Order orderid="10308">
      <orderdate>2006-09-18T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-04T00:00:00</orderdate>
    </Order>
  </Customer>
</CustomersOrders>';
```

4. Write a query that retrieves the first customer name as a scalar value. The result should be similar to the result here.

```
First Customer Name
--------------------
Customer NRZBB
```

Use the following query to get the desired result.

```
SELECT @x.value('(/CustomersOrders/Customer/companyname)[1]',
      'NVARCHAR(20)')
       AS [First Customer Name];
```

5. Now check whether companyname and address nodes exist under the Customer node. The result should be similar to the result here.

```
Company Name Exists Address Exists
------------------- --------------
1                   0
```

Use the following query to get the desired result.

```
SELECT @x.exist('(/CustomersOrders/Customer/companyname)')
        AS [Company Name Exists],
       @x.exist('(/CustomersOrders/Customer/address)')
        AS [Address Exists];
```

EXERCISE 2 Use the query(), nodes(), and modify() Methods

In this exercise, you use the query(), nodes(), and modify() XML data type methods.

1. Use the following XML instance (the same instance as in the previous exercise) for testing the XML data type methods.

```
DECLARE @x AS XML;
SET @x = N'
<CustomersOrders>
  <Customer custid="1">
    <!-- Comment 111 -->
    <companyname>Customer NRZBB</companyname>
    <Order orderid="10692">
      <orderdate>2007-10-03T00:00:00</orderdate>
    </Order>
    <Order orderid="10702">
      <orderdate>2007-10-13T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-16T00:00:00</orderdate>
    </Order>
  </Customer>
  <Customer custid="2">
    <!-- Comment 222 -->
    <companyname>Customer MLTDN</companyname>
    <Order orderid="10308">
      <orderdate>2006-09-18T00:00:00</orderdate>
    </Order>
    <Order orderid="10952">
      <orderdate>2008-03-04T00:00:00</orderdate>
    </Order>
  </Customer>
</CustomersOrders>';
```

2. Return all orders for the customer with @custid equal to 1 (the first customer in the XML document) as XML. The result should be similar to the result here.

```
<Order orderid="10692">
  <orderdate>2007-10-03T00:00:00</orderdate>
</Order>
<Order orderid="10702">
  <orderdate>2007-10-13T00:00:00</orderdate>
</Order>
<Order orderid="10952">
  <orderdate>2008-03-16T00:00:00</orderdate>
</Order>
```

Use the following query to get the desired result.

```
SELECT @x.query('//Customer[@custid=1]/Order')
       AS [Customer 1 orders];
```

3. Shred all orders information for the customer with @custid equal to 1 (the first customer in the XML document). The result should be similar to the result here.

```
Order Id    Order Date
----------- -----------------------
10692       2007-10-03 00:00:00.000
10702       2007-10-13 00:00:00.000
10952       2008-03-16 00:00:00.000
```

Use the following query to get the desired result.

```
SELECT  T.c.value('./@orderid[1]', 'INT') AS [Order Id],
 T.c.value('./orderdate[1]', 'DATETIME') AS [Order Date]
FROM @x.nodes('//Customer[@custid=1]/Order')
      AS T(c);
```

4. Update the name of the first customer and then retrieve the new name. The result should be similar to the result here.

```
First Customer New Name
-----------------------
New Company Name
```

Use the following query to get the desired result.

```
SET @x.modify('replace value of
    /CustomersOrders[1]/Customer[1]/companyname[1]/text()[1]
  with "New Company Name"');
SELECT @x.value('(/CustomersOrders/Customer/companyname)[1]',
       'NVARCHAR(20)')
       AS [First Customer New Name];
```

5. Now Exit SSMS.

Lesson Summary

- The XML data type is useful for many scenarios inside a relational database.
- You can validate XML instances against a schema collection.
- You can work with XML data through XML data type methods.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. Which of the following is not an XML data type method?
   - **A.** merge()
   - **B.** nodes()
   - **C.** exist()
   - **D.** value()
2. What kind of XML indexes can you create? (Choose all that apply.)
   - **A.** PRIMARY
   - **B.** PATH
   - **C.** ATTRIBUTE
   - **D.** PRINCIPALNODES
3. Which XML data type method do you use to shred XML data to tabular format?
   - **A.** modify()
   - **B.** nodes()
   - **C.** exist()
   - **D.** value()

Case Scenarios

In the following case scenarios, you apply what you've learned about querying and managing XML data. You can find the answers to these questions in the "Answers" section at the end of this chapter.

Case Scenario 1: Reports from XML Data

A company that hired you as a consultant uses a website to get reviews of their products from their customers. They store those reviews in an XML column called reviewsXML of a table called ProductReviews. The XML column is validated against a schema and contains, among others, firstname, lastname, and datereviewed elements. The company wants to generate a report with names of the reviewers and dates of reviews. Additionally, because there are already many very long reviews, the company worries about the performance of this report.

1. How could you get the data needed for the report?
2. What would you do to maximize the performance of the report?

Case Scenario 2: Dynamic Schema

You need to provide a solution for a dynamic schema for the Products table in your company. All products have the same basic attributes, like product ID, product name, and list price. However, different groups of products have different additional attributes. Besides dynamic schema for the variable part of the attributes, you need to ensure at least basic constraints, like data types, for these variable attributes.

1. How would you make the schema of the Products table dynamic?
2. How would you ensure that at least basic constraints would be enforced?

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks.

Query XML Data

In the AdventureWorks2012 demo database, there is the HumanResources.JobCandidate table. It contains a Resume XML data type column.

- **Practice 1** Find all first and last names in this column.
- **Practice 2** Find all candidates from Chicago.
- **Practice 3** Return distinct states found in all resumes.

Answers

This section contains the answers to the lesson review questions and solutions to the case scenarios in this chapter.

Lesson 1

1. **Correct Answers: A and D**
   - **A. Correct:** FOR XML AUTO is a valid option to produce automatically formatted XML.
   - **B. Incorrect:** There is no FOR XML MANUAL option.
   - **C. Incorrect:** There is no FOR XML DOCUMENT option.
   - **D. Correct:** With the FOR XML PATH option, you can format XML explicitly.
2. **Correct Answer: C**
   - **A. Incorrect:** There is no specific ATTRIBUTES directive. Attribute-centric formatting is the default.
   - **B. Incorrect:** With the ROOT option, you can specify a name for the root element.
   - **C. Correct:** Use the ELEMENTS option to produce element-centric XML.
   - **D. Incorrect:** With the XMLSCHEMA option, you produce inline XSD.
3. **Correct Answers: B and D**
   - **A. Incorrect:** FOR XML AUTO automatically formats the XML retuned.
   - **B. Correct:** FOR XML EXPLICIT allows you to manually format the XML returned.
   - **C. Incorrect:** FOR XML RAW automatically formats the XML retuned.
   - **D. Correct:** FOR XML PATH allows you to manually format the XML returned.

Lesson 2

1. **Correct Answer: D**
   - **A. Incorrect:** for is a FLWOR clause.
   - **B. Incorrect:** let is a FLWOR clause.
   - **C. Incorrect:** where is a FLWOR clause.
   - **D. Correct:** over is not a FLWOR clause; O stands for the order by clause.
   - **E. Incorrect:** return is a FLWOR clause.

2. **Correct Answer: C**
   - A. **Incorrect:** With the asterisk (*), you retrieve all principal nodes.
   - B. **Incorrect:** With comment(), you retrieve comment nodes.
   - C. **Correct:** You use the node() node-type test to retrieve all nodes.
   - D. **Incorrect:** With text(), you retrieve text nodes.
3. **Correct Answer: B**
   - A. **Incorrect:** IIF is not an XQuery expression.
   - B. **Correct:** XQuery supports the if..then..else conditional expression.
   - C. **Incorrect:** CASE is not an XQuery expression.
   - D. **Incorrect:** switch is not an XQuery expression.

Lesson 3

1. **Correct Answer: A**
   - A. **Correct:** merge() is not an XML data type method.
   - B. **Incorrect:** nodes() is an XML data type method.
   - C. **Incorrect:** exist() is an XML data type method.
   - D. **Incorrect:** value() is an XML data type method.
2. **Correct Answers: A and B**
   - A. **Correct:** You create a PRIMARY XML index before any other XML indexes.
   - B. **Correct:** A PATH XML index is especially useful if your queries specify path expressions.
   - C. **Incorrect:** There is no general ATTRIBUTE XML index.
   - D. **Incorrect:** There is no general PRINCIPALNODES XML index.
3. **Correct Answer: B**
   - A. **Incorrect:** You use the modify() method to update XML data.
   - B. **Correct:** You use the nodes() method to shred XML data.
   - C. **Incorrect:** You use the exist() method to test whether a node exists.
   - D. **Incorrect:** You use the value() method to retrieve a scalar value from XML data.

Case Scenario 1

1. You could use the value() XML data type method to retrieve the scalar values needed for the report.
2. You should consider using XML indexes in order to maximize the performance of the report.

Case Scenario 2

1. You could use the XML data type column to store the variable attributes in XML format.
2. You could validate the XML against an XML schema collection.

Index

Symbols

- $action function, 400
- $ (dollar sign), 34, 271
- & (ampersand), 223
- ' (apostrophe), 223
- * (asterisk), 31, 241
- @ (at sign), 34, 229, 240, 271
- .bak extension, 484
- : (colon), 227, 237
- = (equal), 242
- @@ERROR function, 435, 440, 444–445, 474
  - error handling using, 440
- @error_message string, 443
- @@FETCH_STATUS function, 602
- > (greater than), 223, 242
- >= (greater than or equal to), 71, 242
- @@IDENTITY function, 371–372
  - SCOPE_IDENTITY vs., 372
- < (less than) operator, 71, 223, 242
- <= (less than or equal to), 242
- .NET assemblies, 470
- != (not equal) operator, 3, 242
- <> (not equal) operator, 3
- # (number sign), 34, 271
- @numrows, 476
- ( ) (parentheses), 308
- % (percent sign), 48
- + (plus) operator, 38, 47
- ? (question mark), 223
- " (quotation mark), 223, 271
- @range_first_value output parameter, 377
- @range_size input parameter, 377
- @@ROWCOUNT, 492
- @rowsreturned, 476
- ; (semicolon), 223, 308
- / (slash character), 229
- [] (square brackets), 48, 271
- @statement input parameter, 457
- @@TRANCOUNT function, 415–419, 429, 444
  - output of, 428
  - XACT_STATE() vs., 416
- _ (underscore), 34, 271
  - as wildcard, 48

A

- abstraction layer, 317
- accents, 194
- ACCENT_SENSITIVITY option, 195
- access control and permissions, 127
- ACID properties, 413–414, 421, 426
  - atomicity, 413
  - consistency, 413
  - isolation, 413
- across batches, 612
- Actual Execution Mode, 655
- Actual Number Of Rows property, 635–636
- addition time functions, 45
- ad hoc queries, 521
- advanced locking modes, 422
- AFTER triggers, 491–496, 512
  - nested, 494–495
  - writing, 498–499
- aggregate data by criteria, 159
- aggregate functions, 150, 152, 306
  - window, 172–176
- aggregate functions (XQuery), 238
- aggregation elements and PIVOT operator, 165
- aliases
  - column, 178
  - for namespaces, 224
  - with table expressions, 121

  - and table names, 30
  - using short, 106
- aliasing
  - attribute, 32
  - inline vs. external, 124
  - problems with, 22
  - of tables in joins, 104
- “all-at-once” property, 124
- all-at-once UPDATE, 351–352
- allocation order scan, 633–634
- ALTER command, 275, 473
- ALTER DATABASE command, 590
- ALTER FULLTEXT CATALOG statement, 195
- ALTER INDEX ... REBUILD statement, 558
- ALTER INDEX ... REORGANIZE statement, 558
- ALTER SCHEMA TRANSFER statement, 270
- ALTER SEQUENCE command, 375
- ALTER statement, 471
  - with synonyms, 316
- ALTER TABLE command, 276–277
  - and constraints, 281
  - declaring column as primary key with, 282
- ALTER VIEW command, 305
- American National Standards Institute (ANSI), 3
- ampersand (&), 223
- analyze a query, 537–538
- analyzing error messages, 436
- AND (&) bitwise operator, 496
- AND (logical operator), 66–67, 519, 580–583
  - support, 583
  - and WHERE clause, 106
- ANSI SQL standard, 271, 273, 453
- apostrophe ('), 223
- APPLY operator, 128–132, 388
  - CROSS APPLY operator, 129–131
  - OUTER APPLY operator, 131–132
  - UPDATE statement, 347
- approximate numeric data types, 37
- AS command, 474
- asterisk (*), 31, 241
- AS <type>, 374
- “at” (@) character, 34, 229, 240, 271
- atomicity, 413
- atomic types (XQuery), 238
- attribute:: axis (XQuery), 240
- attribute-centric XML, 229
- attribute(s), 4
  - aliasing, 32
  - of elements, 224
- authorized database users, 270
- autocommit mode, 416
- auto_created, 587
- AUTO_CREATE_STATISTICS, 585, 589
- AUTO option (XML), 227–229
- auto-parameterization, 521
- AUTO_UPDATE_STATISTICS, 585, 589
- AUTO_UPDATE_STATISTICS_ASYNC, 585
- avg_fragmentation_in_percent, 562
- AVG function, 172
  - as aggregate function, 152
- avg_page_space_used_in_percent, 561–562
- avg_space_used_in_percent, 553
- Avoiding MERGE Conflicts, 386
- axis (Xquery navigation), 240

B

- backing up the database, 266
- BACKUP DATABASE commands, 484–485
- bags, 7
- balanced tree pages, 556
- balanced tree(s), 550–563, 639–640
- barcode numbers, 39
- base tables, 266
- basic joins, 638
- basic locking, 422–426
- batch operations, 647–660
- batch processing, 653–658
- BEGIN/END block, 474, 477, 479, 502–503
- BEGIN/END statement, 502
- begin tag (XML), 222
- BEGIN (TRAN or TRANSACTION) command, 415–416, 418, 420
- Ben-Gan, Itzik, 183
- BETWEEN operator, 71, 578
- BIGINT data type, 42, 84, 374
- BINARY data type, 38, 39
- binary strings, 37
- bitmap
  - filtered hash join, 640
  - filtering optimized hash, 519
- Bitmap operator, 655
- blocking, 423, 426, 429–431
  - exclusive lock, 423
  - writers, 430
  - Writers, 426

body (of relation), 4, 6
Books Online for SQL Server 2012, 37, 39, 41, 44, 48, 57, 202, 226, 239, 267, 306, 316
Books Online for SQL Server 2012 article, 377
Boolean constructor functions (XQuery), 238
Boolean functions (XQuery), 238
Boolean predicates, 241
brancing logic, 477–481
BREAK statement, 477, 479
Bubishi (Patrick McCarthy), 42
built-in database schemas, 269
built-in functions, T-SQL, 37
business key, 282

C

CACHE <some value> function, 377
- NO CACHE vs., 377

CALLED ON NULL INPUT, 506
calling other stored procedures, 482
Cantor, Georg, 4
Cartesian product (of two input tables), 102
CASE expression and related functions, 49–53
case scenarios, 363–364, 405–406
- filtering and sorting data, 95
- queries and querying, 24
- SELECT statement, 56–57
- T-SQL, 24
  - code reviewer position, interviewing for a, 24
  - theory, importance of, 24

case sensitivity
- in XML, 223
- of XQuery, 236

CAST function, 3, 40–41, 68, 439
- SELECT INTO statement, 336

catalogs, full-text
- backup and restore of, 215–216
- as container for full-text indexes, 194
- creating, 194–200
- syntax for creating, 195

CATCH block, 438, 440–444, 448, 464
- error handling with, 474
- error reporting, 442

CDATA section (XML), 223
CHANGE_TRACKING [=] { MANUAL | AUTO | OFF [, NO POPULATION] } option, 196
character data, filtering, 68–69
character functions, 46–49
- concatenation, 46–47
- string formatting, 49
- string length, 48
- substring extraction/position, 47–48

character strings, 37, 40
- Unicode, 37

CHAR data type, 37, 39–40
- full-text indexes on columns of, 192

CHARINDEX function, 48
check constraints, 286–287
CHECK constraint violation, 373
child:: axis (XQuery), 240
CHOOSE function, 52
cleanup with unpivoting, 168
CLOSE command, 602
cloud computing, 3
CLR (Common Language Runtime), 530
- assemblies, synonyms used for, 316
- routines, 501
- stored procedures, 470

clustered indexes, 550, 555–564, 574, 582, 615, 618, 635
- table, 556

Clustered Index Scan iterator, 633–634, 658, 668
Clustered Index Scan operator, 591
Clustered Index Seek, 635
clustered table, 550, 633, 635
clustering key, 566, 572
COALESCE expression, 397
COALESCE function, 47, 51, 65
Codd, Edgar F., 4, 9
code
- T-SQL, 435

code reviewer position, interviewing for a, 24
coding standards, 3
colon (:), 227, 237
column aliases, 178
column identifiers, pivot queries and, 164
column operator value, 65
columns
- choosing data types for, 272–273
- computed, 274
- constraints and computed, 284
- as elements, 229
- modifying, 277
- naming, 270–272
- synonyms referring to, 323

columnstore indexes, 648, 656

Columnstore Index Scan operator, 657
COLUMNS_UPDATED(), 496
combining sets, 101–148, 144
answers to review questions, 145–148
APPLY operator and, 128–132
case scenarios, 143–144
with joins, 102–117
subqueries and, 118–121
suggested practices, 144
table expressions and, 121–128
and using set operators, 136–143
comma (,)
grouping sets separated by, 155
in pivot queries, 164
separating multiple CTEs by, 125
specifying multiple clauses with, 159
between table names, 104
commands, 415
comment() (node type test), 241
COMMIT (TRAN, TRANSACTION or WORK) command, 415–419, 422, 429
common table expressions. *See* CTEs (common table expressions)
common table expressions (CTEs). *See* CTEs (common table expressions)
compare old and new features, 406
composable DML, 399–400
composable DML, using, 402–403
composite key, 564
compression, 275
computed columns, 274
constraints and, 284
concatenation, 46–47
CONCAT function, 46
concurrency, 412–435
managing, 412–435
consistency, 413
constant monitoring, 545
constraints, 281–292
check, 286–287
data types as, 38
default, 288
foreign key, 285–286, 289–290
modification statements and, 331
primary key, 282–283
unique, 283–284, 291
using, 281–282
working with, 293
CONTAINS predicate, 202–203, 210
FREETEXT predicate vs., 204
language_term with, 210
CONTAINSTABLE function, 209
language_term with, 210
CONTINUE statement, 477, 479
control flow statements, 477
GOTO, 477
IF/ELSE, 477
RETURN, 477
WAITFOR, 477
WHILE, 477
CONVERT function, 3, 40–41, 70, 439, 483–484
SELECT INTO statement, 336
TRY_CONVERT vs., 439
correlated subqueries, 119–121
correlations, 119
COUNT_BIG aggregate function, 569
COUNT function, 165, 172, 239
as aggregate function, 152
COUNT(*) vs., 153
COUNT(*) function, 150, 153, 165
covered queries, 577
CPU, 648, 656
consumption, 541
CREATE AGGREGATE statement, 471
CREATE FUNCTION privileges, 505
Create Indexed Views article, 569
CREATE INDEX statement, 412, 558, 566, 578
CREATE PROCEDURE (or PROC) statement, 473, 476
using RECOMPILE query hint, 669
CREATE SEQUENCE command, 374
CREATE statement, 471, 473
CREATE statistics command, 586
CREATE SYNONYM statement, 315–317
CREATE TABLE statement, 231, 267–268, 274–275, 301, 412
and constraints, 281
as DML trigger, 491
CREATE VIEW statement, 300, 305
basic syntax for, 301–302
creating a sequence
using nondefault options, 379–381
using default options, 378–379
CROSS APPLY operator, 129–131
OUTER APPLY vs., 131
cross-column density statistic, 589
cross-database queries
using synonyms to simplify, 320–321
cross-database transactions, 421

CROSS JOIN command, 102–104, 306
cross join, explicit, 350
cross join, implied, 350
CTEs, 360
  UPDATE statement and, 348–349
  updating data using, 354
CTEs (common table expressions), 4, 124–127, 383, 388, 392, 617, 622
  defining multiple, 125
  recursive form of, 126
  using, 621–622
  window aggregate functions, 175
CUBE clause, 156, 159
current date and time, 44
CURRENT_TIMESTAMP function, 44
cursor-based solution, 607
cursor/iterative solutions, 600–611
  set-based vs., 600–611
cursors, 6, 8
  case scenario, 624–625
  compute aggregate using, 608–609
  options, 602
  performance improvement for, 625
  suggested practices, 626
  types, 602
  WHILE statement, 478
custom coding, 440
CYCLE | NO CYCLE property, 374

D

Darwen, Hugh, 4
data, 38
  improving process for updating, 364
  inserting, 330–341
  modifying, 369–410
  updating, 341–355
data access layer, 471
data accessor functions functions (XQuery), 238
data analysis functions, 149
data analysis operations, 149–190
  grouping, 150–162
  pivoting/unpivoting, 163–171
  windowing, 172–184
database administrator (DBA), 523, 594
databases
  backing up, 266
  querying, 266
database schemas
  built-in, 269
  nesting and, 270
  specifying, 269–270
  table schemas vs., 269
database tables. *See* table(s)
database users, authorized, 270
data changes
  T-SQL, 435
DataColumn, 525
data definition language. *See* DDL (data definition language)
data definition languages (DLLs), 470
data, deleting, 356–363
  based on a join, 359
  DELETE statement, 357–358
  sample data, 356
  TRUNCATE statement, 358–359
  using table expressions, 360
data() function, 236
data integrity
  enforcing, 281–292
  relational model and, 38
  suggested practices, 294
DATALENGTH function, 48
"The Data Loading Performance Guide", 333
data manipulation language (DML). *See* DML (data manipulation language)
data model, using predicates to define, 5
data modification statements, 414, 435
data type(s)
  choice of, for keys, 41–44
  choosing appropriate, 37–41
  column, 272–273
  fixed vs. dynamic, 39
  imprecise, 39
  regular vs. Unicode, 40
  size of, 42
  XQuery, 238
data warehouses, 301
data warehousing scenarios, 382
DATEADD function, 45
date and time data types, 38
  filtering, 70–71
date and time functions, 44–46
  addition and difference functions, 45
  current date and time, 44
  offsets, 45–46
  parts, date and time, 44–45

Date, Chris, 4, 6
DATE data type, 38, 70–71, 336
 columns, 273
DATEDIFF function, 45
DATEFORMAT, 70
DATEFROMPARTS function, 45, 54
DATENAME function, 45
DATEPART function, 44
DATETIME2 data type, 38, 44, 70
 columns, 273
DATETIME2FROMPARTS function, 45
DATETIME data type, 38, 44, 56, 70, 336
DATETIMEFROMPARTS function, 45
DATETIMEOFFSET data type, 38, 44
DATETIMEOFFSETFROMPARTS function, 45
DAY function, 44
DBCC CHECKIDENT command, 372
DBCC DROPCLEANBUFFERS command, 530–531
DBCC FREEPROCCACHE command, 648, 659
 production, 648
DBCC SHOW_STATISTICS command, 586–587
dbo database schema, 269
dbo.Fact table, 656
dbo schema, 614
dbo.sp_spaceused procedure, 553, 555
dbo.sp_spaceused system procedure, 552
dbo.TestStructure table, 552, 561, 570
dbo.Transactions, 608
dbo (user name), 270
DDL (data definition language), 250
 indexes and, 613–615
DDL statements, 412–414, 416–418, 435
 CREATE INDEX, 412
 CREATE TABLE, 412
deadlocking, 423–426, 429–431
 locking sequences, 423
 troubleshooting, 426
DEALLOCATE command, 602
DECIMAL data type, 273
declarative data integrity, 281
declarative plain language query, 601
DECLARE command, 602
DECLARE CURSOR, 602
DECLARE syntax, 473
DEFAULT constraint, 288, 375–376
default element namespace, 238
default initialization, 474
default language, changing the, 193
default options,creating a sequence using, 378–379
DEFAULT statement, 471
default values in tables, 273
deleted tables, 496–498
DELETE statements, 305, 357–358, 384, 388, 396–398, 412, 433, 456, 491, 493, 495, 504, 661
 join, DELETE based on, 359
 NOLOCK table hint and, 433
 OUTPUT clause and, 396
 synonyms with, 316
 TRUNCATE vs., 358, 364
 using table expressions, 360
 without WHERE clause vs. Truncate, 372
delimited identifiers, 271–272
delimiters
 identifiers, 34
 required vs. optional, 34
DENSE_RANK function, 177
 RANK vs., 177
deprecated rules, 281
derived tables, 122–124, 266, 360
 nesting of, 124
 subqueries vs., 122
 UPDATE statement and, 348
descendant:: axis (XQuery), 240
descendant-or-self:: axis (XQuery), 240
Detecting and Ending Deadlocks article, 426
deterministic functions, 274
 ORDER BY clause, 82
deterministic queries, 96
developer position, interviewing for, 185
diacritics_sensitive element, 194
difference time functions, 45
Disallow Results From Triggers, 493, 497
Discard Results After Execution, 606
discounts, 343
disk I/O, 648
DISTINCT clause, 7, 78
 in general set functions, 153
distributed partitioned views, 306
distributed transactions, 421
 local vs., 421
distribution statistics, 618
DML (data manipulation language), 305, 412–413, 416–418, 440
 composable, using, 402–403
 DELETE, 412

INSERT, 412
UPDATE, 412
DML statements, 496, 501
DML triggers, 491–492
AFTER, 491
functions, 496
INSTEAD OF, 491
writing, 496–499
DMOs (Dynamic Management Objects), 539–546
about, 539–540
categories, 540–542
document properties
full-text queries for searching on, 194
dollar sign ($), 34, 271
DROP FULLTEXT CATALOG statement, 195
DROP statement, 456, 471, 473, 502
DROP STATISTICS command, 586
DROP SYNONYM statement, 317
DROP VIEW, 305
duplicates, 33
durability, 413
dynamic batch, 612
dynamic data types, 39
dynamic management functions, 539
dynamic management view (DMV), 415
Dynamic Management Views And Functions articles, 545
Dynamic Management Views and Functions (Transact-SQL) article, 540
dynamic schema (XML data type for), 252–256
dynamic SQL
EXECUTE command, 454–468
overview, 451–455
parameterized, 651
sp_executesql, 457–458
usage, 450–462
uses for, 452

E

element namespace, default, 238
elements (XML), 222
attributes of, 224
columns as, 229
ELSE clause, 50
ELSE statement, 477
empty grouping set, 155, 159
encapsulate, 471
encapsulation, behavior, 38
ENCRYPTION, 506
END CATCH statement, 441
end tag (XML), 222
Enterprise edition of SQL Server 2012, 568
EOMONTH function, 45
equal (=) operator, 105, 242, 638
equijoins, 105, 639
error conditions, 435
error handling
implementing, 435–450, 463
store procedures and, 482
structured, 440, 448–449
unstructured, 440, 445–446
using XACT_ABORT, 446–447
ERROR_LINE function, 442
ERROR_MESSAGE function, 442
error messages
analyzing, 436
length limit on, 436
number, 436
severity level, 436
state, 436
T-SQL, 435
and Windows Application log, 436
error number, 436, 438
ERROR_NUMBER function, 442, 474
ERROR_PROCEDURE function, 442
errors
anticipating, 443
detecting, 435–440
handling after detection, 440–444
RAISERROR, 437
raising, 435–440
reporting, 442
structured handling using TRY/CATCH, 441–443
THROW, 438–439
TRY_CONVERT, 439–440
TRY_PARSE, 439–440
unstructured handling using @@ERROR, 440
using the CATCH block, 442
using XACT_ABORT with transactions, 441
ERROR_SEVERITY function, 442
ERROR_STATE function, 442
escalation, locks, 357
escaped values (XML), 223
Estimated Execution Mode, 655

ETL process, 568
ET STATISTICS TIME, 531
evaluation, of FROM clause, 16
EventCategory (SQL Trace/SQL Server Profiler), 524
EventClass (SQL Trace/SQL Server Profiler), 524
eventdata() function, 250
Event (SQL Trace/SQL Server Profiler), 524
exact numeric data types, 37
EXCEPT operator, 140–141, 306
  using, 141
excluded middle, law of, 9
exclusive locks, 422–423
EXEC command, 454, 457–458, 461
EXEC sys.sp_help_fulltext_system_components 'filter' query, 192
EXECUTE AS, 506
EXECUTE statement, 452, 470, 475, 501
  dynamic SQL, 454–455
  synonyms with, 316
execution plans, 532–536, 539, 545
  analyze, 672
  analyzing, 645–646
  icons, 641
  operators, 641
  prediction, 644–645
Execution-related DMOs, 540
exist() method (XML data type), 250–251
EXISTS predicate, 120
  negation of, 121
expansion element, 194
expansion words, 194
explicit cross join, 350
explicit inner join, 350
EXPLICIT option (FOR XML clause), 229
explicit transactions, 416, 435
  implicit vs., 422
explicit transactions mode, 418–419
expressions
  defining columns as values computed based on, 274
  table, 266
Extended Events. *See* SQL Server Extended Events
extended stored procedures, 470
extents, 550
external aliasing, 124
external fragmentation, 561

F

FAST_FORWARD option, 602
fatal error, 442
Features Supported by the Editions of SQL Server 2012 article, 569
FETCH NEXT command, 601–602
fields, 10
FILLFACTOR option, 558
filtered statistics, 589
filtering data, 61
  answers to review questions, 97–100
  case scenarios, 95
  character data, 68–69
  date and time data, 70–71
  full-text search, 192
  in grouped queries, 151
  with OFFSET-FETCH, 88–90
  performance recommendations, 95
  with predicates, 62–74
  suggested practices, 96
  with TOP option, 84–87
  views, 307
filtering rows
  based on the HAVING clause, 18–19
  based on the WHERE clause, 17
filters, 525
  explicit cross joins with, 350
fine-grained locks, 357
FIRST keyword, 88
FIRST_VALUE function, 179–180
fixed data types, 39
flags, 231
FLOAT (data type), 37, 39
  typecasting issues with, 39
FLWOR expressions (XQuery), 239, 243–245
FLWOR statement, 243
fn_FilteredExtension, 509
fn namespace, 236
force plans, analyze, 672
FOR clause with pivot queries, 164
foreign key constraints, 285–286, 289–290
foreign key—unique key relationships, 106
For (FLWOR statement), 243
FORMAT function, 49
FORMATMESSAGE function, 437–438
formatting
  RAISERROR, 437

string, 49
type vs., of value, 38
FOR statement, 492
FOR XML AUTO option, 227–229
FOR XML clause, 222–235
examples using, 233–234
FOR XML RAW option, 226–227
FOX XML PATH option, 229–230
fragmentation, 561
framing with window aggregate functions, 174
FREETEXT predicate, 204
CONTAINS predicate vs., 204
language_term with, 210
FREETEXTTABLE function, 209
FROM clause, 383, 388, 501, 503–504
SELECT statements and, 388
UPDATE statement with, 350
FROM statement, 30–31, 452, 455, 519–548, 663–664
GROUP BY clause and, 153
in logical query processing phases, 16
with ROW_NUMBER clause, 123
SELECT clause and, 31
UNPIVOT operator and, 166
and WHERE clause, 106
FULL backup, 484
full logging, 337
FULL OUTER JOIN keywords, 112
full-text data queries, 191–220
case scenarios, 215
catalogs and indexes, creating, 192–201
catalogs and indexes, managing full-text, 194–196
components, 192–194
CONTAINS predicate for, 202–203
FREETEXT predicate for, 204
full-text search functions, 209–210
semantic search functions, 210–211
suggested practices, 215–216
full-text search functions, 209–210
dynamic management views, 215–216
example using, 211–212
functions
aggregate, 150, 306
data analysis, 149
deterministic, 274
inline, 307–313
standard vs. nonstandard, 3
user-defined, 316, 501–510
XQuery, 238–239
FUNCTION statement, 471

G

general comparison operators, 242
generating keys, improved solution for, 405
generating T-SQL strings, 453–454
generation terms (in searches), 192
GEOGRAPHY types, 530
GEOMETRY types, 530
GETDATE function, 44
GetNums function, 604
GETUTCDATE function, 44
globally unique identifiers (GUIDs), 43, 562–563
nonsequential vs. sequential, 42
global vs. local temporary tables, 612–613
GO delimiter, 455
GO statements, 419
GOTO construct, 481
GOTO statement, 477
Graphical Execution Plan Icons (SQL Server Management Studio) article, 534
greater than (>), 223, 242
greater than or equal to (>=), 71, 242
GROUP BY clause, 452, 519, 575–576, 642
explicit, 151
grouped queries and, 150
in logical query processing phases, 17–18
order of evaluation, 153
workarounds with, 154
GROUP BY statement, 399
grouped queries, 150–162
without explicit GROUP BY clause, 150
tables, defining with, 149
uses for, 150
writing, 159–161
group functions
grouped queries and, 150
window functions vs., 172
grouping
fixing problems with, 21
with multiple grouping sets, 155–161
and pivoting/unpivoting data, 163–171
pivoting as specialized form of, 149
with single grouping set, 150–154
GROUPING function, 157
GROUPING_ID function, 158
grouping rows (based on GROUP BY clause), 17–18
grouping sets, 150
empty, 155, 159
multiple elements in, 151

GROUPING SETS clause, 155, 159, 161
guaranteed order, 75–76
guarantees, 75–76
guest database schema, 269

H

handling errors after detection, 440–444
hash aggregation, 663
  stream aggregation vs., 643
hash (as join algorithm), 519–548
hash function, 640
hash join, 640, 643, 655
Hash Match Aggregate operator, 643
Hash Match iterator, 655
Hash Match Join iterator, 643
Hash Match operator, 655, 657
HAVING clause, 62, 151, 452
  GROUP BY clause and, 153
  in logical query processing phases, 18–19
HAVING statement, 399
heading (of relation), 4, 6
heap(s), 550–563, 572, 634, 636
  allocation, 555
  allocation check, 552
hints
  defined, 631
  HOLDLOCK, 386
  SERIALIZABLE, 386
hints (joins), 664
  details, 665
  HASH, 664
  LOOP, 664
  MERGE, 664
  REMOTE, 664
hints (SQL Server Query Optimizer), 661–666
  { CONCAT | HASH | MERGE } UNION, 662
  details, 662
  EXPAND VIEWS, 662
  FAST number_rows, 662
  FORCE ORDER, 662
  { HASH | ORDER } GROUP, 662
  IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX, 662
  KEEPFIXED PLAN, 662
  KEEP PLAN, 662
  { LOOP | MERGE | HASH } JOIN, 662
  MAXDOP number_of_processors, 662
  MAXRECURSION number, 662
  OPTIMIZE FOR UNKNOWN, 662
  OPTIMIZE FOR (@variable_name { UNKNOWN | = literal_constant } [,...n], 662
  PARAMETERIZATION { SIMPLE | FORCED }, 662
  RECOMPILE, 662, 669
  ROBUST PLAN, 662
  TABLE HINT (exposed_object_name [, <table_hint> [[,]...n]], 662
  use, 661
  USE PLAN N'xml_plan, 662
hints (tables), 663
  case scenario, 671
  details, 664
  FORCESCAN, 663
  FORCESEEK, 663
  HOLDLOCK, 663
  IGNORE_CONSTRAINTS, 663
  IGNORE_TRIGGERS, 663
  INDEX (index_value [,...n]) | INDEX = (index_value) | FORCESEEK [(index_value (index_column_name [,...]))], 663
  KEEPDEFAULTS, 663
  KEEPIDENTITY, 663
  NOEXPAND, 663
  NOLOCK, 663
  NOWAIT, 663
  PAGLOCK, 663
  READCOMMITTED, 664
  READCOMMITTEDLOCK, 664
  READPAST, 664
  READUNCOMMITTED, 664
  REPEATABLEREAD, 664
  ROWLOCK, 664
  SERIALIZABLE, 664
  SPATIAL_WINDOW_MAX_CELLS = integer, 664
  TABLOCK, 664
  TABLOCKX, 664
  UPDLOCK, 664
  XLOCK, 664
histograms, 618
HOLDLOCK hint, 386
hypercubes. *See* Star schema

I

IAM pages, 551–553, 555–556, 632
IDENT_CURRENT function, 358, 371–372
  DELETE statement, 358
identifiers
  delimited, 271–272
  delimiting, 34
  regular, 34, 271–272
IDENTITY column property, 42, 376–378, 395, 406
  DELETE statement without WHERE clause in, 372
  INSERT EXEC statement and, 334
  INSERT SELECT statement, 333
  INSERT VALUES statement and, 331
  limitations of, 378
  SELECT INTO statement, 336
  sequence numbers and, 273–274
  sequence object vs., 374
  TRUNCATE statement in, 372
  using, 370–373
IDENTITY_INSERT option, 331, 333, 373
  INSERT EXEC statement, 334
IF clause, 440
IF/ELSE construct, 477–478
IF/ELSE statement, 477
IF @errnum clause, 447
IF statement, 477
if..then..else, 242
IIF function, 52
IMAGE data type, 530
  full-text indexes on columns of, 192
implementing error handling, 435–450, 464
implementing nonclustered indexes, 564–568
implementing transactions, 428–433
implementing triggers
  case scenario, 511–512
implicit transactions, 416, 428
  advantages, 417
  disadvantages, 417
  explicit vs., 422
implicit transactions mode, 416–418
implied cross join, 350
imprecise data types, 39
improving modifications, 405–406
IN clause, 452
  of PIVOT operator, 166
  with pivot queries, 164
Include Actual Query Plan option, 618
INCLUDE clause, 578
inclusive operators, 580
INCREMENT BY property, 374
indexed views, 266, 304
  implementing, 568–570
indexes, 106, 550–573, 639
  clustered, 550, 555–563, 635
  columnstore, 648, 656
  DDL and, 613–615
  full-text, 192
  implementing, 568–570
  nonclustered, 551, 634–635, 646
  search arguments, 573–584
  supporting queries, 574–578
  table, 276
  XML, 256
indexes, creating full-text, 194–200
  examples, 196–200
indexes, full-text
  backup and restore of, 215–216
  installing a semantic database and creating a, 200
  syntax for creating, 195
index idx_nc_orderdate, 621
index keys, 564
index leaf level, 633
index-related DMOs, 541
  using, 542–543
Index Scan iterator, 634
Index Scan (NonClustered) operator, 591
Index Seek operator, 583, 636, 658–659
index usage, 574
inequality operator, 388
infinite loop, 478
inflectional forms, 204
INFORMATION_SCHEMA schema, 269
IN() function, 455
inline aliasing, 124
inline functions, 307–313
  converting views into, 312–313
  options with, 309–313
  suggested practices, 324
inline TVFs vs. views, 127–128
inner batch, 612
inner join, explicit, 350
Inner Join operator, 655, 657
inner joins, 105–108
inner queries
  with table expressions, 121
  with CTEs, 125
IN operator, 579

IN PATH option, 195
input elements, hierarchy of, 156
input parameters, 470, 475–476
INSERT, 305
  synonyms with, 316
inserted tables, 496–498
INSERT EXEC statement, 334–335
inserting data, 330–341
  for customers without orders, 338
  INSERT EXEC statement, 334–335
  INSERT SELECT statement, 333
  INSERT VALUES statement, 331–332
  sample data, 330–331
  SELECT INTO statement, 335–337
INSERT SELECT statements, 333, 375–376, 399–400
  INSERT EXEC vs., 334
Insert Snippet menu (SSMS), 492, 502
INSERT statements, 371, 377, 384, 386, 392, 395, 397–400, 412, 445, 448, 486, 491–497, 661
  failure of, 373
  OUTPUT clause, 395–396
  query functions in, 371
INSERT VALUES statements, 331–332, 375–376
  INSERT EXEC vs., 334
  INSERT SELECT vs., 333
INSTEAD OF trigger, 491–492, 495–496
INT data type, 37–38, 41–42, 67
INTEGER data type, 652
intelligent keys, 41
intent locks, 422
internal fragmentation, 561
International Organization for Standards (ISO), 3
INTERSECT operator, 139, 306
  EXCEPT operator vs., 140
  using, 142
INTO clause, 394, 396
invoicing systems, 373
I/O
  data size and, 38
  statistics, 530
ISNULL function, 51–52, 65
  SELECT INTO statement, 336
isolation (ACID property), 413
isolation level(s), 414, 540, 633
  common, 431
  READ COMMITTED, 426
  READ COMMITTED SNAPSHOT, 426
  READ UNCOMMITED, 426
  REPEATABLE READ, 427
  SERIALIZABLE, 427
  SNAPSHOT, 427
  SQL server row versioning and, 433
  transactions, 426–433
is_user_process flag, 540
iterative constructs, 6
iterative solutions, 601–604

J

join
  DELETE based on, 359
JOIN clause, 575
Join Hints (Transact-SQL) article, 665
JOIN keyword, 108
JOIN operator, 359, 388
  DELETE statement based on, 359
join predicate, 638
joins, 102–117
  algorithms used for, 638–641
  aliasing tables in, 104
  APPLY operator and, 128
  cross, 102–104
  deleting data using, 361
  equi-, 105
  explicit cross, 350
  explicit inner, 350
  hash, 640, 643, 655
  hints, 661, 664
  implied cross, 350
  inner, 105–108
  merge, 639, 644, 665
  multi-join queries, 112–114
  nested loops, 638, 665
  non-equijoin, 638
  outer, 108–112
  self-, 104, 107
  updating data using, 353
junior developer, tutoring a, 95

K

Kejser, Thomas, 43
KEY column, 209
keyed-in order, 15
KEY INDEX index_name option, 196

Key Lookup operator, 575, 577, 637, 644–645, 658–659
keys, 564
- choice of data type for, 41–44
- intelligent, 41
- nonsequential, 43
- sequential, 43
- surrogate, 41–42

keywords
- reserved, 34

KILL command, 442
Kutschera, Wolfgang 'Rick', 43

L

LAG function, 178–179
language, changing the default, 193
language ID, 194
languages, data in multiple, 40
language_term, 210
LAST_VALUE function, 179–180
law of excluded middle, 9
LEAD function, 178–179
leaf level pages, 556, 558
LEFT function, 47
LEFT OUTER JOIN keywords, 108
legacy RAISERROR command, 436
LEN function, 48
less than (<) operator, 71, 223, 242
less than or equal to (<=), 242
Let (FLWOR statement), 243
levels (of transactions), 415–416
like_i_sql_unicode_string, 526
LIKE operator, 578
LIKE predicate, 68–69
line-of-business (LOB) applications, 215
linked servers, objects referenced by, 317
literals, 40
literal types, 68
LOBs (large objects), 530
locale identifier (LCID), 210
local temporary tables, 612
- global vs., 612–613

local transactions
- distributed vs., 421

lock compatibility, 422–423
- exclusive locks, 423
- shared locks, 423

lock escalation, 357
locking, 414
- basic, 422–426
- blocking, 423
- compatibility, 422–423
- deadlocking, 423–426
- sequences, 423

locks
- advanced, 422
- escalation, 357
- exclusive, 422
- fine-grained, 357
- intent, 422
- rows, 357
- schema, 422
- shared, 422
- tables, 357
- update, 422

logging
- full vs. minimal, 337
- using modifications that support optimized, 364

logical CPUs, 540
logical fragmentation, 632
logical query processing, 14–24
- answers to review questions and case scenarios, 26–28
- phases of. *See* logical query processing phases
- review questions, 23–24
- suggested practices, 185
- summary, 23
- and T-SQL as declarative English-like language, 14–15

logical query processing phases, 15–23
- filter rows based on HAVING clause, 18–19
- filter rows based on WHERE clause, 17
- FROM clause, evaluating, 16
- group rows based on GROUP BY clause, 17–18
- ordering using the ORDER BY clause, 20–21
- processing the SELECT clause, 19–20

logical reads, 530
loops, 6
LOWER function (string formatting function), 49
LTRIM function (string formatting function), 49

M

marking transactions, 420–421
mathematical foundations of T-SQL, 2
max() function, 239
MAX function, 172
  as aggregate function, 152
  workaround using, 154
MAXVALUE property, 374
McCarthy, Patrick, 42
MERGE INTO statement, 383
merge join, 639–640, 644, 665
Merge Join iterator, 639, 645
MERGE statement, 394, 397, 406, 661
  OUTPUT clause and, 397–398
  role of ON clause in, 391
  UPDATE vs., 346
  usage, 390
merging data, 382–393
message, 438
message ID, 437
metadata, 473
  tables, 318
  views and, 306–307
  in XML, 224
Microsoft .NET, 253
Microsoft Office 2010, 193
Microsoft SQL Server 2012
  full-text search support in, 191
  XML support in, 221
Microsoft SQL Server 2012 High-Performance T-SQL Using Window Functions (Itzik Ben-Gan), 183
Microsoft Visual Basic, T-SQL vs., 3
Microsoft Visual C#, T-SQL vs., 3
Microsoft Windows Azure SQL Database, 3
MIN() aggregation function, 152, 172, 575
minimal logging, 337
MINVALUE property, 374–375
missing indexes, 543
missing values, 9
mixed extent, 550, 554
modes (of transactions), 416–419, 428–429
modification statements
  constraints defined in target table and, 331
  optimized logging supported by, 364
modifying data, 369–410
  IDENTITY column property, using, 370–373
  merging data, 382–393
  OUTPUT option, using, 394–404
  sequence object, using, 374–381
modify() method (XML data type), 250–251
MONTH function, 44, 79
MSDTC (Distributed Transaction Coordinator), 421
multicolumn statistics, 589
multi-join queries, 112–114
multiple CTEs, defining, 125
multiple grouping sets
  defining, 161
  working with, 155–161
multiple languages, data in, 40
multiple queries
  INSERT EXEC statement and, 335
multiple rows
  INSERT VALUES statement with, 332
multisets, 7
multiset theory, 7

N

names
  object, 272
  of views, 303–307
namespace(s), 224, 227, 236
  and database schemas, 269
  default element, 238
naming
  tables, 270–272
  two-part, 268
natural key, 282
NCHAR data type, 37, 39–40, 68
  full-text indexes on columns of, 192
negation of EXISTS predicate, 121
nested AFTER triggers, 494–495
nested elements (XML), 222
nested loops, 519–548
  algorithm, 638–639
nested loops join, 638, 640, 665
nested transactions, 418–420, 419–420
nested triggers, 494
nesting
  database schemas, 270
  of derived tables, 124
NEWID() T-SQL function, 43, 56, 505, 562
New Session UI, 523–548
New Session Wizard, 523–548

NEXT keyword, 88
NEXT VALUE FOR function, 375–376, 378
NO CACHE function, 377
    CACHE <some value> vs., 377
NOCOUNT, 570
    ON, 474
node() (node type test), 241
nodes functions functions (XQuery), 238
nodes in XML, 236
nodes() method (XML data type), 250–251
node test (Xquery navigation), 240–241
node types (XQuery), 238
noise words, 193
NOLOCK, 433
nonclustered indexes, 551, 577, 581, 588, 615, 634–635, 646
    analyzing, 570–572
    on a clustered table, 571–572
    on a heap, 570–571
    implementing, 564–568
    scanning, 583
    seeking, 636–637
nondefault options
    creating a sequence using, 379–381
nondeterministic ordering
    ORDER BY clause with, 81
nondeterministic queries, 96
nondeterministic UPDATE, 346–348
non-equijoin, 638
nonexistent objects, references to, 317
nonsequential GUIDs, 42
nonsequential keys, 43
nonstandard functions, 3
not equal (!=) operator, 242
    <> operator vs., 3
NOT (logical operator), 66
NOT NULL data type, 38
NOWAIT command, 439
NTEXT data type (Unicode), 530
    full-text indexes on columns of, 192
NTILE function, 177–178
NULL columns, 278
NULL data type, 38
NULLIF function, 50, 52
NULLs, 10, 47, 388, 397, 439, 476, 506, 566, 603
    comparing two, 65
    filtering rows with, using WHERE clause, 72
    general set functions and, 152
    in grouped queries, 151
    INSERT SELECT statement and, 333
    INSERT VALUES statement and, 332
    interaction of predicates with, 62
    INTERSECT operator and, 139
    with LEFT JOINs, 108
    ordering and treatment of, 80
    as placeholders, 157
    as placeholders in grouped queries, 156
    and primary keys, 283
    SELECT INTO statement, 336
    in tables, 273
    UNPIVOT operator and, 168
    XML data type and, 250
NULLS FIRST option, 80
NULLS LAST option, 80
number sign (#), 34, 271
NUMERIC data type, 37
    columns, 273
numeric data types
    approximate, 37
    exact, 37
numeric functions (XQuery), 238
numeric predicates, 241
NVARCHAR data type, 37, 39–40, 68, 478
    columns, 272
    full-text indexes on columns of, 192
NVARCHAR(MAX) data type, 457, 530
    columns, 273

O

OBJECT_ID() function, 305, 473, 493, 501
object names, length of, 272
OBJECT plan guides, 666
objects, 550
    allowed, for synonyms, 316
    linked servers, referenced by, 317
OFFSET-FETCH filter, 21, 306
    filtering data with, 88–90
    with inner queries, 121
offset functions, window, 178–180
offset of date/time functions, 45–46
offsets, 179
old features
    new vs., 406

OLTP (online transaction processing)
- environment, 648
- scenarios, 382

ON clause, 62
- join's matching predicate specified in, 106
- with LEFT JOINs, 109
- role in MERGE statement, 391
- WHERE clause vs., 106

ON FILEGROUP option, 195
on-line transactional processing applications. *See* OLTP applications
online transaction processing (OLTP), 301, 558, 563, 636. *See* OLTP (online transaction processing)
on-premise SQL Server, 427
ON statement, 384
OPEN command, 602
OPENROWSET function (OPENXML), 383, 388
- nodes() method vs., 251

OPENXML function, 231–232, 388
- flag parameter for, 231

operators
- general comparison, 242
- value comparison, 242

optimization, 14
- SQL Server and, 104
- of table expressions, 122

optional delimiters, 34
OPTION clause, 661, 666, 669
options
- views, 302

ORDER BY clause, 8, 452, 556, 574, 576, 600, 607
- DELETE statement using, 360
- with deterministic ordering, 82
- as FLWOR statement, 243
- GROUP BY clause and, 153
- in logical query processing phases, 20–21
- with nondeterministic ordering, 81
- OFFSET-FETCH with, 88
- and presentation vs. window ordering, 177
- with ROW_NUMBER function, 123
- and SELECT statement in views, 304
- sorting data with, 76–81
- TOP or OFFSET-FETCH option with, 121
- window aggregate functions, 175
- window functions allowed in, 178
- XML queries, 228

ORDER BY list, 376
ordered partial scan, 636–637
Ordered property, 633
ordered sets, 5
OR (logical operator), 66–67, 579–583
- support, 581–582

orphans, synonym, 317
OUTER APPLY operator, 131–132
- CROSS APPLY vs., 129

outer joins, 108–112
- matching customers and orders with, 115

outer queries (with CTEs), 125
OUTPUT clause, 394–395, 399, 406
- DELETE statement with, 396
- INSERT statement with, 395–396
- MERGE statement and, 398
- MERGE statement with, 397–398
- SELECT vs., 394
- UPDATE statement with, 397
- using in UPDATE statement, 401–402
- working with, 394–395

OUTPUT keyword, 474
output limits, SSMS and, 454
OUTPUT option
- using, 394–404

OUTPUT parameters, 458, 470, 474–477, 482
- sp_executesql with, 461–462

OUTPUT statement, 461, 476
OVER clause, 172, 376
overflow error, 373

P

PAD_INDEX option, 558
page, 550
page-level compression, 275
Parallelism iterator, 655
parameterization, 666
parameterized dynamic SQL, 651
parameterized propositions, 5
parameterized queries, 647–660
parameterizing queries, 650
parameters
- CATCH block, 443
- inline table-valued functions, 308
- input, 475–476
- output, 461–462, 476–477
- THROW command, 438

parent:: axis (XQuery), 240

parentheses (( )), 237, 308
  with grouping sets, 155
  with OVER clause, 172
PARSE function, 40–41, 70, 439
partial scan, 635
PARTITION BY actid, 607
partitioned views, 306
parts, date and time, 44–45
PATH option (FOR XML clause), 229–230
PATH (secondary XML index), 256
PATINDEX function, 48
peers, 176
PERCENT option, 85
performance considerations
  query filters and, 65
performance optimization
  COALESCE / ISNULL and, 66
performance recommendations, filtering and sorting, 95
permanent tables, views referencing, 304
permissions, 270
  and access control, 127
phases, logical query processing, 15
physical fragmentation, 632
physical memory, 540
physical processing, 14
physical reads, 530
PIVOT clause, 452
PIVOT() function, 455
pivoting
  as inverse of unpivoting, 149
  as specialized form of grouping, 149
pivoting data, 163
PIVOT operator, 163–166, 388
  limitations of, 165
pivot queries, 163
Plan Caching in SQL Server 2008 article, 650
plan guides, 666–668
  create, 667–668
  OBJECT, 666
  SQL, 666
  TEMPLATE, 666–667
plan iterators, 632–647
  access methods, 632–638
  join algorithms, 638–641
plus (+) operator, 38
predicate logic, 5
predicates, 5
  Boolean, 241
  combining, 66–68
  filtering data with, 62–74
  numeric, 241
  search arguments and, 62–66
  three-valued logic and, 62–66
predicate (Xquery navigation), 240
prefix terms (in searches), 192
prepare data, 536–537
presentation ordering
  ORDER BY clause for, 20–21
  window ordering vs., 177
PRIMARY KEY constraint, 282–283, 373, 614–615
  IDENTITY property using, 373
printf style formatting, 437
PRINT statement, 437, 441, 454, 459, 478, 481, 483–484, 602
probe phase, 640
procedure recompilation, 659
PROCEDURE statement, 471
processing-instruction() (node type test), 241
processing instructions (XML), 223
programming languages, 2
proof-of-concept (POC) projects. *See* POC projects
PROPERTY (secondary XML index), 256
proximity terms (in searches), 192

Q

QName (qualified name), 236
quadratic (N2) scaling, 607
qualified name (QName), 236
queries, 574–578. *See also* subqueries
  against CTEs, 124
  grouped, 150–162
  multi-join, 112–114
  parameterized, 647–660
queries and querying, 6–8
  answers to review questions, 26–28
  case scenarios, 24
  suggested practices, 25
query filters. *See* filters
query hints, 661
Query Hints (Transact-SQL) article, 662
querying
  databases, 266
  from views, 304

query() method (XML data type), 250–251
query optimization, 518–528
case scenario, 671
with parameterized queries, 648
problems, 518–522
with plan iterators, 632–647
Query Optimizer, 518–522, 578–580, 585, 589, 644
hints, 661–666
query parameterization
with batch processing, 653–658
exercise, 658
query performance
case scenario, 544–545
suggested practices, 545
query plans
analyzing, 529–539
reusing, 631
question mark (?), 223
quotation mark ("), 223, 271
single, 40
QUOTED_IDENTIFIER setting, 453
QUOTENAME function, 454
generate, 458–459
T-SQL strings and, 458–459

R

RAISERROR command, 436–441, 448, 491, 496
formatting, 437
simple form, 437
THROW vs., 438
THROW vs., in TRY/CATCH, 443
RAND() function, 505
random key generators, 43
RANGE option, 180
RANGE window frame extent, 176
RANK column, 209
RANK function, 177
DENSE_RANK vs., 177
ranking functions, window, 176–178
rank value, 209
RAW option (XML), 226–227
RCSI (READ COMMITTED SNAPSHOT isolation level), 426
read-ahead reads, 530
READ COMMITTED isolation level, 423, 426–428, 433
READ COMMITTED SNAPSHOT isolation level, 426–428, 433
READ COMMITTED statement, 431
READCOMMITTED table hint, 433
read-only environment, 633
read-only transactions, 412
read performance, 41
Read Uncommitted, 633
READ UNCOMMITTED isolation level, 426, 432, 634–635
READ UNCOMMITTED statement, 432
REAL data type, 37, 39
REBUILD, 563
records, 10
RECOVERY statement, 421
recursive queries, 126
regular data types, 40
regular identifiers, 34, 271–272
relational data
producing XML from, 226–230
relational database management system (RDBMS), 3, 253, 412
relational model, 4
data integrity and, 38
relation (mathematical concept), 4
heading and body of, 4
tables vs., 4
renaming, 32
REORGANIZE, 563
REPEATABLE READ isolation level, 427
REPLACE function, 48, 484
replacement element, 194
replacement words, 194
REPLICATE function, 49, 54
reporting
building views for, 310
synonyms and descriptive names for, 319–320
required delimiters, 34
reserved keywords, 34
results of views, ordering, 304
RETURN clause, 128
return codes, 474
Return (FLWOR statement), 243
RETURNS NULL ON NULL INPUT (UDF option), 506
RETURNS TABLE
inline table-valued functions, 308
RETURN statement, 474, 477, 482, 492, 503
inline table-valued functions, 308
RID Lookup, 636
RID Lookup operator, 564, 636–637

RID (row identifier), 564, 572, 575, 577, 634, 636
RIGHT function, 47
RIGHT OUTER JOIN keywords, 111
roll back (of transaction), 617
ROLLBACK (TRAN, TRANSACTION or WORK) statement, 415–419, 421–422, 429, 491
ROLLUP clause, 156, 159
row-by-row operations, 601
row identifier. *See* RID (row identifier)
row-level compression, 275
row locator, 564
row locks, 357
ROW_NUMBER function, 123, 177, 181
ROW/ROWS, 88
  RANGE clause vs., 176, 180
rows
  DELETE statement and, 357
  filtering, based on the HAVING clause, 18–19
  filtering, based on WHERE clause, 17
  grouping, based on GROUP BY clause, 17–18
  locks, 357
  ranking, 176
ROWS UNBOUNDED PRECEDING, 175, 607
ROWVERSION data type, 273
row versioning, 427
RTRIM function, 49
rules, deprecated, 281
RULE statement, 471

S

SARGs. *See* search arguments (SARGs)
Savepoints, 421
SAVE TRANSACTION command, 421
scalar subqueries, 118
scalar UDFs, 502–503
  writing, 507–508
scan count, 530
SCHEMABINDING option, 506, 569
schema locks, 422
schema name (of table), 30
schemas, database, 269–270
scope, 612–613
SCOPE_IDENTITY function, 371
  @@IDENTITY vs, 372
search arguments (SARGs), 65, 578–580
  predicates and, 62–66
search condition, 631
searches
  enhancing, 215
security
  database, 471
seek, 564
SELECT clause, 8, 19–20, 31–33, 394, 452, 455
  GROUP BY clause and, 153
  in logical query processing phases, 19–20
  OUTPUT clause vs., 394
  with ROW_NUMBER function, 123
  UPDATE based on join and, 344
  window aggregate functions, 175
  window functions allowed in, 178
SELECT FROM statement
  with views, 307
SELECT INTO statement, 267, 335–337
  using, 339
selective query, 581
SELECT phase (window aggregate functions), 175
SELECT query, 383, 594
  DELETE statement based on, 360
  in SQL vs. T-SQL, 33
  two main roles of, 30
SELECT statement, 29–60, 255, 388, 412, 416, 425–427, 431–433, 443, 452, 454, 456, 502–503, 506, 508, 661
  answers to review questions, 58–60
  CASE expression and related functions and, 49–53
  character functions and, 46–49
  and choice of data type, 37
  and choice of data type for keys, 41–44
  date/time functions and, 44–46
  delimiting identifiers and, 34–35
  FOR XML clause of, 222
  FROM clause and, 30–31
  inline table-valued functions, 308
  review, lesson, 55–56
  SELECT clause and, 31–33
  suggested practices, 57
  summary, lesson, 55
  synonyms with, 316
  UPDATE and, 350–351
  UPDATE based on join and, 344
  UPDATE statement and, 348
  in views, 303
  views defined by, 300
  without a FROM clause, 385
self:: axis (XQuery), 240

self-contained subqueries, 118–119
self-documenting views, 302
self-joins, 104, 107
semantic database, installing, 200
semantic key phrases, 192
SEMANTICKEYPHRASETABLE function, 210
Semantic Language Statistics Database, 196
semantic search, 196
  table-valued functions and, 210
  using, 215
semantic search functions, 210–211
  example using, 213
SEMANTICSIMILARITYDETAILSTABLE function, 211
SEMANTICSIMILARITYTABLE function, 211
semicolon (;), 4, 223, 308
SEQUEL, 14
sequence numbers
  Identity property and, 273–274
sequence object, 42, 377
  IDENTITY column property vs., 374
  using, 374–378
sequences (XQuery), 236
sequential GUIDs, 42
sequential keys, 43
SERIALIZABLE hint, 386
SERIALIZABLE isolation level, 427
Service Broker, 481
set-based solutions, 600–611
  compute an aggregate using, 609–610
  cursor/iterative vs., 600–611
  cursor vs., 604–608
  set theory, 600–601
SET clause, 375
  UPDATE based on join and, 344
  UPDATE statement and, 342, 350–351
SET IDENTITY_INSERT <table> session option, 371
SET IMPLICIT_TRANSACTIONS article, 422
set operators, 136–143
  EXCEPT operator, 140–141
  explaining, 144
  general form of code using, 136
  guidelines for working with, 137
  INTERSECT operator, 139
  UNION/UNION ALL operators, 137–139
SET QUOTED_IDENTIFIER, 453
sets. *See also* combining sets
  case scenario, 624–625
  combining, 144
  grouping, 150
  ordered, 5
  suggested practices, 626
SET session options, 529–539
SET SHOWPLAN_ALL command, 532
SET SHOWPLAN_TEXT command, 532
SET SHOWPLAN_XML statement, 250, 532
SET statement, 478
SET STATISTICS IO T-SQL command, 529
SET STATISTICS PROFILE command, 532
SET STATISTICS TIME (session command), 531
SET STATISTICS XML statement, 250, 532
set theory, 4–5, 75
SET XACT_ABORT, 441
severity levels, 436–437
shared locks, 422–423
short circuits, 67
Showplan Logical and Physical Operators Reference article, 545, 641
side-by-side sessions, 428
side effect, 505
simple terms (in searches), 192
single-column statistics, 589
single data modification, 416
single grouping set
  workarounds, 154
  working with, 150–154
single quotation marks, 40
size
  data, 38
  of data type, 42
slash (/) character, 229, 240
slow updates, 594
SMALLDATETIME data type, 38, 70
SMALLDATETIMEFROMPARTS function, 45
SNAPSHOT isolation level, 427
sorting data, 61, 74–84
  answers to review questions, 97–100
  case scenarios, 95
  and guaranteed order, 75–76
  with ORDER BY, 76–81
  performance recommendations, 95
  suggested practices, 96
Sort operator, 576, 641–642, 663
sought keys, 637
spaghetti code, 481
sparse data, 250
sp_configure procedure, 475
  Disallow Results From Triggers option, 493

sp_estimate_data_compression_savings stored procedure, 275
sp_executesql (system stored procedure), 452, 457–458
  output parameters, 457, 461–462
  parameters, 457
  stored procedure, 459
spreading elements, 165
sp_sequence_get_range procedure, 377
SQL and Relational Theory (Date), 6
sql_handle, 541
SQL identifier length, 451
SQL Injection attacks, 456–457
  article, 456
  prevention, 459–461
SQLOS, 540
SQL plan guides, 666
SQL Server 2012, 3, 43, 56, 65, 67, 435, 453, 493, 501, 517, 523
  data types supported by, 37
  Extended Events, 523–525
  features, 569
  filtering character data, 68
  generating T-SQL strings in, 453
  and optimization, 104
  profiler, 523–525
  RAISERROR in, 437
  registering filters in, 193
  transaction durability and, 414
SQL Server 2012 Express edition, 666
SQL Server 2012 instance, 632
SQL Server Database Engine, 223
sqlserver.database_name, 526
SQL Server Extended Events, 518–548, 523–525, 539, 545
  article, 545
  creating sessions, 525–527
  SQL Trace/SQL Server Profiler vs., 525
  usage, 527
SQL Server Extended Events Live Data, 527
SQL Server Extended Events objects, 523–548
  actions, 523
  events, 523–548
  maps, 523
  predicates, 523
  targets, 523–548
  types, 523
SQL Server extension functions (XQuery), 238
SQL Server Integration Services (SSIS), 196
SQL Server Management Studio (SSMS), 270, 306, 401, 424, 428–430, 444–446, 454, 458, 502, 517, 632
SQL Server Profiler, 523–525, 539
SQL Server Query Optimizer, 304, 306, 518–548, 566, 568, 573, 607, 634
  hints, 634–672
  plan guides, 661–671
SQL server row versioning, 433
  isolation levels, 433
sqlserver.sql_text, 526
sql_statement_completed event, 526
SQL Trace, 523–525
  DMOs and, 539
sqltypes namespace, 236
square brackets ([]), 48, 271
standard vs. nonstandard functions, 3
Star join optimization, 519, 653
star schema, 653
START WITH property, 374, 375
state, 436, 438
statement_end_offset, 541
statement_start_offset, 541
statements, termination of, 4
states, transactions, 415–416
statistical semantic search, 192
STATISTICAL_SEMANTICS option, 196
statistics, 585–593
  auto-created, 585–589
  auto-creation, disabled, 591–592
  case scenarios, 593–594
  disable auto-creation, 590
  filtered, 589
  manually maintaining, 589–590
  multicolumn, 589
  single-column, 589
  suggested practices, 594
  temporary tables and, 618–620
  updating, 586
STATISTICS IO, 569–570, 619, 657
STATS_DATE() system function, 588
stemmers, 193
stemming, 204
STOPATMARK statement, 421
STOPBEFOREMARK statement, 421
stoplists, 193
stopwords, 193

stored procedures, 435, 470–490, 612, 666
- about, 470–474
- advantages, 471
- calling, 482
- case scenario, 511
- create, 483–486
- designing, 470–490
- developing, 481–483
- dynamic SQL in, 482–483
- error handling, 482
- executing, 475–477
- existence, 473
- implementing, 470–490
- parameters, 473–474
- recompilation, 659
- results, 482
- suggested practices, 512
- synonyms used for, 316
- testing for the existence of, 473
- VIEW statements and, 471

Stream Aggregate iterator, 642, 646
Stream Aggregate operator, 577, 642, 663
stream aggregation, 643
string functions (XQuery), 238
strings, 437
- character, 40
- formatting, 49
- length, 48
- literals, 455
- T-SQL, 458–459
- variables, 455

structured error handling, 440, 448–449
- using TRY/CATCH, 448–449

STUFF function, 49
subqueries, 118–121
- correlated, 119–121
- derived tables vs., 122
- scalar, 118
- self-contained, 118–119
- table-valued, 118

substring extraction and position, 47–48
SUBSTRING function, 47
suggested practices
- combining sets, 144
- filtering and sorting data, 96
- queries and querying, 25
- SELECT statement, 57
- T-SQL, 25
    - logical query processing, describing, 25
    - public newsgroups, visiting, 25

SUM aggregate, 174
SUM function, 172
- as aggregate function, 152

SUM window aggregate function, 607
supersets, 7
surrogate keys, 41–42
SWITCHOFFSET function, 45
synonym chaining, 316
synonym permissions, 317
synonyms, 315–322
- abstraction layer and, 317
- advantages of, over views, 318
- ALTER statement with, 316
- case scenarios, 323
- converting, to other objects, 323
- creating, 315–317
- and descriptive names for reporting, 319–320
- disadvantages of, 318
- dropping, 317
- editing/using thesaurus file to add, 206–208
- finding, in thesaurus files, 194
- names of, as T-SQL identifiers, 316
- other database objects vs., 318
- other objects, converting synonyms to, 323
- permissions and, 317
- and references to nonexistent objects, 317
- simplifying cross-database queries with, 320–321
- suggested practices, 324
- in T-SQL statements, 316

syntax, creating views, 301–302
sysadmin, 442
sys database schema, 269
SYSDATETIME(), 625
SYSDATETIME function, 44
SYSDATETIMEOFFSET function, 44
sys.dm_db_index_physical_stats, 552, 559, 571
sys.dm_db_index_usage_stats, 574
sys.dm_db_index_usage_stats dynamic management view, 541
sys.dm_db_missing_index_columns, 541
sys.dm_db_missing_index_details, 541
sys.dm_db_missing_index_groups, 541
sys.dm_db_missing_index_group_stats DMOs, 541
sys.dm_exec_query_stats, 648
sys.dm_exec_query_stats DMO, 541
sys.dm_exec_requests, 541

sys.dm_exec_sessions DMO, 540
sys.dm_exec_sessions dynamic management view, 541
sys.dm_exec_sql_text DMO, 541
sys.dm_exec_sql_text DMO., 542
sys.dm_exec_sql_text dynamic management function, 541
sys.dm_os_sys_info, 540
sys.dm_os_waiting_tasks DMO, 540
sys.dm_tran_active_transactions, 420
sys.dm_tran_active_transactions (DMV), 415
sys.dm_tran_database_transactions, 412
sys.fn_validate_plan_guide, 667
sys.fulltext_document_types catalog view, 193
sys.indexes, 558, 570
sys.indexes catalog, 541
sys.indexes catalog view, 552
sys.messages, 438
sys.objects, 473, 501, 616–617
sys.plan_guides, 668
sys.sequences view, 377
sys.sp_control_plan_guide, 667
sys.sp_create_plan_guide, 667
sys.sp_create_plan_guide_from_handle, 667
sys.sp_executesql system procedure, 521, 651
sys.sp_get_query_template, 667
sys.sp_updatestats, 586
sys.sp_updatestats system, 586
system statistical page, 585
system tables, 266
system transactions, 415
SYSUTCDATE function, 44

T

table expressions, 121–128, 266
    and views vs. inline table-valued functions, 127–128
    CTEs and, 124–127
    DELETE using, 360
    derived tables and, 122–124
    optimization of, 122
    pivoting data using, 168
    UPDATE statement and, 348–350
table functions
    OPENROWSET, 388
    OPENXML, 388
table hints, 661
    Table Hints (Transact-SQL) article, 664
table lock, 357
table metadata, views and, 318
table name, 30
    specifying target column name after, 331
table(s), 265–280, 556
    aliasing of, in joins, 104
    altering, 276–277
    base, 266
    case scenarios, 293
    choosing indexes for, 276
    clustered, 633, 635
    compressing data in, 275
    creating, 267–275
    creating, with full-text components, 197
    default values in, 273
    derived, 122–124, 266
    fields and records vs., 10
    grouped, 149
    naming, 270–272
    NULL values in, 273
    permanent vs. temporary, 304
    relations vs., 4
    schema name vs. table name of, 30
    shredding XML to, 230–232
    size of, 38
    specifying database schemas for, 269–270
    suggested practices, 294
    synonyms referring to, 323
    system, 266
    temporary, 266
    two-part naming of, 268
    views appearing as, 300
    views vs., 304
    windowed, 149
tables
    derived, 360
    locks, 357
Table Scan iterator, 632
    scenario for, 593
table schemas vs. database schemas, 269
table-valued subqueries, 118. *See also* table expressions
table-valued UDFs, 503–505
    create, 508–509
    inline, 503
    multistatement, 504–505
table variables, 266, 611, 620
    temporary tables vs., 611
    using, 622–623
TABLOCK hint, 333
tags, XML, 222

target column names
  specifying, 331
target table
  modification statements and constraints defined in, 331
tempdb, 426–427, 572, 614
TEMPLATE plan guides, 666–667
template (SQL Trace/SQL Server Profiler), 525
temporary tables, 266, 304, 611, 620
  case scenario, 624–625
  DDL and, 613–615
  global, 612–613
  indexes and, 613–615
  local, 612–613
  physical representation in tempdb, 616–617
  statistics and, 618–620
  suggested practices, 626
  table variables vs., 611
  transaction and, 617–618
termination, T-SQL statements, 4
test procedure using RECOMPILE query hint, 669–670
TEXT data type, 530
  full-text indexes on columns of, 192
text mining, 196
text() (node type test), 241
theory, importance of, 24
thesaurus files
  finding synonyms in, 194
  manually editing, 194
thesaurus terms (in searches), 192
three-valued logic, 62–66
THROW statements, 436, 438–439, 482, 491, 496
  parameters, 438
  RAISERROR vs., 438, 443
tiebreakers, 177
TIME data type, 38
  columns, 273
TIMEFROMPARTS function, 45
time functions. *See* date and time functions
TINYINT data type, 38
TODATETIMEOFFSET function, 45
tokens, 193
TOP filter
  DELETE filter using, 360
TOP operator, 306
TOP option (SELECT queries), 21
  DELETE statement with, 357, 360
  filtering data with, 84–87
  with inner queries, 121
  performance considerations with, 604
  specifying number of rows for, 85
trace, 525
transaction commands
  BEGIN (TRAN or TRANSACTION), 415, 416
  COMMIT (TRAN, TRANSACTION or WORK), 415, 416
  ROLLBACK (TRAN, TRANSACTION or WORK), 415, 416
transaction modes, 418–419, 428–429
  autocommit, 416
  explicit, 416, 418–419
  implicit, 416–418
transactions, 412–435
  ACID properties of, 413–414
  additional options, 421–422
  commands, 415
  cross-database, 421
  distributed, 421
  durability, 414
  exclusive locks, 423
  explicit, 435
  implementing, 463
  implicit, 428
  isolation levels, 426–428, 431–433
  levels, 415–416
  managing, 412–435
  marking, 420–421
  modes, 416–419
  nested, 418–420, 419–420
  states, 415–416
  system, 415
  temporary tables and, 617–618
  @@TRANCOUNT, detecting levels with, 415
  types, 415–422
  understanding, 412–414
  user, 415
  user transactions, default name of, 415
  XACT_ABORT with, 441
  XACT_STATE(), finding state with, 415
Transactions table, 605–607
Transact-SQL (T-SQL). *See* T-SQL (Transact-SQL)
triggers
  AFTER, 491–495
  AFTER, nested, 494–495
  DML, 491–492
  implementing, 490–500
  INSTEAD OF, 491, 495
  suggested practices, 512

TRIGGER statement, 471
troubleshooting deadlocks, 426
TRUNCATE statement, 358–359, 374, 378
  DELETE statement and, 356
  DELETE statement without WHERE clause vs., 372
  DELETE vs., 358, 364–365
truncating data, 362
TRY block, 442
  RAISERROR and, 443
  XACT_ABORT and, 444
TRY_CAST function, 40, 68
TRY/CATCH construct, 413, 444, 487, 512
  stored procedures and, 482
  structured error-handling with, 441–443
  THROW command and, 438
  using XACT_ABORT with, 444
TRY_CONVERT function, 40, 439–440
  CONVERT vs., 439
TRY_PARSE function, 40, 439–440
T-SQL strings
  generating, 453–454
  QUOTENAME and, 458–459
T-SQL (Transact-SQL), 2–13
  built-in functions in, 37
  case scenarios, 24
    code reviewer position, interviewing for a, 24
    theory, importance of, 24
  as declarative English-like language, 14–15
  developers, 443
  encapsulating code, 471
  error handling, 435–450
  evolution of, 2–5
  generating strings, 453–454
  mathematical foundations of, 2
  multiple grouping sets defined in, 150
  queries, grouping sets defined in, 150
  review questions, 13
  routine, 501
  statements, 428, 440–441
  stored procedures and, 470–490
  suggested practices, 25
    logical query processing, describing, 25
    public newsgroups, visiting, 25
  summary, 13
  synonyms and, 316
  terminology associated with, 10–12
  using, in relational way, 5–10
tuples, 4
two-part naming (of tables), 268
two-valued logic, 63
type of vs. formatting of value, 38

U

UDFs. *See* user-defined functions (UDFs)
UNBOUNDED FOLLOWING (ROWS delimiting option), 180
UNBOUNDED PRECEDING (ROWS delimiting option), 174
underscore (_), 34, 271
Understanding Row Versioning-Based Isolation Levels article, 433
Unicode
  character strings, types of, 37
  literals, delimiting, 68
  storage requirements, 40
  XML and, 223
uniform extent (data storage), 550
UNION ALL operator, 137–139
  view columns and, 306
UNION clause, SELECT statement and, 303
UNION operation
  optimizer hints and, 661
  view columns and, 306
UNION operator, 137–139
  EXCEPT operator vs., 140
UNIQUE constraints (keys), 283–284
  IDENTITY property using, 373
  indexes and, 615
UNIQUEIDENTIFIER data type
  GUIDs and, 43
  surrogate key generators and, 42
unpivoting data, 166–168
  and column types, 168
  identification of three elements involved in, 167
  as inverse of pivoting, 149
UNPIVOT operator, 166–168
  USING clause and, 388
unqualified UPDATE statements, 342
unstructured error handling (@@ERROR), 440, 445–446
unused indexes, 542–543
UPDATE action, 384, 386–388, 397
UPDATED function, 491
UPDATE() function as DML trigger, 496

update locks, 422
UPDATE statements, 342–343, 494
  AFTER triggers and, 492–493
  all-at-once updates, 351–352
  based on a variable, 350–351
  based on join, 344–345
  DBCC SHOW_STATISTICS command and, 586
  as DML trigger, 491
  IDENTITY property and, 378
  inline TVFs and, 504
  INSTEAD OF triggers and, 495
  limits on, when used in views, 305
  MERGE vs., 346
  MERGE/WHEN MATCHED statements vs., 387
  modify() XML method and, 251
  NOLOCK table hint and, 433
  nondeterministic update, 346–348
  OUTPUT clause, with, 397, 401–402
  query hints and, 661
  sequence object and, 375
  synonyms with, 316
  and table expressions, 348–350
  target table columns and, 398
  as transaction, 412
  transaction failure, ACID properties and, 413
  unqualified, 342
updating data, 341–355
  improving process for, 364
  nondeterministic UPDATE, 346–348
  sample data, 341–342
  UPDATE all-at-once, 351–352
  UPDATE and table expressions, 348–350
  UPDATE based on a variable, 350–351
  UPDATE based on join, 344–345
  UPDATE statement, 342–343
UPPER function (strings), 49
USE command, 452, 455, 471
user-defined functions (UDFs), 501–510
  about, 501
  CALLED ON NULL INPUT option, 506
  case scenario, 511
  ENCRYPTION option, 506
  EXECUTE AS option, 506
  limitations, 505
  options, 506
  performance considerations, 506
  RETURNS NULL ON NULL INPUT option, 506
  scalar, 502–503
  SCHEMABINDING option, 506
  suggested practices, 512
  synonyms for, 316
  table-valued, 503–505
user transactions, 415–422. *See also* transactions
USING clause (MERGE statement), 383, 388

V

value comparison operators, 242
value() method (XML data type), 250–251
value operator column, as search argument, 65
VALUE (secondary XML index), 256
VALUES table, MERGE statements and, 385
value vs. type formatting, 38
VARBINARY data type, 38, 39
VARBINARY(MAX) data type, 530
  columns, 273
  full-text indexes on columns of, 192
VARCHAR data type, 41
  columns, 272
  full-text indexes on columns of, 192
  Unicode types vs., 40
VARCHAR(MAX) data type
  as LOB, 530
  columns, 273
variables
  UPDATE based on, 350–351
Venn diagrams
  EXCEPT operator, 140
  INTERSECT operator, 139
  UNION ALL operator, 138
  UNION operator, 137
VIEW DEFINITION (permission level), 306
views, 266, 300–307, 323. *See also* inline functions
  abstracted layers presented by, 301
  advantages of synonyms over, 318
  altering, 305
  appearance of, as tables, 300
  building, for reports, 310
  case scenarios, 323
  converting, into inline functions, 312–313
  creating, 300
  distributed partitioned, 306
  dropping, 305
  filtering, 307
  indexed, 266, 304
  inline table-valued functions vs., 127–128

- and metadata, 306–307
- modifying data through, 305–306
- names of, 303–307
- names of, as T-SQL identifiers, 303
- options with, 302
- ordering results of, 304
- partitioned, 306
- passing parameters to, 304
- querying from, 304
- reading from, 301
- restrictions on, 304
- SELECT and UNION statements in, 303
- self-documenting, 302
- suggested practices, 324
- synonyms referring to, 323
- WITH CHECK OPTION with, 303

W

WAITFOR command, 481
WAITFOR DELAY option (WAITFOR command), 481
WAITFOR RECEIVE option (WAITFOR command), 481
WAITFOR TIME option (WAITFOR command), 481
Warnings property (Properties window in SSMS), 591
weighted terms (in searches), 192
WHEN clause, 387, 391
WHEN MATCHED [AND <predicate>] THEN statement (MERGE statement), 384
WHEN MATCHED clause, 387
WHEN MATCHED statement, 406
WHEN NOT MATCHED BY SOURCE [AND <predicate>] THEN statement (MERGE statement), 384
WHEN NOT MATCHED BY SOURCE clause (T-SQL extension to USING clause), 388
WHEN NOT MATCHED BY SOURCE statement (MERGE statement), 406
WHEN NOT MATCHED [BY TARGET] [AND <predicate>] THEN statement (MERGE statement), 384
WHEN NOT MATCHED clause, 387, 391
WHEN NOT MATCHED statement, 406
WHERE clause, 62
- aliases from SELECT clause and, 17
- combining predicates in, 66–67
- CONTAINS predicate with, 202
- DELETE statement and, 357
- Dynamic SQL and, 452
- filtered indexes, creating with, 566
- filtering range of dates using, 73
- filtering rows with NULLs using, 72
- filtering views with, 307
- full-text predicates as part of, 194
- functions, using to limit output, 503
- GROUP BY clause and, 153
- heaps and, 632
- indexes and, for optimization, 574–578
- INSERT SELECT statements and, 399
- in logical query processing phases, 17
- ON clause vs., 106
- performance considerations with, 506
- query optimization within, 519
- Query Optimizer vs., 578
- referring to window functions in, 178
- with ROW_NUMBER function, 123
- UPDATE statement and, 342, 351
- USE statements and, 455
- variables vs. literals, using with, 472
- with views, 309
- window functions and, 181
- with XML queries, 227

Where (FLWOR statement), 243
WHILE statements, 478–481
- BEGIN/END blocks and, 479
- branching logic and, 477
- ensuring termination of, 479
- unique iterator values for, 480

wildcards, 48, 69
- with XQuery, 241

windowed tables, 149
window frame
- clauses, 174
- extents, 175
- when not specified, 180

window functions, 172–184
- aggregate, 180
- aggregate functions, 172–176
- group functions vs., 172
- group queries vs., 172
- offset functions, 178–180
- ranking functions, 176–178
- using, 180–182

window offset functions, 181
window ordering clauses, LAG and LEAD functions and, 178
window partition clauses, LAG and LEAD functions and, 178

window ranking functions, 181
Windows Application log, 436
Windows Azure SQL Database, 426–427
window vs. presentation ordering, 177
WITH CHECK OPTION (view options), 303
WITH ENCRYPTION (view option), 302, 309
WITH HISTOGRAM option (DBCC SHOW_STATISTICS command), 587
WITH keyword, use with table hints, 663
WITH LOG clause, THROW statements and, 439
WITH MARK statement, 421
WITH (NOLOCK) table hint, 428, 432
WITH NOWAIT command, 440
WITH (READ UNCOMMITTED) table hint, 428, 432
WITH RECOMPILE option for stored procedures, 652, 659
WITH SCHEMABINDING (view option), 301–302, 309
WITH STOPATMARK statement, 421
WITH TIES option (SELECT command), 87
WITH VIEW_METADATA (view option), 302
word breakers, 193
World Wide Web Consortium (W3C), 235
WRITE method, 251
write performance, 41
writers, 426, 430
  blocking, 426

X

XACT_ABORT, 440–441, 441
  error handling, 446–447
  TRY/CATCH with, 444
XACT_STATE() function, 415, 442, 444
  @@TRANCOUNT vs., 416
XACT_STATE() values, 444
xdt namespace, 236
XML, 221–264, 530
  attribute-centric, 229
  basics of, 222–226
  case scenarios, 260–261
  indexes, 256
  Microsoft SQL Server 2012 support for, 221
  ordering in, 223
  producing, from relational data, 226–230
  shredding, to tables, 230–232
  using FOR XML to return results as, 222–235
  and XML data type, 249–260
  XQuery for querying data in, 235–249
XML data type, 249–260
  for dynamic schema, 252–256
  full-text indexes on columns of, 192
  methods, 250–252, 256–259
  when to use, 250
XML DML, 235
XML documents
  formatting of, 222
  navigating through, 240–243
  returning, 233
XML fragments, 223
  returning, 234
xml namespace, 236
XML nodes, 236
XML plans, 532
  showing, 250
XML Schema Description (XSD) documents, 225, 228
  XMLSCHEMA directive, returning with, 228
XPath expressions
  simple, 246
  with predicates in, 247
  XQuery, specifying with, 240
  XQuery vs., 235
XQuery, 221, 235–249
  atomic data types, list of, 238
  basics of, 236–239
  data types in, 238
  expressions with predicates in, 247
  FLWOR expressions in, 243–245
  functions in, 238–239
  navigation in, 245–248
  navigation using, 240–243
  simple expressions in, 246
xsi namespace, 236
xs namespace, 236

Y

YEAR function, 44

About the Authors

ITZIK BEN-GAN is a mentor and cofounder of SolidQ. A Microsoft SQL Server MVP since 1999, Itzik has delivered numerous training events around the world that are focused on T-SQL querying, query tuning, and programming. Itzik is the author of several books about T-SQL. He has written many articles for *SQL Server Pro*, in addition to articles and white papers for MSDN and *The SolidQ Journal*. Itzik's speaking engagements include Tech-Ed, SQL PASS, SQL Server Connections, presentations to various SQL Server user groups, and SolidQ events. Itzik is a subject matter expert within SolidQ for the company's T-SQL–related activities. He authored SolidQ's Advanced T-SQL and T-SQL Fundamentals courses and delivers them regularly worldwide.

DEJAN SARKA, MCT and SQL Server MVP, focuses on development of database and business intelligence applications. Besides working on projects, he spends a large part of his time training and mentoring. He is the founder of the Slovenian SQL Server and .NET Users Group. Dejan has authored or coauthored 11 books about databases and SQL Server. He also developed two courses and many seminars for Microsoft and SolidQ.

RON TALMAGE is a SolidQ database consultant who lives in Seattle. He is a mentor and cofounder of SolidQ, a SQL Server MVP, PASS Regional Mentor, and Chapter Leader of the Seattle SQL Server User Group (PNWSQL). He's been active in the SQL Server world since SQL Server 4.21a, and has authored numerous articles and white papers.