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*.

Querying Microsoft® SQL Server® 2012

Exam 70-461
Training Kit

Itzik Ben-Gan
Dejan Sarka
Ron Talmage

ISBN: 978-0-7356-6605-4

Ninth Printing: March 2015

Printed and bound in the United States of America.

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

Contents

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.

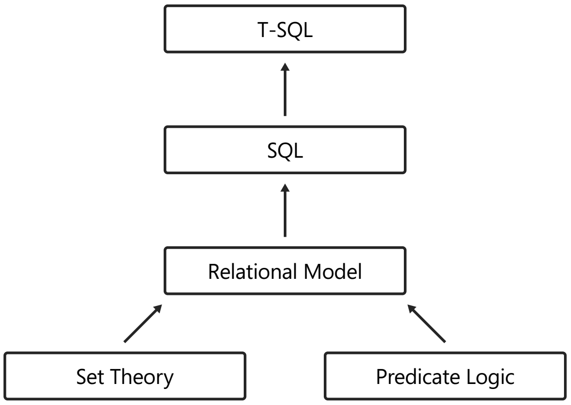

FIGURE 1-1 Evolution of T-SQL.

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)	&amp;
" (quotation mark)	&quot;
< (less than)	&lt;
> (greater than)	&gt;
' (apostrophe)	&apos;

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

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.