Microsoft

Microsoft® SQL Server® 2008 T-SQL Fundamentals

Itzik Ben-Gan

SOLID QUALITY MENTORS

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008938209

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 3 2 1 0 9 8

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ken Jones
Developmental Editor: Sally Stickney
Project Editor: Maria Gargiulo
Editorial Production: S4Carlisle Publishing Services
Technical Reviewer: Ron Talmage ; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X15-12280

To Dato

To live in hearts we leave behind,
Is not to die.

—Thomas Campbell

Contents at a Glance

Table of Contents

Acknowledgments

Many people have contributed to the book, directly and indirectly, and I'd like to acknowledge their contributions.

To Ron Talmage, the book's technical editor: I've asked Microsoft Press to work with you for a reason. You seek a true understanding of things; you look for subtleties; you appreciate SQL and logic; and on top of all this you have superb English. You've done an outstanding job!

To Dejan Sarka: I'd like to thank you for your help with the first chapter of the book, and for your insights regarding set theory, predicate logic, and the relational model. I like the fact that you always question things, even those that most people take for granted. You're one of the people whose thoughts and ideas I heed most. Your understanding of the relational model and your capacity for drinking beer are truly admirable, albeit that the examples you choose for demonstrating your ideas are not always politically correct. ;-)

Several people from Microsoft Press and S4Carlisle Publishing Services are due thanks. To Ken Jones, the project planner: it's a real pleasure working with you. I appreciate your attentiveness and the way you manage to handle us authors and our tempers. I also appreciate your friendship. Thanks to Sally Stickney, the development editor, for lifting the project off the ground, and to Maria Gargiulo, the project editor, for managing the project on a day-to-day basis. It was great to work with you! Thanks is also due to Christian Holdener and Tracy Ball, the vendor project managers, and to Becka McKay, the copy editor.

I'd like to thank my company, Solid Quality Mentors, for the best job I could ever hope for, which mainly involves teaching, and for making me feel like I'm part of family and friends. Fernando G. Guerrero, Brian Moran, and Douglas McDowell, who manage the company: you have a lot to be proud of. The company has grown and matured, and has accomplished great things. To my friends and colleagues from the company, Ron Talmage, Andrew J. Kelly, Eladio Rincón, Dejan Sarka, Herbert Albert, Fritz Lechnitz, Gianluca Hotz, Erik Veerman, Daniel A. Seara, Davide Mauri, Andrea Benedetti, Miguel Egea, Adolfo Wiernik, Javier Loria, Rushabh B. Mehta, and many others: it's an honor and pleasure to be part of the gang; I always look forward to spending more time with you over beer talking about SQL and other things! I'd like to thank Jeanne Reeves for making many of my classes possible, and all the back office team for their support. I'd also like to thank Kathy Blomstrom for managing our writing projects and for your excellent edits.

To Lubor Kollar, who's with the Microsoft SQL Server Customer Advisory Team (SQL CAT): I'd like to thank you for being such a great example, and for your friendship. You're always there to help or to find the right address for help when I have a question about SQL Server, and this contributed a lot to my T-SQL understanding. I always look forward to spending time together!

I'd like to thank several people from the product team. To Michael Wang, Michael Rys, and all others involved in the development of T-SQL: thanks for making T-SQL such a great language, notwithstanding the fact that the OVER clause is not yet fully implemented ;-). To Umachandar Jayachandran (UC); I know very few people who understand the true depths of T-SQL the way you do, and I can't tell you how glad I was when you joined the programmability team. I knew that T-SQL was in good hands!

To Sensei Yehuda Pantanowitz: you were my greatest teacher, and a friend; your passing away is unbearable.

To the team at *SQL Server Magazine:* Megan Bearly, Sheila Molnar, Mary Waterloo, Karen Forster, Michele Crockett, Mike Otey, Lavon Peters, and Anne Grubb: we've been working together for almost 10 years now, and I feel like it's my home. Thanks for giving me the freedom to write every month about a subject that is burning in my veins, and for all the work you do to enable the articles to be published.

I'd like to thank my fellow MVPs for your contribution to the SQL community and to my knowledge. A few deserve special thanks: Steve Kass, when I grow up, I want to be just like you! To Erland Sommarskog, Alejandro Mesa, Aaron Bertrand, and Tibor Karaszi: your participation in the newsgroups is truly astounding! Erland, your papers are a great source of information. To Marcello Poletti (Marc): I believe that we share similar feelings towards SQL and puzzles; your puzzles are wicked and they have deprived me of sleep more than once.

My true passion is for teaching; I'd like to thank my students for enabling me to fulfill my passion. Student questions and inquiries make me do a lot of research, and a lot of my knowledge today is due to those questions.

I'd like to thank my family for their support. To my parents, Gabriel and Emilia Ben-Gan, for supporting me in pursuing my passion, even if it means that we see each other less. And to my brother, Michael Ben-Gan and my sister, Ina Aviram, for being there for me.

Finally, Lilach, you give meaning to everything I do; contrary to the common cliché, I probably could finish the book without you. But then, why would I want to?

Introduction

This book walks you through your first steps in T-SQL (also known as Transact-SQL), which is the Microsoft SQL Server dialect of the standard ANSI-SQL language. You'll learn the theory behind T-SQL querying and programming, how to develop T-SQL code to query and modify data, and get an overview of programmable objects.

Although this book is intended for beginners, it is not merely a step-by-step book. It goes beyond the syntactical elements of T-SQL and explains the logic behind the language and its elements.

Occasionally the book covers subjects that may be considered advanced for readers who are new to T-SQL; therefore, those sections are optional reading. If you already feel comfortable with the material discussed in the book up to that point, you may want to tackle the more advanced subjects; otherwise, feel free to skip those sections and return to them after you've gained more experience. The text will indicate when a section may be considered more advanced and is provided as optional reading.

Many aspects of SQL are unique to the language, and are very different from other programming languages. This book helps you adopt the right state of mind and gain a true understanding of the language elements. You learn how to think in terms of sets and follow good SQL programming practices.

The book is not version-specific; it does, however, cover language elements that were introduced in recent versions of SQL Server, including SQL Server 2008. When I discuss language elements that were introduced recently, I specify the version in which they were added.

To complement the learning experience, the book provides exercises that enable you to practice what you've learned. The book occasionally provides optional exercises that are more advanced. Those exercises are intended for readers who feel very comfortable with the material and want to challenge themselves with more difficult problems. The optional exercises for advanced readers are labeled as such.

Who This Book Is For

This book is intended for T-SQL programmers, DBAs, architects, analysts, and SQL Server power users who just started working with SQL Server and need to write queries and develop code using Transact-SQL.

What This Book Is About

The book starts with both a theoretical background to T-SQL querying and programming in Chapter 1, laying the foundations for the rest of the book, and also coverage of creating tables and defining data integrity. The book moves on to various aspects of querying and modifying data, in Chapters 2 through 8, then to a discussion of concurrency and transactions in Chapter 9, and finally provides an overview of programmable objects in Chapter 10. The following section lists the chapter titles along with a short description:

Chapter 1, "Background to T-SQL Querying and Programming," provides a theoretical background about SQL, set theory, and predicate logic; examines the relational model and more; describes SQL Server's architecture; and explains how to create tables and define data integrity.

Chapter 2, "Single-Table Queries," covers various aspects of querying a single table using the *SELECT* statement.

Chapter 3, "Joins," covers querying multiple tables using joins, including cross joins, inner joins, and outer joins.

Chapter 4, "Subqueries," covers queries within queries, otherwise known as subqueries.

Chapter 5, "Table Expressions," covers derived tables, CTEs, views, inline table-valued functions, and the APPLY operator.

Chapter 6, "Set Operations," covers the set operations UNION, INTERSECT, and EXCEPT.

Chapter 7, "Pivot, Unpivot, and Grouping Sets," covers data-rotation techniques and working with grouping sets.

Chapter 8, "Data Modification," covers inserting, updating, deleting, and merging data.

Chapter 9, "Transactions and Concurrency," covers concurrency of user connections that work with the same data simultaneously; it covers concepts including transactions, locks, blocking, isolation levels, and deadlocks.

Chapter 10, "Programmable Objects," provides an overview to the T-SQL programming capabilities in SQL Server.

The book also provides an appendix, "Getting Started," to help you set up your environment, download the book's source code, install the sample database TSQLFundamentals2008, start writing code against SQL Server, and learn how to get help by working with SQL Server Books Online.

Companion Content

This book features a companion Web site that makes available to you all the code used in the book, the errata, additional resources, and more. The companion Web site is *http://www.insidetsql.com*. Please refer to Appendix A, "Getting Started," for details about the source code.

Hardware and Software Requirements

In Appendix A, "Getting Started," I explain which editions of SQL Server 2008 you can use to work with the code samples included with this book. Each edition of SQL Server may have different hardware and software requirements, and those requirements are well-documented in SQL Server Books Online under "Hardware and Software Requirements for Installing SQL Server 2008." Appendix A also explains how to work with SQL Server Books Online.

Find Additional Content Online

For more great information from Microsoft Press, visit the new Microsoft Press Online sites—your one-stop online resource for access to updates, sample chapters, articles, scripts, and e-books related to our industry-leading Microsoft Press titles. Check out the following sites: *http://www.microsoft.com/learning/books/online/developer* and *http://www.microsoft.com/learning/books/online/serverclient*.

Support for This Book

Every effort has been made to ensure the accuracy of this book and the contents of the companion Web site. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

Microsoft Press provides support for books at the following Web site:

http://www.microsoft.com/learning/support/books/

Questions and Comments

If you have comments, questions, or ideas regarding the book, or questions that are not answered by visiting the sites above, please send them to me via e-mail at:

itzik@SolidQ.com

Or via postal mail at:

Microsoft Press
Attn: *Microsoft SQL Server 2008 T-SQL Fundamentals* Editor
One Microsoft Way
Redmond, WA 98052-6399

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

Chapter 3
Joins

In this chapter:

The FROM clause of a query is the first clause to be logically processed, and within the FROM clause table operators operate on input tables. Microsoft SQL Server 2008 supports four table operators—JOIN, APPLY, PIVOT, and UNPIVOT. The JOIN table operator is standard, while APPLY, PIVOT, and UNPIVOT are T-SQL extensions to the standard. These last three were introduced in SQL Server 2005. Each table operator acts on tables provided to it as input, applies a set of logical query processing phases, and returns a table result. This chapter focuses on the JOIN table operator. The APPLY operator will be covered in Chapter 5, "Table Expressions," and the PIVOT and UNPIVOT operators will be covered in Chapter 7, "Pivot, Unpivot, and Grouping Sets."

A JOIN table operator operates on two input tables. The three fundamental types of joins are cross, inner, and outer. The three types of joins differ in how they apply their logical query processing phases; each type applies a different set of phases. A cross join applies only one phase—Cartesian Product. An inner join applies two phases—Cartesian Product and Filter. An outer join applies three phases—Cartesian Product, Filter, and Add Outer Rows. This chapter explains each of the join types and the phases involved in detail.

Logical query processing describes a generic series of logical steps that for any given query produces the correct result, while physical query processing is the way the query is processed by the RDBMS engine in practice. Some phases of logical query processing of joins may sound inefficient, but the physical implementation may be optimized. It's important to stress the term *logical* in logical query processing. The steps in the process apply operations to the input tables based on relational algebra. The database engine does not have to follow logical query processing phases literally as long as it can guarantee that the result that it produces is the same as dictated by logical query processing. The SQL Server relational engine often applies many shortcuts for optimization purposes when it knows that it can still produce the correct result. Even though this book's focus is to understand the logical aspects of querying, I want to stress this point to avoid any misunderstanding and confusion.

Cross Joins

Logically, a cross join is the simplest type of join. A cross join implements only one logical query processing phase—a Cartesian Product. This phase operates on the two tables provided as inputs to the join, and produces a Cartesian product of the two. That is, each row from one input is matched with all rows from the other. So if you have *m* rows in one table and *n* rows in the other, you get *m* × *n* rows in the result.

SQL Server supports two standard syntaxes for cross joins—the ANSI SQL-92 and ANSI SQL-89 syntaxes. I recommend that you use the ANSI-SQL 92 syntax for reasons that I'll describe shortly. Therefore, ANSI-SQL 92 syntax is the main syntax that I use throughout the book. For the sake of completeness, I describe both syntaxes in this section.

ANSI SQL-92 Syntax

The following query applies a cross join between the Customers and Employees tables (using the ANSI SQL-92 syntax) in the TSQLFundamentals2008 database, and returns the custid and empid attributes in the result set:

```
USE TSQLFundamentals2008;

SELECT C.custid, E.empid
FROM Sales.Customers AS C
  CROSS JOIN HR.Employees AS E;
```

Because there are 91 rows in the Customers table and 9 rows in the Employees table, this query produces a result set with 819 rows, as shown here in abbreviated form:

```
custid      empid
----------- -----------
1           1
1           2
1           3
1           4
1           5
1           6
1           7
1           8
1           9
2           1
2           2
2           3
2           4
2           5
2           6
2           7
2           8
2           9
...

(819 row(s) affected)
```

Using the ANSI SQL-92 syntax, you specify the *CROSS JOIN* keywords between the two tables involved in the join.

Notice that in the FROM clause of the preceding query, I assigned the aliases C and E to the Customers and Employees tables, respectively. The result set produced by the cross join is a virtual table with attributes that originate from both sides of the join. Because I assigned aliases to the source tables, the names of the columns in the virtual table are prefixed by the table aliases (for example, C.custid, E.empid). If you do not assign aliases to the tables in the FROM clause, the names of the columns in the virtual table are prefixed by the full source table names (for example, Customers.custid, Employees.empid). The purpose of the prefixes is to enable the identification of columns in an unambiguous manner when the same column name appears in both tables. The aliases of the tables are assigned for brevity. Note that you are required to use column prefixes only when referring to ambiguous column names (column names that appear in more than one table); in unambiguous cases column prefixes are optional. However, some people find it a good practice to always use column prefixes for the sake of clarity. Also note that if you assign an alias to a table, it is invalid to use the full table name as a column prefix; in ambiguous cases you have to use the table alias as a prefix.

ANSI SQL-89 Syntax

SQL Server also supports an older syntax for cross joins that was introduced in ANSI SQL-89. In this syntax you simply specify a comma between the table names like so:

```
SELECT C.custid, E.empid
FROM Sales.Customers AS C, HR.Employees AS E;
```

There is no logical or performance difference between the two syntaxes. Both syntaxes are integral parts of the latest SQL standard (ANSI SQL:2006 at the time of this writing), and both are fully supported by the latest version of SQL Server (SQL Server 2008 at the time of this writing). I am not aware of any plans to deprecate the older syntax, and I don't see any reason to do so while it's an integral part of the standard. However, I recommend using the ANSI SQL-92 syntax for reasons that will become clear after inner joins are explained.

Self Cross Joins

You can join multiple instances of the same table. This capability is known as *self-join* and is supported with all fundamental join types (cross, inner, and outer). For example, the following query performs a self cross join between two instances of the Employees table:

```
SELECT
  E1.empid, E1.firstname, E1.lastname,
  E2.empid, E2.firstname, E2.lastname
FROM HR.Employees AS E1
  CROSS JOIN HR.Employees AS E2;
```

This query produces all possible combinations of pairs of employees. Because the Employees table has 9 rows, this query returns 81 rows, shown here in abbreviated form:

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

(81 row(s) affected)
```

In a self-join, aliasing tables is not optional. Without table aliases, all column names in the result of the join would be ambiguous.

Producing Tables of Numbers

One situation in which cross joins can be very handy is when they are used to produce a result set with a sequence of integers (1, 2, 3, and so on). Such a sequence of numbers is an extremely powerful tool that I use for many purposes. Using cross joins you can produce the sequence of integers in a very efficient manner.

You can start by creating a table called Digits with a column called digit, and populate the table with 10 rows with the digits 0 through 9. Run the following code to create the Digits table in the tempdb database (for test purposes) and populate it with the 10 digits:

```
USE tempdb;
IF OBJECT_ID('dbo.Digits', 'U') IS NOT NULL DROP TABLE dbo.Digits;
CREATE TABLE dbo.Digits(digit INT NOT NULL PRIMARY KEY);

INSERT INTO dbo.Digits(digit)
  VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9);

/*
Note:
Above INSERT syntax is new in Microsoft SQL Server 2008.
```

```
In earlier versions use:

INSERT INTO dbo.Digits(digit) VALUES(0);
INSERT INTO dbo.Digits(digit) VALUES(1);
INSERT INTO dbo.Digits(digit) VALUES(2);
INSERT INTO dbo.Digits(digit) VALUES(3);
INSERT INTO dbo.Digits(digit) VALUES(4);
INSERT INTO dbo.Digits(digit) VALUES(5);
INSERT INTO dbo.Digits(digit) VALUES(6);
INSERT INTO dbo.Digits(digit) VALUES(7);
INSERT INTO dbo.Digits(digit) VALUES(8);
INSERT INTO dbo.Digits(digit) VALUES(9);
*/

SELECT digit FROM dbo.Digits;
```

This code uses a couple of syntax elements for the first time in this book, so I'll briefly explain them. Any text residing within a block starting with /* and ending with */ is treated as a block comment and is ignored by SQL Server. This code also uses an *INSERT* statement to populate the Digits table. If you're not familiar with the syntax of the *INSERT* statement, see Chapter 8, "Data Modification," for details. Note, however, that this code uses new syntax that was introduced in SQL Server 2008 for the *INSERT VALUES* statement, allowing a single statement to insert multiple rows. A block comment embedded in the code explains that in earlier versions you need to use a separate *INSERT VALUES* statement for each row.

The contents of the Digits table are shown here:

```
digit
-----------
0
1
2
3
4
5
6
7
8
9
```

Suppose you need to write a query that produces a sequence of integers in the range 1 through 1,000. You can cross three instances of the Digits table, each representing a different power of 10 (1, 10, 100). By crossing three instances of the same table, each instance with 10 rows, you get a result set with 1,000 rows. To produce the actual number, multiply the digit from each instance by the power of 10 it represents, sum the results, and add 1. Here's the complete query:

```
SELECT D3.digit * 100 + D2.digit * 10 + D1.digit + 1 AS n
FROM         dbo.Digits AS D1
  CROSS JOIN dbo.Digits AS D2
  CROSS JOIN dbo.Digits AS D3
ORDER BY n;
```

This query returns the following output, shown here in abbreviated form:

```
n
-----------
1
2
3
4
5
6
7
8
9
10
...
998
999
1000

(1000 row(s) affected)
```

This was just an example producing a sequence of 1,000 integers. If you need more, you can add more instances of the Digits table to the query. For example, if you need to produce a sequence of 1,000,000 rows, you would need to join six instances.

Inner Joins

An inner join applies two logical query processing phases—it applies a Cartesian product between the two input tables like a cross join, and then it filters rows based on a predicate that you specify. Like cross joins, inner joins have two standard syntaxes: ANSI SQL-92 and ANSI SQL-89.

ANSI SQL-92 Syntax

Using the ANSI SQL-92 syntax, you specify the *INNER JOIN* keywords between the table names. The *INNER* keyword is optional because an inner join is the default, so you can specify the *JOIN* keyword alone. You specify the predicate that is used to filter rows in a designated clause called ON. This predicate is also known as the *join condition*.

For example, the following query performs an inner join between the Employees and Orders tables in the TSQLFundamentals2008 database, matching employees and orders based on the predicate E.empid = O.empid:

```
USE TSQLFundamentals2008;

SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E
  JOIN Sales.Orders AS O
    ON E.empid = O.empid;
```

This query produces the following result set, shown here in abbreviated form:

```
empid       firstname  lastname             orderid
----------- ---------- -------------------- -----------
1           Sara       Davis                10258
1           Sara       Davis                10270
1           Sara       Davis                10275
1           Sara       Davis                10285
1           Sara       Davis                10292
...
2           Don        Funk                 10265
2           Don        Funk                 10277
2           Don        Funk                 10280
2           Don        Funk                 10295
2           Don        Funk                 10300
...

(830 row(s) affected)
```

For most people the easiest way to think of such an inner join is as matching each employee row to all order rows that have the same employee ID as the employee's employee ID. This is a simplified way to think of the join. The more formal way to think of the join based on relational algebra is that first the join performs a Cartesian product of the two tables (9 employee rows × 830 order rows = 7,470 rows), and then filters rows based on the predicate E.empid = O.empid, eventually returning 830 rows. As mentioned earlier, that's just the logical way the join is processed; in practice, physical processing of the query by the database engine can be different.

Recall the discussion from previous chapters about the three-valued predicate logic used by SQL. Like with the WHERE and HAVING clauses, the ON clause also returns only rows for which the predicate returns TRUE, and does not return rows for which the predicate evaluates to FALSE or UNKNOWN.

In the TSQLFundamentals2008 database all employees have related orders, so all employees show up in the output. However, had there been employees with no related orders, they would have been filtered out by the filter phase.

ANSI SQL-89 Syntax

Similar to cross joins, inner joins can be expressed using the ANSI SQL-89 syntax. You specify a comma between the table names just like in a cross join, and specify the join condition in the query's WHERE clause, like so:

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E, Sales.Orders AS O
WHERE E.empid = O.empid;
```

Note that the ANSI SQL-89 syntax has no ON clause.

Again, both syntaxes are standard, fully supported by SQL Server, and interpreted the same by the engine, so you shouldn't expect any performance difference between the two. But one syntax is safer, as explained in the next section.

Inner Join Safety

I strongly recommend that you stick to the ANSI SQL-92 join syntax because it is safer in several ways. Say you intend to write an inner join query, and by mistake forget to specify the join condition. With the ANSI SQL-92 syntax the query becomes invalid and the parser generates an error. For example, try to run the following code:

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E
  JOIN Sales.Orders AS O;
```

You get the following error:

```
Msg 102, Level 15, State 1, Line 3
Incorrect syntax near ';'.
```

Even though it might not be obvious immediately that the error involves a missing join condition, you will figure it out eventually and fix the query. However, if you forget to specify the join condition using the ANSI SQL-89 syntax, you get a valid query that performs a cross join:

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E, Sales.Orders AS O;
```

Because the query doesn't fail, the logical error might go unnoticed for a while, and users of your application might end up relying on incorrect results. It is unlikely that a programmer would forget to specify the join condition with such short and simple queries; however, most production queries are much more complicated and have multiple tables, filters, and other query elements. In those cases the likelihood of forgetting to specify a join condition increases.

If I've convinced you that it is important to use the ANSI SQL-92 syntax for inner joins, you might wonder whether the recommendation holds for cross joins. Because no join condition is involved, you might think that both syntaxes are just as good for cross joins. However, I recommend staying with the ANSI SQL-92 syntax with cross joins for a couple of reasons—one being consistency. Also, let's say you do use the ANSI SQL-89 syntax. Even if you intended to write a cross join, when other developers need to review or maintain your code, how will they know whether you intended to write a cross join or intended to write an inner join and forgot to specify the join condition?

Further Join Examples

This section covers a few join examples that are known by specific names, including composite joins, non-equi joins, and multi-table joins.

Composite Joins

A composite join is simply a join based on a predicate that involves more than one attribute from each side. A composite join is commonly required when you need to join two tables based on a primary key–foreign key relationship, and the relationship is composite: that is, based on more than one attribute. For example, suppose you have a foreign key defined on dbo.Table2, columns col1, col2, referencing dbo.Table1, columns col1, col2, and you need to write a query that joins the two based on primary key–foreign key relationship. The FROM clause of the query would look like this:

```
FROM dbo.Table1 AS T1
  JOIN dbo.Table2 AS T2
    ON T1.col1 = T2.col1
    AND T1.col2 = T2.col2
```

For a more tangible example, suppose that you need to audit updates to column values against the OrderDetails table in the TSQLFundamentals2008 database. You create a custom auditing table called OrderDetailsAudit:

```
USE TSQLFundamentals2008;
IF OBJECT_ID('Sales.OrderDetailsAudit', 'U') IS NOT NULL
  DROP TABLE Sales.OrderDetailsAudit;
CREATE TABLE Sales.OrderDetailsAudit
(
  lsn        INT NOT NULL IDENTITY,
  orderid    INT NOT NULL,
  productid  INT NOT NULL,
  dt         DATETIME NOT NULL,
  loginname  sysname NOT NULL,
  columnname sysname NOT NULL,
  oldval     SQL_VARIANT,
  newval     SQL_VARIANT,
  CONSTRAINT PK_OrderDetailsAudit PRIMARY KEY(lsn),
  CONSTRAINT FK_OrderDetailsAudit_OrderDetails
    FOREIGN KEY(orderid, productid)
    REFERENCES Sales.OrderDetails(orderid, productid)
);
```

Each audit row stores a log serial number (lsn), the key of the modified row (orderid, productid), the name of the modified column (columnname), the old value (*oldval*), new value (*newval*), when the change took place (dt), and who made the change (loginname). The table has a foreign key defined on the attributes orderid, productid, referencing the primary key of the OrderDetails table, which is defined on the attributes orderid, productid.

Suppose that you already have in place all the required processes that audit column value changes taking place in the OrderDetails table in the OrderDetailsAudit table.

You need to write a query that returns all value changes that took place against the column qty, but in each result row you need to return the current value from the OrderDetails table, and the values before and after the change from the OrderDetailsAudit table. You need to join the two tables based on primary key–foreign key relationship like so:

```
SELECT OD.orderid, OD.productid, OD.qty,
  ODA.dt, ODA.loginname, ODA.oldval, ODA.newval
FROM Sales.OrderDetails AS OD
  JOIN Sales.OrderDetailsAudit AS ODA
    ON OD.orderid = ODA.orderid
    AND OD.productid = ODA.productid
WHERE ODA.columnname = N'qty';
```

Because the relationship is based on multiple attributes, the join condition is composite.

Non-Equi Joins

When the join condition involves only an equality operator, the join is said to be an equi join. When the join condition involves any operator besides equality, the join is said to be a non-equi join. As an example of a non-equi join, the following query joins two instances of the Employees table to produce unique pairs of employees:

```
SELECT
  E1.empid, E1.firstname, E1.lastname,
  E2.empid, E2.firstname, E2.lastname
FROM HR.Employees AS E1
  JOIN HR.Employees AS E2
    ON E1.empid < E2.empid;
```

Notice the predicate specified in the ON clause. The purpose of the query is to produce unique pairs of employees. Had you used a cross join, you would have gotten self pairs (for example, 1 with 1), and also mirrored pairs (for example, 1 with 2 and also 2 with 1). Using an inner join with a join condition that says that the key in the left side must be smaller than the key in the right side eliminates the two inapplicable cases. Self pairs are eliminated because both sides are equal. With mirrored pairs, only one of the two cases qualifies because out of the two cases, only one will have a left key that is smaller than the right key. In our case, out of the 81 possible pairs of employees that a cross join would have returned, our query returns the 36 unique pairs shown here:

```
empid firstname  lastname         empid  firstname  lastname
----- ---------- ---------------- ------ ---------- -----------------
1     Sara       Davis            2      Don        Funk
1     Sara       Davis            3      Judy       Lew
2     Don        Funk             3      Judy       Lew
```

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

(36 row(s) affected)
```

If it is still not clear to you what this query does, try to process it one step at a time with a smaller set of employees. For example, suppose the Employees table contained only employees 1, 2, and 3. First, produce the Cartesian product of two instances of the table:

```
E1.empid      E2.empid
------------- -------------
1             1
1             2
1             3
2             1
2             2
2             3
3             1
3             2
3             3
```

Next, filter the rows based on the predicate E1.empid < E2.empid, and you are left with only three rows:

```
E1.empid      E2.empid
------------- -------------
1             2
1             3
2             3
```

Multi-Table Joins

A join table operator operates only on two tables, but a single query can have multiple joins. In general, when more than one table operator appears in the FROM clause, the table operators are logically processed from left to right. That is, the result table of the first table operator is served as the left input to the second table operator; the result of the second table operator is served as the left input to the third table operator and so on. So if there are multiple joins in the FROM clause, logically the first join operates on two base tables, but all other joins get the result of the preceding join as their left input. With cross joins and inner joins, the database engine can (and often does) internally rearrange join ordering for optimization purposes because it won't have an impact on the correctness of the result of the query.

As an example, the following query joins the Customers and Orders tables to match customers with their orders, and joins the result of the first join with the OrderDetails table to match orders with their order lines:

```
SELECT
  C.custid, C.companyname, O.orderid,
  OD.productid, OD.qty
FROM Sales.Customers AS C
  JOIN Sales.Orders AS O
    ON C.custid = O.custid
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

This query returns the following output, shown here in abbreviated form:

```
custid      companyname       orderid     productid   qty
----------- ----------------- ----------- ----------- ------
85          Customer ENQZT    10248       11          12
85          Customer ENQZT    10248       42          10
85          Customer ENQZT    10248       72          5
79          Customer FAPSM    10249       14          9
79          Customer FAPSM    10249       51          40
34          Customer IBVRG    10250       41          10
34          Customer IBVRG    10250       51          35
34          Customer IBVRG    10250       65          15
84          Customer NRCSK    10251       22          6
84          Customer NRCSK    10251       57          15
...

(2155 row(s) affected)
```

Outer Joins

Outer joins are usually harder for people to grasp compared to the other types of joins. First I will describe the fundamentals of outer joins. If by the end of the section "Fundamentals of Outer Joins," you feel very comfortable with the material and are ready for more advanced content, you can read an optional section describing aspects of outer joins that are beyond the fundamentals. Otherwise, feel free to skip that part and return to it when you feel comfortable with the material.

Fundamentals of Outer Joins

Outer joins were introduced in ANSI SQL-92 and unlike inner and cross joins, they only have one standard syntax—the one where you specify the *JOIN* keyword between the table names, and the join condition in the ON clause. Outer joins apply the two logical processing phases that inner joins apply (Cartesian product and the ON filter), plus a third phase called Adding Outer Rows that is unique to this type of join.

In an outer join you mark a table as a "preserved" table by using the keywords *LEFT OUTER JOIN*, *RIGHT OUTER JOIN*, or *FULL OUTER JOIN* between the table names. The *OUTER* keyword is optional. The *LEFT* keyword means that the rows of the left table are preserved, the *RIGHT* keyword means that the rows in the right table are preserved, and the *FULL* keyword means that the rows in both the left and right tables are preserved. The third logical query processing phase of an outer join identifies the rows from the preserved table that did not find matches in the other table based on the ON predicate. This phase adds those rows to the result table produced by the first two phases of the join, and uses NULLs as place holders for the attributes from the nonpreserved side of the join in those outer rows.

A good way to understand outer joins is through an example. The following query joins the Customers and Orders tables based on a match between the customer's customer ID and the order's customer ID to return customers and their orders. The join type is a left outer join; therefore, the query also returns customers who did not place any orders in the result:

```
SELECT C.custid, C.companyname, O.orderid
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid;
```

This query returns the following output, shown here in abbreviated form:

```
custid      companyname     orderid
----------- --------------- -----------
1           Customer NRZBB  10643
1           Customer NRZBB  10692
1           Customer NRZBB  10702
1           Customer NRZBB  10835
1           Customer NRZBB  10952
...
```

```
21          Customer KIDPX  10414
21          Customer KIDPX  10512
21          Customer KIDPX  10581
21          Customer KIDPX  10650
21          Customer KIDPX  10725
22          Customer DTDMN  NULL
23          Customer WVFAF  10408
23          Customer WVFAF  10480
23          Customer WVFAF  10634
23          Customer WVFAF  10763
23          Customer WVFAF  10789
...
56          Customer QNIVZ  10684
56          Customer QNIVZ  10766
56          Customer QNIVZ  10833
56          Customer QNIVZ  10999
56          Customer QNIVZ  11020
57          Customer WVAXS  NULL
58          Customer AHXHT  10322
58          Customer AHXHT  10354
58          Customer AHXHT  10474
58          Customer AHXHT  10502
58          Customer AHXHT  10995
...
91          Customer CCFIZ  10792
91          Customer CCFIZ  10870
91          Customer CCFIZ  10906
91          Customer CCFIZ  10998
91          Customer CCFIZ  11044

(832 row(s) affected)
```

Two customers in the Customers table did not place any orders. Their IDs are 22 and 57. Observe that in the output of the query both customers are returned with NULLs in the attributes from the Orders table. Logically, the rows for these two customers were filtered out by the second phase of the join (filter based on the ON predicate), but the third phase added those as outer rows. Had the join been an inner join, these two rows would not have been returned. These two rows are added to preserve all the rows of the left table.

You can consider two kinds of rows in the result of an outer join in respect to the preserved side—inner rows and outer rows. Inner rows are rows that have matches in the other side based on the ON predicate, and outer rows are rows that don't. An inner join returns only inner rows, while an outer join returns both inner and outer rows.

A common question when using outer joins that is the source of a lot of confusion is whether to specify a predicate in the ON or WHERE clauses of a query. You can see that with respect to rows from the preserved side of an outer join, the filter based on the ON predicate is not final. In other words, the ON predicate does not determine whether the row will show up in the output, only whether it will be matched with rows from the other side. So when you need to express a predicate that is not final—meaning a predicate that determines which rows

to match from the nonpreserved side—specify the predicate in the ON clause. When you need a filter to be applied after outer rows are produced, and you want the filter to be final, specify the predicate in the WHERE clause. The WHERE clause is processed after the FROM clause—namely, after all table operators were processed and (in the case of outer joins), after all outer rows were produced. Also, the WHERE clause is final with respect to rows that it filters out, unlike the ON clause.

Suppose that you need to return only customers who did not place any orders, or more technically speaking, you need to return only outer rows. You can use the previous query as your basis, and add a WHERE clause that filters only outer rows. Remember that outer rows are identified by the NULLs in the attributes from the nonpreserved side of the join. So you can filter only the rows where one of the attributes in the nonpreserved side of the join is NULL, like so:

```
SELECT C.custid, C.companyname
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
WHERE O.orderid IS NULL;
```

This query returns only two rows, with the customers 22 and 57:

```
custid      companyname
----------- ---------------
22          Customer DTDMN
57          Customer WVAXS

(2 row(s) affected)
```

Notice a couple of important things about this query. Recall the discussions about NULLs earlier in the book: When looking for a NULL you should use the operator IS NULL and not an equality operator, because an equality operator comparing something with a NULL always returns UNKNOWN—even when comparing two NULLs. Also, the choice of which attribute from the nonpreserved side of the join to filter is important. You should choose an attribute that can only have a NULL when the row is an outer row and not otherwise (for example, a NULL originating from the base table). For this purpose, three cases are safe to consider—a primary key column, a join column, and a column defined as NOT NULL. A primary key column cannot be NULL; therefore, a NULL in such a column can only mean that the row is an outer row. If a row has a NULL in the join column, that row is filtered out by the second phase of the join, so a NULL in such a column can only mean that it's an outer row. And obviously a NULL in a column that is defined as NOT NULL can only mean that the row is an outer row.

To practice what you've learned and get a better grasp of outer joins, make sure that you perform the exercises for this chapter.

Beyond the Fundamentals of Outer Joins

This section covers more advanced aspects of outer joins and is provided as optional reading for when you feel very comfortable with the fundamentals of outer joins.

Including Missing Values

You can use outer joins to identify and include missing values when querying data. For example, suppose that you need to query all orders from the Orders table in the TSQLFundamentals2008 database. You need to ensure that you get at least one row in the output for each date in the range January 1, 2006 through December 31, 2008. You don't want to do anything special with dates within the range that have orders. But you do want the output to include the dates with no orders, with NULLs as placeholders in the attributes of the order.

To solve the problem, you can first write a query that returns a sequence of all dates in the requested date range. You can then perform a left outer join between that set and the Orders table. This way the result also includes the missing order dates.

To produce a sequence of dates in a given range, I usually use an auxiliary table of numbers. I create a table called Nums with a column called n, and populate it with a sequence of integers (1, 2, 3, and so on). I find that an auxiliary table of numbers is an extremely powerful general-purpose tool that I end up using to solve many problems. You need to create it only once in the database and populate it with as many numbers as you might need. Run the code in Listing 3-1 to create the Nums table in the dbo schema and populate it with 100,000 rows:

LISTING 3-1 Code to Create and Populate the Auxiliary Table Nums

```
SET NOCOUNT ON;
USE TSQLFundamentals2008;
IF OBJECT_ID('dbo.Nums', 'U') IS NOT NULL DROP TABLE dbo.Nums;
CREATE TABLE dbo.Nums(n INT NOT NULL PRIMARY KEY);

DECLARE @i AS INT = 1;
/*
Note:
The ability to declare and initialize variables in one statement
is new in Microsoft SQL Server 2008.
In earlier versions use separate DECLARE and SET statements:

DECLARE @i AS INT;
SET @i = 1;
*/
BEGIN TRAN
  WHILE @i <= 100000
  BEGIN
    INSERT INTO dbo.Nums VALUES(@i);
    SET @i = @i + 1;
  END
COMMIT TRAN
SET NOCOUNT OFF;
```

Note Don't worry if you don't yet understand some parts of the code, such as using variables and loops—those are explained later in the book. For now, it's enough to understand what this code is supposed to do; how it does it is not the focus of discussion here. But in case you're curious and cannot resist, you can find details in Chapter 10, "Programmable Objects." I should point out, however, that declaring and initializing variables in the same statement is new in SQL Server 2008 as the block comment that appears in the code explains. If you're working with an earlier version, you should use separate *DECLARE* and *SET* statements.

As the first step in the solution, you need to produce a sequence of all dates in the requested range. You can achieve this by querying the Nums table, and filtering as many numbers as the number of days in the requested date range. You can use the *DATEDIFF* function to calculate that number. By adding *n* - 1 days to the starting point of the date range (January 1, 2006) you get the actual date in the sequence. Here's the solution query:

```
SELECT DATEADD(day, n-1, '20060101') AS orderdate
FROM dbo.Nums
WHERE n <= DATEDIFF(day, '20060101', '20081231') + 1
ORDER BY orderdate;
```

This query returns a sequence of all dates in the range January 1, 2006 through December 31, 2008, as shown here in abbreviated form:

```
orderdate
-----------------------
2006-01-01 00:00:00.000
2006-01-02 00:00:00.000
2006-01-03 00:00:00.000
2006-01-04 00:00:00.000
2006-01-05 00:00:00.000
...
2008-12-27 00:00:00.000
2008-12-28 00:00:00.000
2008-12-29 00:00:00.000
2008-12-30 00:00:00.000
2008-12-31 00:00:00.000

(1096 row(s) affected)
```

The next step is to extend the previous query, adding a left outer join between Nums and the Orders tables. The join condition compares the order date produced from the Nums table using the expression DATEADD(day, Nums.n - 1, '20060101') and the orderdate from the Orders table like so:

```
SELECT DATEADD(day, Nums.n - 1, '20060101') AS orderdate,
  O.orderid, O.custid, O.empid
FROM dbo.Nums
  LEFT OUTER JOIN Sales.Orders AS O
    ON DATEADD(day, Nums.n - 1, '20060101') = O.orderdate
WHERE Nums.n <= DATEDIFF(day, '20060101', '20081231') + 1
ORDER BY orderdate;
```

This query produces the following output, shown here in abbreviated form:

```
orderdate                orderid     custid      empid
------------------------ ----------- ----------- -----------
2006-01-01 00:00:00.000  NULL        NULL        NULL
2006-01-02 00:00:00.000  NULL        NULL        NULL
2006-01-03 00:00:00.000  NULL        NULL        NULL
2006-01-04 00:00:00.000  NULL        NULL        NULL
2006-01-05 00:00:00.000  NULL        NULL        NULL
...
2006-06-29 00:00:00.000  NULL        NULL        NULL
2006-06-30 00:00:00.000  NULL        NULL        NULL
2006-07-01 00:00:00.000  NULL        NULL        NULL
2006-07-02 00:00:00.000  NULL        NULL        NULL
2006-07-03 00:00:00.000  NULL        NULL        NULL
2006-07-04 00:00:00.000  10248       85          5
2006-07-05 00:00:00.000  10249       79          6
2006-07-06 00:00:00.000  NULL        NULL        NULL
2006-07-07 00:00:00.000  NULL        NULL        NULL
2006-07-08 00:00:00.000  10250       34          4
2006-07-08 00:00:00.000  10251       84          3
2006-07-09 00:00:00.000  10252       76          4
2006-07-10 00:00:00.000  10253       34          3
2006-07-11 00:00:00.000  10254       14          5
2006-07-12 00:00:00.000  10255       68          9
2006-07-13 00:00:00.000  NULL        NULL        NULL
2006-07-14 00:00:00.000  NULL        NULL        NULL
2006-07-15 00:00:00.000  10256       88          3
2006-07-16 00:00:00.000  10257       35          4
...
2008-12-27 00:00:00.000  NULL        NULL        NULL
2008-12-28 00:00:00.000  NULL        NULL        NULL
2008-12-29 00:00:00.000  NULL        NULL        NULL
2008-12-30 00:00:00.000  NULL        NULL        NULL
2008-12-31 00:00:00.000  NULL        NULL        NULL

(1446 row(s) affected)
```

Order dates that do not appear in the Orders table appear in the output of the query with NULLs in the order attributes.

Filtering Attributes from the Nonpreserved Side of an Outer Join

When you need to review code involving outer joins to look for logical bugs, one of the things you should examine is the WHERE clause. If the predicate in the WHERE clause refers to an attribute from the nonpreserved side of the join using an expression in the form <attribute> <operator> <value>, it's usually an indication of a bug. This is because attributes from the nonpreserved side of the join are NULLs in outer rows, and an expression in the form NULL <operator> <value> yields UNKNOWN (unless it's the IS NULL operator explicitly looking for NULLs). Recall that a WHERE clause filters UNKNOWN out. Such a predicate in

the WHERE clause causes all outer rows to be filtered out, effectively nullifying the outer join. In other words, it's as if the join type logically becomes an inner join. So the programmer either made a mistake in the choice of the join type, or made a mistake in the predicate. If this is not clear yet, the following example might help. Consider the following query:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
WHERE O.orderdate >= '20070101';
```

The query performs a left outer join between the Customers and Orders tables. Prior to applying the WHERE filter, the join operator returns inner rows for customers who placed orders, and outer rows for customers who didn't place orders, with NULLs in the order attributes. The predicate O.orderdate >= '20070101' in the WHERE clause evaluates to UNKNOWN for all outer rows because those have a NULL in the O.orderdate attribute. All outer rows are eliminated by the WHERE filter, as you can see in the output of the query, shown here in abbreviated form:

```
custid      companyname       orderid     orderdate
----------- ----------------- ----------- -----------------------
19          Customer RFNQC    10400       2007-01-01 00:00:00.000
65          Customer NYUHS    10401       2007-01-01 00:00:00.000
20          Customer THHDP    10402       2007-01-02 00:00:00.000
20          Customer THHDP    10403       2007-01-03 00:00:00.000
49          Customer CQRAA    10404       2007-01-03 00:00:00.000
...
58          Customer AHXHT    11073       2008-05-05 00:00:00.000
73          Customer JMIKW    11074       2008-05-06 00:00:00.000
68          Customer CCKOT    11075       2008-05-06 00:00:00.000
9           Customer RTXGC    11076       2008-05-06 00:00:00.000
65          Customer NYUHS    11077       2008-05-06 00:00:00.000

(678 row(s) affected)
```

This means that the use of an outer join here was futile. The programmer either made a mistake in using an outer join or made a mistake in the WHERE predicate.

Using Outer Joins in a Multi-Table Join

Recall the discussion about all-at-once operations in Chapter 2, "Single Table Queries." The concept means that all expressions that appear in the same logical query processing phase are logically evaluated at the same point in time. However, this concept is not applicable to the processing of table operators in the FROM phase. Table operators are logically evaluated from left to right. Rearranging the order in which outer joins are processed might result in different output, so you cannot rearrange them at will.

Some interesting logical bugs have to do with the logical order in which outer joins are processed. For example, a common logical bug involving outer joins could be considered a variation of the bug in the previous section. Suppose that you write a multi-table join query with an outer join between two tables, followed by an inner join with a third table. If the predicate in the inner join's ON clause compares an attribute from the nonpreserved side of the outer join and an attribute from the third table, all outer rows are filtered out. Remember that outer rows have NULLs in the attributes from the nonpreserved side of the join, and comparing a NULL with anything yields UNKNOWN, and UNKNOWN is filtered out by the ON filter. In other words, such a predicate would nullify the outer join and logically it would be as if you specified an inner join. For example, consider the following query:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

The first join is an outer join returning customers and their orders and also customers who did not place any orders. The outer rows representing customers with no orders have NULLs in the order attributes. The second join matches order lines from the OrderDetails table with rows from the result of the first join based on the predicate O.orderid = OD.orderid; however, in the rows representing customers with no orders, the O.orderid attribute is NULL. Therefore, the predicate evaluates to UNKNOWN and those rows are filtered out. The output shown here in abbreviated form doesn't contain the customers 22 and 57, the two customers who did not place orders:

```
custid      orderid     productid   qty
----------- ----------- ----------- ------
85          10248       11          12
85          10248       42          10
85          10248       72          5
79          10249       14          9
79          10249       51          40
...
65          11077       64          2
65          11077       66          1
65          11077       73          2
65          11077       75          4
65          11077       77          2

(2155 row(s) affected)
```

To generalize the problem: outer rows are nullified whenever any kind of outer join (left, right, or full) is followed by a subsequent inner join or right outer join. That's assuming, of course, that the join condition compares the NULLs from the left side with something from the right side.

You have several ways to get around the problem if you want to return customers with no orders in the output. One option is to use a left outer join in the second join as well:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
  LEFT OUTER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

This way, the outer rows produced by the first join aren't filtered out, as you can see in the output shown here in abbreviated form:

```
custid      orderid     productid   qty
----------- ----------- ----------- ------
85          10248       11          12
85          10248       42          10
85          10248       72          5
79          10249       14          9
79          10249       51          40
...
65          11077       64          2
65          11077       66          1
65          11077       73          2
65          11077       75          4
65          11077       77          2
22          NULL        NULL        NULL
57          NULL        NULL        NULL

(2157 row(s) affected)
```

A second option is to first join Orders and OrderDetails using an inner join, and then join to the Customers table using a right outer join:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
  RIGHT OUTER JOIN Sales.Customers AS C
     ON O.custid = C.custid;
```

This way, the outer rows are produced by the last join, and are not filtered out.

A third option is to use parentheses to make the inner join between Orders and OrderDetails become an independent logical phase. This way you can apply a left outer join between the Customers table and the result of the inner join between Orders and OrderDetails. The query would look like this:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN
```

```
      (Sales.Orders AS O
         JOIN Sales.OrderDetails AS OD
           ON O.orderid = OD.orderid)
    ON C.custid = O.custid;
```

Using the COUNT Aggregate with Outer Joins

Another common logical bug involves using COUNT with outer joins. When you group the result of an outer join and use the *COUNT(*)* aggregate, the aggregate takes into consideration both inner rows and outer rows because it counts rows regardless of their contents. Usually, you're not supposed to take outer rows into consideration for the purposes of counting. For example, the following query is supposed to return the count of orders for each customer:

```
SELECT C.custid, COUNT(*) AS numorders
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
GROUP BY C.custid;
```

However, the *COUNT(*)* aggregate counts rows regardless of their meaning or contents, and customers who did not place orders—like 22 and 57—each have an outer row in the result of the join. As you can see in the output of the query shown here in abbreviated form, both 22 and 57 show up with a count of 1, while the number of orders they place is actually 0:

```
custid      numorders
----------- -----------
1           6
2           4
3           7
4           13
5           18
...
22          1
...
57          1
...
87          15
88          9
89          14
90          7
91          7

(91 row(s) affected)
```

The *COUNT(*)* aggregate function cannot detect whether a row really represents an order. To fix the problem you should use COUNT(<column>) instead of *COUNT(*)*, and provide a column from the nonpreserved side of the join. This way, the *COUNT()* aggregate ignores

outer rows because they have a NULL in that column. Remember to use a column that can only be NULL in case the row is an outer row—for example, the primary key column orderid:

```
SELECT C.custid, COUNT(O.orderid) AS numorders
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
GROUP BY C.custid;
```

Notice in the output shown here in abbreviated form that the customers 22 and 57 now show up with a count of 0:

```
custid      numorders
----------- -----------
1           6
2           4
3           7
4           13
5           18
...
22          0
...
57          0
...
87          15
88          9
89          14
90          7
91          7

(91 row(s) affected)
```

Conclusion

This chapter covered the join table operator. It described the logical query processing phases involved in the three fundamental types of joins—cross, inner, and outer. The chapter also covered further join examples including composite joins, non-equi joins, and multi-table joins. The chapter concluded with an optional reading section covering more advanced aspects of outer joins. To practice what you've learned, go over the exercises for this chapter.

Exercises

This section provides exercises to help you familiarize yourself with the subjects discussed in this chapter. All exercises involve querying objects in the TSQLFundamentals2008 database.

1-1

Run the following code to create the dbo.Nums auxiliary table in the TSQLFundamentals2008 database:

```
SET NOCOUNT ON;
USE TSQLFundamentals2008;
IF OBJECT_ID('dbo.Nums', 'U') IS NOT NULL DROP TABLE dbo.Nums;
CREATE TABLE dbo.Nums(n INT NOT NULL PRIMARY KEY);

DECLARE @i AS INT = 1;
BEGIN TRAN
  WHILE @i <= 100000
  BEGIN
    INSERT INTO dbo.Nums VALUES(@i);
    SET @i = @i + 1;
  END
COMMIT TRAN
SET NOCOUNT OFF;
```

1-2

Write a query that generates five copies out of each employee row.

Tables involved: HR.Employees, and dbo.Nums tables.

Desired output:

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

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

(45 row(s) affected)
```

1-3 (Optional, Advanced)

Write a query that returns a row for each employee and day in the range June 12, 2009 – June 16, 2009.
Tables involved: HR.Employees, and dbo.Nums tables.
Desired output:

```
empid       dt
----------- -----------------------
1           2009-06-12 00:00:00.000
1           2009-06-13 00:00:00.000
1           2009-06-14 00:00:00.000
1           2009-06-15 00:00:00.000
1           2009-06-16 00:00:00.000
2           2009-06-12 00:00:00.000
2           2009-06-13 00:00:00.000
2           2009-06-14 00:00:00.000
2           2009-06-15 00:00:00.000
2           2009-06-16 00:00:00.000
3           2009-06-12 00:00:00.000
3           2009-06-13 00:00:00.000
3           2009-06-14 00:00:00.000
3           2009-06-15 00:00:00.000
3           2009-06-16 00:00:00.000
4           2009-06-12 00:00:00.000
4           2009-06-13 00:00:00.000
4           2009-06-14 00:00:00.000
4           2009-06-15 00:00:00.000
4           2009-06-16 00:00:00.000
5           2009-06-12 00:00:00.000
5           2009-06-13 00:00:00.000
```

```
5           2009-06-14 00:00:00.000
5           2009-06-15 00:00:00.000
5           2009-06-16 00:00:00.000
6           2009-06-12 00:00:00.000
6           2009-06-13 00:00:00.000
6           2009-06-14 00:00:00.000
6           2009-06-15 00:00:00.000
6           2009-06-16 00:00:00.000
7           2009-06-12 00:00:00.000
7           2009-06-13 00:00:00.000
7           2009-06-14 00:00:00.000
7           2009-06-15 00:00:00.000
7           2009-06-16 00:00:00.000
8           2009-06-12 00:00:00.000
8           2009-06-13 00:00:00.000
8           2009-06-14 00:00:00.000
8           2009-06-15 00:00:00.000
8           2009-06-16 00:00:00.000
9           2009-06-12 00:00:00.000
9           2009-06-13 00:00:00.000
9           2009-06-14 00:00:00.000
9           2009-06-15 00:00:00.000
9           2009-06-16 00:00:00.000

(45 row(s) affected)
```

2

Return U.S. customers, and for each customer return the total number of orders and total quantities.
Tables involved: Sales.Customers, Sales.Orders, and Sales.OrderDetails tables.
Desired output:

```
custid      numorders   totalqty
----------- ----------- -----------
32          11          345
36          5           122
43          2           20
45          4           181
48          8           134
55          10          603
65          18          1383
71          31          4958
75          9           327
77          4           46
78          3           59
82          3           89
89          14          1063

(13 row(s) affected)
```

3

Return customers and their orders including customers who placed no orders.
Tables involved: Sales.Customers, and Sales.Orders tables.
Desired output (abbreviated):

```
custid      companyname     orderid     orderdate
----------- --------------- ----------- -----------------------
85          Customer ENQZT  10248       2006-07-04 00:00:00.000
79          Customer FAPSM  10249       2006-07-05 00:00:00.000
34          Customer IBVRG  10250       2006-07-08 00:00:00.000
84          Customer NRCSK  10251       2006-07-08 00:00:00.000
...
73          Customer JMIKW  11074       2008-05-06 00:00:00.000
68          Customer CCKOT  11075       2008-05-06 00:00:00.000
9           Customer RTXGC  11076       2008-05-06 00:00:00.000
65          Customer NYUHS  11077       2008-05-06 00:00:00.000
22          Customer DTDMN  NULL        NULL
57          Customer WVAXS  NULL        NULL

(832 row(s) affected)
```

4

Return customers who placed no orders.
Tables involved: Sales.Customers, and Sales.Orders tables.
Desired output:

```
custid      companyname
----------- ---------------
22          Customer DTDMN
57          Customer WVAXS

(2 row(s) affected)
```

5

Return customers with orders placed on Feb 12, 2007 along with their orders.
Tables involved: Sales.Customers, and Sales.Orders tables.
Desired output:

```
custid      companyname     orderid     orderdate
----------- --------------- ----------- -----------------------
66          Customer LHANT  10443       2007-02-12 00:00:00.000
5           Customer HGVLZ  10444       2007-02-12 00:00:00.000

(2 row(s) affected)
```

6 (Optional, Advanced)

Return customers with orders placed on Feb 12, 2007 along with their orders. Also return customers who didn't place orders on Feb 12, 2007.
Tables involved: Sales.Customers, and Sales.Orders tables.
Desired output (abbreviated):

```
custid      companyname       orderid     orderdate
----------- ----------------- ----------- -----------------------
72          Customer AHPOP    NULL        NULL
58          Customer AHXHT    NULL        NULL
25          Customer AZJED    NULL        NULL
18          Customer BSVAR    NULL        NULL
91          Customer CCFIZ    NULL        NULL
...
33          Customer FVXPQ    NULL        NULL
53          Customer GCJSG    NULL        NULL
39          Customer GLLAG    NULL        NULL
16          Customer GYBBY    NULL        NULL
4           Customer HFBZG    NULL        NULL
5           Customer HGVLZ    10444       2007-02-12 00:00:00.000
42          Customer IAIJK    NULL        NULL
34          Customer IBVRG    NULL        NULL
63          Customer IRRVL    NULL        NULL
73          Customer JMIKW    NULL        NULL
15          Customer JUWXK    NULL        NULL
...
21          Customer KIDPX    NULL        NULL
30          Customer KSLQF    NULL        NULL
55          Customer KZQZT    NULL        NULL
71          Customer LCOUJ    NULL        NULL
77          Customer LCYBZ    NULL        NULL
66          Customer LHANT    10443       2007-02-12 00:00:00.000
38          Customer LJUCA    NULL        NULL
59          Customer LOLJO    NULL        NULL
36          Customer LVJSO    NULL        NULL
64          Customer LWGMD    NULL        NULL
29          Customer MDLWA    NULL        NULL
...

(91 row(s) affected)
```

7 (Optional, Advanced)

Return all customers, and for each return a Yes/No value depending on whether the customer placed an order on Feb 12, 2007.
Tables involved: Sales.Customers, and Sales.Orders tables.
Desired output (abbreviated):

```
custid      companyname       HasOrderOn20070212
----------- ----------------- ------------------
1           Customer NRZBB    No
2           Customer MLTDN    No
3           Customer KBUDE    No
```

```
4           Customer HFBZG    No
5           Customer HGVLZ    Yes
6           Customer XHXJV    No
7           Customer QXVLA    No
8           Customer QUHWH    No
9           Customer RTXGC    No
10          Customer EEALV    No
...

(91 row(s) affected)
```

Solutions

This section provides solutions to the exercises for this chapter.

1-2

Producing multiple copies of rows can be achieved with a fundamental technique that utilizes a cross join. If you need to produce five copies out of each employee row, you need to perform a cross join between the Employees table and a table that has five rows; alternatively, you can perform a cross join between Employees and a table that has more than five rows, but filter only five from that table in the WHERE clause. The Nums table is very convenient for this purpose. Simply cross Employees and Nums, and filter from Nums as many rows as the number of requested copies (five in this case). Here's the solution query:

```
SELECT E.empid, E.FirstName, E.LastName, Nums.n
FROM HR.Employees AS E
  CROSS JOIN dbo.Nums
WHERE Nums.n <= 5
ORDER BY n, empid;
```

1-3

This exercise is an extension of the previous exercise. Instead of being asked to produce a predetermined constant number of copies out of each employee row, you are asked to produce a copy for each day in a certain date range. So here you need to calculate the number of days in the requested date range using the *DATEDIFF* function, and refer to the result of that expression in the query's WHERE clause instead of referring to a constant. To produce the dates, simply add *n* - 1 days to the date that starts the requested range. Here's the solution query:

```
SELECT E.empid,
  DATEADD(day, D.n - 1, '20090612') AS dt
FROM HR.Employees AS E
  CROSS JOIN dbo.Nums AS D
WHERE D.n <= DATEDIFF(day, '20090612', '20090616') + 1
ORDER BY empid, dt;
```

The *DATEDIFF* function returns 4 because there is a four-day difference between June 12, 2009 and June 16, 2009. Add 1 to the result, and you get 5 for the five days in the range. So the WHERE clause filters five rows from Nums where *n* is smaller than or equal to 5. By adding *n* - 1 days to June 12, 2009, you get all dates in the range June 12, 2009 and June 16, 2009.

2

This exercise requires you to write a query that joins three tables: Customers, Orders, and OrderDetails. The query should filter in the WHERE clause only rows where the customer's country is USA. Because you are asked to return aggregates per customer, the query should group the rows by customer ID. You need to resolve a tricky issue here to return the right number of orders for each customer. Because of the join between Orders and OrderDetails, you don't get only one row per order—you get one row per order line. So if you use the *COUNT(*)* function in the SELECT list, you get back the number of order lines for each customer and not the number of orders. To resolve this issue, you need to take each order into consideration only once. You can do this by using COUNT(DISTINCT O.orderid) instead of *COUNT(*)*. The total quantities don't create any special issues because the quantity is associated with the order line and not the order. Here's the solution query:

```
SELECT C.custid, COUNT(DISTINCT O.orderid) AS numorders, SUM(OD.qty) AS totalqty
FROM Sales.Customers AS C
  JOIN Sales.Orders AS O
    ON O.custid = C.custid
  JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
WHERE C.country = N'USA'
GROUP BY C.custid;
```

3

To get both customers who placed orders and customers who didn't place orders in the result, you need to use an outer join like so:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  LEFT JOIN Sales.Orders AS O
    ON O.custid = C.custid;
```

This query returns 832 rows (including the customers 22 and 57, who didn't place orders). An inner join between the tables would return only 830 rows without these customers.

4

This exercise is an extension of the previous one. To return only customers who didn't place orders, you need to add a WHERE clause to the query that filters only outer rows; namely, rows

that represent customers with no orders. Outer rows have NULLs in the attributes from the nonpreserved side of the join (Orders). But to make sure that the NULL is a placeholder for an outer row and not a NULL that originated from the table, it is recommended that you refer to an attribute that is the primary key, or the join column, or one defined as not allowing NULLs. Here's the solution query referring to the primary key of the Orders table in the WHERE clause:

```
SELECT C.custid, C.companyname
FROM Sales.Customers AS C
  LEFT JOIN Sales.Orders AS O
    ON O.custid = C.custid
WHERE O.orderid IS NULL;
```

This query returns only two rows for the customers 22 and 57, who didn't place orders.

5

This exercise involves writing a query that performs an inner join between Customers and Orders, and filters only rows where the order date is February 12, 2007:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  JOIN Sales.Orders AS O
    ON O.custid = C.custid
WHERE O.orderdate = '20070212';
```

The WHERE clause filtered out Customers who didn't place orders on February 12, 2007, but that was the request.

6

This exercise builds on the previous one. The trick here is to realize two things. First, you need an outer join because you are supposed to return customers who do not meet a certain criteria. Second, the filter on the order date must appear in the ON clause and not the WHERE clause. Remember that the WHERE filter is applied after outer rows are added and is final. Your goal is to match orders to customers only if the order was placed by the customer and on February 12, 2007. You still want to get customers who didn't place orders on that date in the output; in other words, the filter on the order date should only determine matches and not be considered final in regards to the customer rows. Hence the ON clause should match customers and orders based on both an equality between the customer's customer ID and the order's customer ID, and the order date being February 12, 2007. Here's the solution query:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  LEFT JOIN Sales.Orders AS O
    ON O.custid = C.custid
    AND O.orderdate = '20070212';
```

7

This exercise is an extension of the previous exercise. Here, instead of returning matching orders, you just need to return a Yes/No value indicating whether there is a matching order. Remember that in an outer join a nonmatch is identified as an outer row with NULLs in the attributes of the nonpreserved side. So you can use a simple CASE expression that checks whether the current row is an outer one, in which case it returns 'Yes'; otherwise, it returns 'No'. Because technically you can have more than one match per customer, you should add a DISTINCT clause to the SELECT list. This way you get only one row back for each customer. Here's the solution query:

```
SELECT DISTINCT C.custid, C.companyname,
  CASE WHEN O.orderid IS NOT NULL THEN 'Yes' ELSE 'No' END AS [HasOrderOn20070212]
FROM Sales.Customers AS C
  LEFT JOIN Sales.Orders AS O
    ON O.custid = C.custid
    AND O.orderdate = '20070212';
```

Chapter 5

Table Expressions

In this chapter:

Table expressions are named query expressions that represent a valid relational table. You can use them in data manipulation statements similar to other tables. Microsoft SQL Server supports four types of table expressions: derived tables, common table expressions (CTEs), views, and inline table-valued functions (inline TVFs), each of which I will describe in detail in this chapter. The focus of this chapter is SELECT queries against table expressions; Chapter 8, "Data Modification," covers modifications against table expressions.

Table expressions are not physically materialized anywhere—they are virtual. A query against a table expression is internally translated to a query against the underlying objects. The benefits of using table expressions are typically related to logical aspects of your code and not to performance. For example, table expressions help you simplify your solutions by using a modular approach. Table expressions also help you circumvent certain restrictions in the language, such as the inability to refer to column aliases assigned in the SELECT clause in query clauses that are logically processed prior to the SELECT clause.

This chapter also introduces the APPLY table operator used in conjunction with a table expression. I will explain how to use this operator to apply a table expression to each row of another table.

Derived Tables

Derived tables (also known as table subqueries) are defined in the FROM clause of an outer query. Their scope of existence is the outer query. As soon as the outer query is finished, the derived table is gone.

You specify the query defining the derived table within parentheses, followed by the AS clause and the derived table name. For example, the following code defines a derived table called USACusts based on a query that returns all customers from the United States, and the outer query selects all rows from the derived table:

```
USE TSQLFundamentals2008;

SELECT *
FROM (SELECT custid, companyname
      FROM Sales.Customers
      WHERE country = N'USA') AS USACusts;
```

In this particular case, which is a simple example of the basic syntax, a derived table is not needed because the outer query doesn't apply any manipulation.

The code in this basic example returns the following output:

```
custid      companyname
----------- ---------------
32          Customer YSIQX
36          Customer LVJSO
43          Customer UISOJ
45          Customer QXPPT
48          Customer DVFMB
55          Customer KZQZT
65          Customer NYUHS
71          Customer LCOUJ
75          Customer XOJYP
77          Customer LCYBZ
78          Customer NLTYP
82          Customer EYHKM
89          Customer YBQTI
```

A query must meet three requirements to be valid to define a table expression of any kind:

1. **Order is not guaranteed.** A table expression is supposed to represent a relational table, and the rows in a relational table have no guaranteed order. Recall that this aspect of a relation stems from set theory. For this reason, ANSI SQL disallows an ORDER BY clause in queries that are used to define table expressions. T-SQL follows this restriction for the most part, with one exception—when TOP is also specified. In the context of a query with the TOP option, the ORDER BY clause serves a logical purpose: defining for the TOP option which rows to filter. If you use a query with TOP and ORDER BY to define a table expression, ORDER BY is only guaranteed to serve the logical filtering purpose for the TOP option and not the usual presentation purpose. If the outer query against the table expression does not have a presentation ORDER BY, the output is not guaranteed to be returned in any particular order. The section "Views and the ORDER BY Clause," later in this chapter, provides more detail on this item.
2. **All columns must have names.** All columns in a table must have names; therefore, you must assign column aliases to all expressions in the SELECT list of the query that is used to define a table expression.

3. **All column names must be unique.** All column names in a table must be unique; therefore, a table expression that has multiple columns with the same name is invalid. This might happen when the query defining the table expression joins two tables, and both tables have a column with the same name. If you need to incorporate both columns in your table expression, they must have different column names. You can resolve this by assigning the two columns with different column aliases.

Assigning Column Aliases

One of the benefits of using table expressions is that in any clause of the outer query you can refer to column aliases that were assigned in the SELECT clause of the inner query. This helps you get around the fact that you can't refer to column aliases assigned in the SELECT clause in query clauses that are logically processed prior to the SELECT clause (for example, WHERE or GROUP BY).

For example, suppose that you need to write a query against the Sales.Orders table and return the number of distinct customers handled in each order year. The following attempt is invalid because the GROUP BY clause refers to a column alias that was assigned in the SELECT clause, and the GROUP BY clause is logically processed prior to the SELECT clause:

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

You could solve the problem by referring to the expression YEAR(orderdate) in both the GROUP BY and the SELECT clauses, but this is an example with a short expression. What if the expression were much longer? Maintaining two copies of the same expression might hurt code readability and maintainability and is more prone to errors. To solve the problem in a way that requires only one copy of the expression, you can use a table expression like so:

LISTING 5-1 Query with a Derived Table Using Inline Aliasing Form

```
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM (SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders) AS D
GROUP BY orderyear;
```

This query returns the following output:

```
orderyear   numcusts
----------- -----------
2006        67
2007        86
2008        81
```

This code defines a derived table called D based on a query against the Orders table that returns the order year and customer ID from all rows. The SELECT list of the inner query uses inline aliasing format to assign the alias orderyear to the expression YEAR(orderdate). The outer query can refer to the orderyear column alias in both the GROUP BY and SELECT clauses, because as far as the outer query is concerned, it queries a table called D with columns called orderyear and custid.

As I mentioned earlier, SQL Server expands the definition of the table expression and accesses the underlying objects directly. After expansion, the query in Listing 5-1 looks like this:

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

This is just to emphasize that you use table expressions for logical (not performance-related) reasons. Generally speaking, table expressions have neither positive nor negative performance impact.

The code in Listing 5-1 uses the inline aliasing format to assign column aliases to expressions. The syntax for inline aliasing is <expression> [AS] <alias>. Note that the word AS is optional in the syntax for inline aliasing; however, I find that it helps the readability of the code and recommend using it.

In some cases, you might prefer to use a second supported form for assigning column aliases, which you can think of as an external form. With this form you do not assign column aliases following the expressions in the SELECT list—you specify all target column names in parentheses following the table expression's name like so:

```
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM (SELECT YEAR(orderdate), custid
      FROM Sales.Orders) AS D(orderyear, custid)
GROUP BY orderyear;
```

It is generally recommended that you use the inline form for a couple of reasons. If you need to debug the code when using the inline form, when you highlight the query defining the table expression and run it, the columns in the result appear with the aliases you assigned. With the external form, you cannot include the target column names when you highlight the table expression query, so the result appears with no column names in the case of the unnamed expressions. Also, when the table expression query is lengthy, using the external form it can be quite difficult to figure out which column alias belongs to which expression.

Even though it's a best practice to use the inline aliasing form, in some cases you may find the external form more convenient to work with. For example, when the query defining the table expression isn't going to undergo any further revisions and you want to treat it like a "black box"—you want to focus your attention on the table expression name followed by the target column list when you look at the outer query.

Using Arguments

In the query defining a derived table, you can refer to arguments. The arguments can be local variables and input parameters to a routine such as a stored procedure or function. For example, the following code declares and initializes a local variable called *@empid*, and the query in the code that is used to define the derived table D refers to the local variable in the WHERE clause:

```
DECLARE @empid AS INT = 3;

/*
-- Prior to SQL Server 2008 use separate DECLARE and SET statements:
DECLARE @empid AS INT;
SET @empid = 3;
*/

SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM (SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders
      WHERE empid = @empid) AS D
GROUP BY orderyear;
```

This query returns the number of distinct customers per year that handled the orders of the input employee (the employee whose ID is stored in the variable *@empid*). Here's the output of this query:

```
orderyear   numcusts
----------- -----------
2006        16
2007        46
2008        30
```

Nesting

If you need to define a derived table using a query that by itself refers to a derived table, you end up nesting derived tables. Nesting of derived tables is a result of the fact that a derived table is defined in the FROM clause of the outer query and not separately. Nesting is a problematic aspect of programming in general as it tends to complicate the code and reduce its readability.

For example, the code in Listing 5-2 returns order years and the number of customers handled in each year only for years in which more than 70 customers were handled:

LISTING 5-2 Query with Nested Derived Tables

```
SELECT orderyear, numcusts
FROM (SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
      FROM (SELECT YEAR(orderdate) AS orderyear, custid
            FROM Sales.Orders) AS D1
      GROUP BY orderyear) AS D2
WHERE numcusts > 70;
```

This code returns the following output:

```
orderyear   numcusts
----------- -----------
2007        86
2008        81
```

The purpose of the innermost derived table, D1, is to assign the column alias orderyear to the expression YEAR(orderdate). The query against D1 refers to orderyear in both the GROUP BY and SELECT clauses, and assigns the column alias numcusts to the expression COUNT(DISTINCT custid). The query against D1 is used to define the derived table D2. The query against D2 refers to numcusts in the WHERE clause to filter order years in which more than 70 customers were handled.

The whole purpose of using table expressions in this example was to simplify the solution by reusing column aliases instead of repeating expressions. However, with the complexity added by the nesting aspect of derived tables, I'm not sure that the solution is simpler than the alternative, which does not make any use of derived tables but instead repeats expressions:

```
SELECT YEAR(orderdate) AS orderyear, COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY YEAR(orderdate)
HAVING COUNT(DISTINCT custid) > 70;
```

In short, nesting is a problematic aspect of derived tables.

Multiple References

Another problematic aspect of derived tables stems from the fact that derived tables are defined in the FROM clause of the outer query and not prior to the outer query. As far as the FROM clause of the outer query is concerned, the derived table doesn't exist yet; therefore, if you need to refer to multiple instances of the derived table, you can't. Instead, you have to define multiple derived tables based on the same query. The query in Listing 5-3 provides an example:

LISTING 5-3 Multiple Derived Tables Based on the Same Query

```
SELECT Cur.orderyear,
  Cur.numcusts AS curnumcusts, Prv.numcusts AS prvnumcusts,
  Cur.numcusts - Prv.numcusts AS growth
FROM (SELECT YEAR(orderdate) AS orderyear,
        COUNT(DISTINCT custid) AS numcusts
      FROM Sales.Orders
      GROUP BY YEAR(orderdate)) AS Cur
  LEFT OUTER JOIN
     (SELECT YEAR(orderdate) AS orderyear,
        COUNT(DISTINCT custid) AS numcusts
      FROM Sales.Orders
      GROUP BY YEAR(orderdate)) AS Prv
    ON Cur.orderyear = Prv.orderyear + 1;
```

This query joins two instances of a table expression to create two derived tables: the first derived table, Cur, represents current years, and the second derived table, Prv, represents previous years. The join condition Cur.orderyear = Prv.orderyear + 1 ensures that each row from the first derived table matches with the previous year of the second. By making it a LEFT outer join, the first year that has no previous year is also returned from the Cur table. The SELECT clause of the outer query calculates the difference between the number of customers handled in the current and previous years.

The code in Listing 5-3 produces the following output:

```
orderyear   curnumcusts prvnumcusts growth
----------- ----------- ----------- -----------
2006        67          NULL        NULL
2007        86          67          19
2008        81          86          -5
```

The fact that you cannot refer to multiple instances of the same derived table forces you to maintain multiple copies of the same query definition. This leads to lengthy code that is hard to maintain and is prone to errors.

Common Table Expressions

Common table expressions (CTEs) are another form of table expression very similar to derived tables, yet with a couple of important advantages. CTEs were introduced in SQL Server 2005 and are part of ANSI SQL:1999 and later standards.

CTEs are defined using a *WITH* statement and have the following general form:

```
WITH <CTE_Name>[(<target_column_list>)]
AS
(
  <inner_query_defining_CTE>
)
<outer_query_against_CTE>;
```

The inner query defining the CTE must follow all requirements mentioned earlier to be valid to define a table expression. As a simple example, the following code defines a CTE called USACusts based on a query that returns all customers from the United States, and the outer query selects all rows from the CTE:

```
WITH USACusts AS
(
  SELECT custid, companyname
  FROM Sales.Customers
  WHERE country = N'USA'
)
SELECT * FROM USACusts;
```

As with derived tables, as soon as the outer query finishes, the CTE gets out of scope.

Note The WITH clause is used in T-SQL for several different purposes. To avoid ambiguity, when the WITH clause is used to define a CTE, the preceding statement in the same batch—if one exists—must be terminated with a semicolon. And oddly enough, the semicolon for the entire CTE is not required, though I still recommend specifying it.

Assigning Column Aliases

CTEs also support two forms of column aliasing—inline and external. For the inline form, specify <expression> AS <column_alias>; for the external form, specify the target column list in parentheses immediately after the CTE name.

Here's an example of the inline form:

```
WITH C AS
(
  SELECT YEAR(orderdate) AS orderyear, custid
  FROM Sales.Orders
)
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM C
GROUP BY orderyear;
```

And here's an example of the external form:

```
WITH C(orderyear, custid) AS
(
  SELECT YEAR(orderdate), custid
  FROM Sales.Orders
)
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM C
GROUP BY orderyear;
```

The motivations for using one form or the other are similar to those described in the context of derived tables.

Using Arguments

As with derived tables, you can also use arguments in the query used to define a CTE. Here's an example:

```
DECLARE @empid AS INT = 3;

/*
-- Prior to SQL Server 2008 use separate DECLARE and SET statements:
DECLARE @empid AS INT;
SET @empid = 3;
*/
```

```
WITH C AS
(
  SELECT YEAR(orderdate) AS orderyear, custid
  FROM Sales.Orders
  WHERE empid = @empid
)
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM C
GROUP BY orderyear;
```

Defining Multiple CTEs

On the surface, the difference between derived tables and CTEs might seem to be merely semantic. However, the fact that you first define a CTE and then use it gives it several important advantages over derived tables. One of those advantages is that if you need to refer to one CTE from another, you don't end up nesting them like derived tables. Instead, you simply define multiple CTEs separated by commas under the same *WITH* statement. Each CTE can refer to all previously defined CTEs, and the outer query can refer to all CTEs. For example, the following code is the CTE alternative to the nested derived tables approach in Listing 5-2:

```
WITH C1 AS
(
  SELECT YEAR(orderdate) AS orderyear, custid
  FROM Sales.Orders
),
C2 AS
(
  SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
  FROM C1
  GROUP BY orderyear
)
SELECT orderyear, numcusts
FROM C2
WHERE numcusts > 70;
```

Because you define a CTE before you use it, you don't end up nesting CTEs. Each CTE appears separately in the code in a modular manner. This modular approach substantially improves the readability and maintainability of the code compared to the nested derived table approach.

Technically you cannot nest CTEs, nor can you define a CTE within the parentheses of a derived table. However, nesting is a problematic practice; therefore, think of these restrictions as aids to code clarity rather than obstacles.

Multiple References

The fact that a CTE is defined first and then queried has another advantage: As far as the FROM clause of the outer query is concerned, the CTE already exists; therefore, you

can refer to multiple instances of the same CTE. For example, the following code is the logical equivalent of the code shown earlier in Listing 5-3, using CTEs instead of derived tables:

```
WITH YearlyCount AS
(
  SELECT YEAR(orderdate) AS orderyear,
    COUNT(DISTINCT custid) AS numcusts
  FROM Sales.Orders
  GROUP BY YEAR(orderdate)
)
SELECT Cur.orderyear,
  Cur.numcusts AS curnumcusts, Prv.numcusts AS prvnumcusts,
  Cur.numcusts - Prv.numcusts AS growth
FROM YearlyCount AS Cur
  LEFT OUTER JOIN YearlyCount AS Prv
    ON Cur.orderyear = Prv.orderyear + 1;
```

As you can see, the CTE YearlyCount is defined once and accessed twice in the FROM clause of the outer query—once as Cur and once as Prv. You need to maintain only one copy of the CTE query and not multiple copies as you would with derived tables.

If you're curious about performance, recall that earlier I mentioned that typically table expressions have no performance impact because they are not physically materialized anywhere. Both references to the CTE here are going to be expanded. Internally, this query has a self join between two instances of the Orders table, each of which involves scanning the table data and aggregating it before the join—the same physical processing that takes place with the derived table approach.

Recursive CTEs

This section is optional because it covers subjects that are beyond the fundamentals.

CTEs are unique among table expressions because they have recursive capabilities. A recursive CTE is defined by at least two queries (more are possible)—at least one query known as the *anchor member* and at least one query known as the *recursive member*. The general form of a basic recursive CTE looks like this:

```
WITH <CTE_Name>[(<target_column_list>)]
AS
(
  <anchor_member>
  UNION ALL
  <recursive_member>
)
<outer_query_against_CTE>;
```

The anchor member is a query that returns a valid relational result table—like a query that is used to define a nonrecursive table expression. The anchor member query is invoked only once.

The recursive member is a query that has a reference to the CTE name. The reference to the CTE name represents what is logically the previous result set in a sequence of executions. The first time that the recursive member is invoked, the previous result set represents whatever the anchor member returned. In each subsequent invocation of the recursive member, the reference to the CTE name represents the result set returned by the previous invocation of the recursive member. The recursive member has no explicit recursion termination check—the termination check is implicit. The recursive member is invoked repeatedly until it returns an empty set, or exceeds some limit.

Both queries must be compatible in terms of the number of columns they return and the data types of the corresponding columns.

The reference to the CTE name in the outer query represents the unified result sets of the invocation of the anchor member and all invocations of the recursive member.

If this is your first encounter with recursive CTEs, you might find this explanation hard to understand. They are best explained with an example. The following code demonstrates how to use a recursive CTE to return information about an employee (Don Funk, employee ID 2) and all of the employee's subordinates in all levels (direct or indirect):

```
WITH EmpsCTE AS
(
  SELECT empid, mgrid, firstname, lastname
  FROM HR.Employees
  WHERE empid = 2

  UNION ALL

  SELECT C.empid, C.mgrid, C.firstname, C.lastname
  FROM EmpsCTE AS P
    JOIN HR.Employees AS C
      ON C.mgrid = P.empid
)
SELECT empid, mgrid, firstname, lastname
FROM EmpsCTE;
```

The anchor member queries the HR.Employees table and simply returns the row for employee 2:

```
  SELECT empid, mgrid, firstname, lastname
  FROM HR.Employees
  WHERE empid = 2
```

The recursive member joins the CTE—representing the previous result set—with the Employees table to return the direct subordinates of the employees returned in the previous result set:

```
  SELECT C.empid, C.mgrid, C.firstname, C.lastname
  FROM EmpsCTE AS P
    JOIN HR.Employees AS C
      ON C.mgrid = P.empid
```

In other words, the recursive member is invoked repeatedly, and in each invocation it returns the next level of subordinates. The first time the recursive member is invoked it returns the direct subordinates of employee 2—employees 3 and 5. The second time the recursive member is invoked, it returns the direct subordinates of employees 3 and 5—employees 4, 6, 7, 8, and 9. The third time the recursive member is invoked, there are no more subordinates; the recursive member returns an empty set and therefore recursion stops.

The reference to the CTE name in the outer query represents the unified result sets; in other words, employee 2 and all of the employee's subordinates.

Here's the output of this code:

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

In the event of a logical error in the join predicate in the recursive member, or problems with the data resulting in cycles, the recursive member can potentially be invoked an infinite number of times. As a safety measure, by default SQL Server restricts the number of times that the recursive member can be invoked to 100. The code will fail upon the 101st invocation of the recursive member. You can change the default maximum recursion limit by specifying the hint OPTION(MAXRECURSION *n*) at the end of the outer query, where *n* is an integer in the range 0 through 32,767 representing the maximum recursion limit you want to set. If you want to remove the restriction altogether, specify MAXRECURSION 0. Note that SQL Server stores the intermediate result sets returned by the anchor and recursive members in a work table in tempdb; if you remove the restriction and have a runaway query, the work table will quickly get very large. If tempdb can't grow anymore—for example, when you run out of disk space—the query will fail.

Views

The two types of table expressions discussed so far—derived tables and CTEs—have a very limited scope, which is the single statement scope. As soon as the outer query against those table expressions is finished, they are gone. This means that derived tables and CTEs are not reusable.

Views and inline table-valued functions (inline TVFs) are two reusable types of table expressions; their definition is stored as a database object. Once created, those objects are permanent parts of the database and are only removed from the database if explicitly dropped.

In most other respects, views and inline TVFs are treated like derived tables and CTEs. For example, when querying a view or an inline TVF, SQL Server expands the definition of the table expression and queries the underlying objects directly, as with derived tables and CTEs.

In this section, I'll describe views; in the next section, I'll describe inline TVFs. As I mentioned earlier, a view is a reusable table expression whose definition is stored in the database. For example, the following code creates a view called USACusts in the Sales schema in the TSQLFundamentals2008 database, representing all customers from the United States:

```
USE TSQLFundamentals2008;
IF OBJECT_ID('Sales.USACusts') IS NOT NULL
  DROP VIEW Sales.USACusts;
GO
CREATE VIEW Sales.USACusts
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

Note that just as with derived tables and CTEs, instead of using inline column aliasing as shown in the preceding code, you can use external column aliasing by specifying the target column names in parentheses immediately after the view name.

Once you create this view, you can query it much like you query other tables in the database:

```
SELECT custid, companyname
FROM Sales.USACusts;
```

Because a view is an object in the database, you can control access to the view with permissions just like other objects that can be queried (for example, SELECT, INSERT, UPDATE, and DELETE permissions). For example, you can deny direct access to the underlying objects while granting access to the view.

Note that the general recommendation to avoid using SELECT * has specific relevance in the context of views. The columns are enumerated in the compiled form of the view and new table columns will not be automatically added to the view. For example, suppose you define a view based on the query SELECT * FROM dbo.T1, and at the view creation time the table T1 has the columns col1 and col2. SQL Server stores information only on those two columns in the view's metadata. If you alter the definition of the table adding new columns, those new columns will not be added to the view. You can refresh the view's metadata using a stored procedure called sp_refreshview, but to avoid confusion, the best practice is to explicitly list the column names that you need in the definition of the view. If columns are added to the underlying tables and you need them in the view, use the *ALTER VIEW* statement to revise the view definition accordingly.

Views and the ORDER BY Clause

The query that you use to define a view must meet all requirements mentioned earlier with respect to table expressions in the context of derived tables. The view should not guarantee any order to the rows, all view columns must have names, and all column names must be unique. In this section, I'll elaborate a bit about the ordering issue, which is a fundamental point that is crucial to understand.

Remember that a presentation ORDER BY clause is not allowed in the query defining a table expression because there's no order among the rows of a relational table. An attempt to create an ordered view is absurd because it violates fundamental properties of a relation as defined by the relational model. If you need to return rows from a view sorted for presentation purposes, you shouldn't try to make the view something it shouldn't be. Instead, you should specify a presentation ORDER BY clause in the outer query against the view, like so:

```
SELECT custid, companyname, region
FROM Sales.USACusts
ORDER BY region;
```

Try running the following code to create a view with a presentation ORDER BY clause:

```
ALTER VIEW Sales.USACusts
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA'
ORDER BY region;
GO
```

This attempt fails and you get the following error:

```
Msg 1033, Level 15, State 1, Procedure USACusts, Line 9
The ORDER BY clause is invalid in views, inline functions, derived tables, subqueries, and
common table expressions, unless TOP or FOR XML is also specified.
```

The error message indicates that SQL Server allows the ORDER BY clause in two exceptional cases—when the TOP or FOR XML options are used. Neither case follows the SQL standard, and in both cases the ORDER BY clause serves a purpose beyond the usual presentation purpose.

Because T-SQL allows an ORDER BY clause in a view when TOP is also specified, some people think that they can create "ordered views" by using TOP (100) PERCENT like so:

```
ALTER VIEW Sales.USACusts
AS

SELECT TOP (100)
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
```

```
FROM Sales.Customers
WHERE country = N'USA'
ORDER BY region;
GO
```

Even though the code is technically valid and the view is created, you should be aware that because the query is used to define a table expression, the ORDER BY clause here is only guaranteed to serve the logical filtering purpose for the TOP option. If you query the view and don't specify an ORDER BY clause in the outer query, presentation order is not guaranteed.

For example, run the following query against the view:

```
SELECT custid, companyname, region
FROM Sales.USACusts;
```

Here is the output from one of my executions showing that the rows are not sorted by region:

```
custid      companyname              region
----------- ------------------------ ---------------
32          Customer YSIQX           OR
36          Customer LVJSO           OR
43          Customer UISOJ           WA
45          Customer QXPPT           CA
48          Customer DVFMB           OR
55          Customer KZQZT           AK
65          Customer NYUHS           NM
71          Customer LCOUJ           ID
75          Customer XOJYP           WY
77          Customer LCYBZ           OR
78          Customer NLTYP           MT
82          Customer EYHKM           WA
89          Customer YBQTI           WA
```

In some cases a query that is used to define a table expression has the TOP option with an ORDER BY clause, and the query against the table expression doesn't have an ORDER BY clause. In those cases, therefore, the output might or might not be returned in the specified order. If the results happen to be ordered, it may be due to optimization reasons, especially when you use values other than TOP (100) PERCENT. The point I'm trying to make is that any order of the rows in the output is considered valid, and no specific order is guaranteed; therefore, when querying a table expression, you should not assume any order unless you specify an ORDER BY clause in the outer query.

Do not confuse the behavior of a query that is used to define a table expression with a query that isn't. A query with TOP and ORDER BY does not guarantee presentation order only in the context of a table expression. In the context of a query that is not used to define a table expression, the ORDER BY clause serves both the logical filtering purpose for the TOP option and the presentation purpose.

View Options

When you create or alter a view, you can specify view attributes and options as part of the view definition. In the header of the view under the WITH clause you can specify attributes such as ENCRYPTION and SCHEMABINDING, and at the end of the query you can specify WITH CHECK OPTION. The following sections describe the purpose of these options.

The ENCRYPTION Option

The ENCRYPTION option is available when you create or alter views, stored procedures, triggers, and user-defined functions (UDFs). The ENCRYPTION option indicates that SQL Server will internally store the text with the definition of the object in an obfuscated format. The obfuscated text is not directly visible to users through any of the catalog objects—only to privileged users through special means.

Before you look at the ENCRYPTION option, run the following code to alter the definition of the USACusts view to its original version:

```
ALTER VIEW Sales.USACusts
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

To get the definition of the view, invoke the *OBJECT_DEFINITION* function like so:

```
SELECT OBJECT_DEFINITION(OBJECT_ID('Sales.USACusts'));
```

The text with the definition of the view is available because the view was created without the ENCRYPTION option. You get the following output:

```
CREATE VIEW Sales.USACusts
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
```

Next, alter the view definition—only this time, include the ENCRYPTION option:

```
ALTER VIEW Sales.USACusts WITH ENCRYPTION
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
```

```
FROM Sales.Customers
WHERE country = N'USA';
GO
```

Try again to get the text with the definition of the view:

```
SELECT OBJECT_DEFINITION(OBJECT_ID('Sales.USACusts'));
```

This time you get a NULL back.

As an alternative to the *OBJECT_DEFINITION* function, you can use the sp_helptext stored procedure to get object definitions. The *OBJECT_DEFINITION* function was added in SQL Server 2005 while sp_helptext was also available in earlier versions. For example, the following code requests the object definition of the USACusts view:

```
EXEC sp_helptext 'Sales.USACusts';
```

Because in our case the view was created with the ENCRYPTION option, you will not get the object definition back, but the following message:

```
The text for object 'Sales.USACusts' is encrypted.
```

The SCHEMABINDING Option

The SCHEMABINDING option is available to views and UDFs, and it binds the schema of referenced objects and columns to the schema of the referencing object. It indicates that referenced objects cannot be dropped and that referenced columns cannot be dropped or altered.

For example, alter the USACusts view with the SCHEMABINDING option:

```
ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA';
GO
```

Now try to drop the Address column from the Customers table:

```
ALTER TABLE Sales.Customers DROP COLUMN address;
```

You get the following error:

```
Msg 5074, Level 16, State 1, Line 1
The object 'USACusts' is dependent on column 'address'.
Msg 4922, Level 16, State 9, Line 1
ALTER TABLE DROP COLUMN address failed because one or more objects access this column.
```

Without the SCHEMABINDING option, such a schema change would have been allowed, as well as dropping the Customers table altogether. This can lead to errors at run time when

you try to query the view, and referenced objects or columns that do not exist. If you create the view with the SCHEMABINDING option, you can avoid these errors.

The object definition must meet a couple of technical requirements to support the SCHEMABINDING option. The query is not allowed to use * in the SELECT clause; instead, you have to explicitly list column names. Also, you must use schema-qualified two-part names when referring to objects. Both requirements are actually good practices in general.

As you can imagine, creating your objects with the SCHEMABINDING option is a good practice.

The Option CHECK OPTION

The purpose of CHECK OPTION is to prevent modifications through the view that conflict with the view's filter—assuming that one exists in the query defining the view.

The query defining the view USACusts filters customers where the country attribute is equal to N'USA'. The view is currently defined without CHECK OPTION. This means that you can currently insert rows through the view with customers from countries other than the United States, and you can update existing customers through the view, changing their country to one other than the United States. For example, the following code successfully inserts a customer with company name Customer ABCDE from the United Kingdom through the view:

```
INSERT INTO Sales.USACusts(
  companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax)
 VALUES(
  N'Customer ABCDE', N'Contact ABCDE', N'Title ABCDE', N'Address ABCDE',
  N'London', NULL, N'12345', N'UK', N'012-3456789', N'012-3456789');
```

The row was inserted through the view into the Customers table. However, because the view filters only customers from the United States, if you query the view looking for the new customer you get an empty set back:

```
SELECT custid, companyname, country
FROM Sales.USACusts
WHERE companyname = N'Customer ABCDE';
```

Query the Customers table directly looking for the new customer:

```
SELECT custid, companyname, country
FROM Sales.Customers
WHERE companyname = N'Customer ABCDE';
```

You get the customer information in the output, because the new row made it to the Customers table:

```
custid      companyname        country
----------- ------------------ ---------------
92          Customer ABCDE     UK
```

Similarly, if you update a customer row through the view, changing the country attribute to a country other than the United States, the update makes it to the table. But that customer doesn't show up anymore in the view because it doesn't qualify to the view's query filter.

If you want to prevent modifications that conflict with the view's filter, add WITH CHECK OPTION at the end of the query defining the view:

```
ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS

SELECT
  custid, companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'USA'
WITH CHECK OPTION;
GO
```

Now try to insert a row that conflicts with the view's filter:

```
INSERT INTO Sales.USACusts(
  companyname, contactname, contacttitle, address,
  city, region, postalcode, country, phone, fax)
 VALUES(
  N'Customer FGHIJ', N'Contact FGHIJ', N'Title FGHIJ', N'Address FGHIJ',
  N'London', NULL, N'12345', N'UK', N'012-3456789', N'012-3456789');
```

You get the following error:

```
Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view either specifies WITH CHECK
OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows resulting from
the operation did not qualify under the CHECK OPTION constraint.
The statement has been terminated.
```

When you're done, run the following code for cleanup:

```
DELETE FROM Sales.Customers
WHERE custid > 91;

DBCC CHECKIDENT('Sales.Customers', RESEED, 91);

IF OBJECT_ID('Sales.USACusts') IS NOT NULL DROP VIEW Sales.USACusts;
```

Inline Table-Valued Functions

Inline TVFs are reusable table expressions that support input parameters. In all respects except for the support for input parameters, inline TVFs are similar to views. For this reason, I like to think of inline TVFs as parameterized views, even though they are not called this formally.

For example, the following code creates an inline TVF called *fn_GetCustOrders* in the TSQLFundamentals2008 database:

```
USE TSQLFundamentals2008;
IF OBJECT_ID('dbo.fn_GetCustOrders') IS NOT NULL
  DROP FUNCTION dbo.fn_GetCustOrders;
GO
CREATE FUNCTION dbo.fn_GetCustOrders
  (@cid AS INT) RETURNS TABLE
AS
RETURN
  SELECT orderid, custid, empid, orderdate, requireddate,
    shippeddate, shipperid, freight, shipname, shipaddress, shipcity,
    shipregion, shippostalcode, shipcountry
  FROM Sales.Orders
  WHERE custid = @cid;
GO
```

This inline TVF accepts an input parameter called *@cid* representing a customer ID, and returns all orders that were placed by the input customer. You query inline TVFs like you query other tables with DML statements. If the function accepts input parameters, you specify those in parentheses following the function's name. Also, make sure you provide an alias to the table expression. Providing a table expression with an alias is not always a requirement but is a good practice because it makes your code more readable and less prone to errors. For example, the following code queries the function requesting all orders that were placed by customer 1:

```
SELECT orderid, custid
FROM dbo.fn_GetCustOrders(1) AS CO;
```

This code returns the following output:

```
orderid     custid
----------- -----------
10643       1
10692       1
10702       1
10835       1
10952       1
11011       1
```

As with other tables, you can refer to an inline TVF as part of a join. For example, the following query joins the inline TVF returning customer 1's orders with the Sales.OrderDetails table, matching customer 1's orders with the related order lines:

```
SELECT CO.orderid, CO.custid, OD.productid, OD.qty
FROM dbo.fn_GetCustOrders(1) AS CO
  JOIN Sales.OrderDetails AS OD
    ON CO.orderid = OD.orderid;
```

This code returns the following output:

```
orderid     custid      productid   qty
----------- ----------- ----------- ------
10643       1           28          15
10643       1           39          21
10643       1           46          2
10692       1           63          20
10702       1           3           6
10702       1           76          15
10835       1           59          15
10835       1           77          2
10952       1           6           16
10952       1           28          2
11011       1           58          40
11011       1           71          20
```

When you're done, run the following code for cleanup:

```
IF OBJECT_ID('dbo.fn_GetCustOrders') IS NOT NULL
  DROP FUNCTION dbo.fn_GetCustOrders;
```

The APPLY Operator

The APPLY operator is a nonstandard table operator that was introduced in SQL Server 2005. This operator is used in the FROM clause of a query like all table operators. The two supported types of the APPLY operator are CROSS APPLY and OUTER APPLY. CROSS APPLY implements only one logical query processing phase, while OUTER APPLY implements two.

The APPLY operator operates on two input tables, the second of which may be a table expression; I'll refer to them as the left and right tables. The right table is usually a derived table or an inline TVF. The CROSS APPLY operator implements one logical query processing phase—it applies the right table expression to each row from the left table, and produces a result table with the unified result sets.

So far it might sound like the CROSS APPLY operator is very similar to a cross join, and in a sense that's true. For example, the following two queries return the same result sets:

```
SELECT S.shipperid, E.empid
FROM Sales.Shippers AS S
  CROSS JOIN HR.Employees AS E;

SELECT S.shipperid, E.empid
FROM Sales.Shippers AS S
  CROSS APPLY HR.Employees AS E;
```

However, with the CROSS APPLY operator the right table expression can represent a different set of rows per each row from the left table, unlike in a join. You can achieve this when you use a derived table in the right side, and in the derived table query refer to attributes from the left side. Or when you use an inline TVF, you can pass attributes from the left side as input arguments.

For example, the following code uses the CROSS APPLY operator to return the three most recent orders for each customer:

```
SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
  CROSS APPLY
    (SELECT TOP(3) orderid, empid, orderdate, requireddate
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
     ORDER BY orderdate DESC, orderid DESC) AS A;
```

You can think of the table expression A as a correlated table subquery. In terms of logical query processing, the right table expression (derived table in our case) is applied to each row from the Customers table. Notice the reference to the attribute C.custid from the left table in the derived table's query filter. The derived table returns the three most recent orders for the customer from the current left row. Because the derived table is applied to each row from the left side, the CROSS APPLY operator returns the three most recent orders for each customer.

Here's the output of this query, shown here in abbreviated form:

```
custid      orderid     orderdate
----------- ----------- -----------------------
1           11011       2008-04-09 00:00:00.000
1           10952       2008-03-16 00:00:00.000
1           10835       2008-01-15 00:00:00.000
2           10926       2008-03-04 00:00:00.000
2           10759       2007-11-28 00:00:00.000
2           10625       2007-08-08 00:00:00.000
3           10856       2008-01-28 00:00:00.000
3           10682       2007-09-25 00:00:00.000
3           10677       2007-09-22 00:00:00.000
...

(263 row(s) affected)
```

If the right table expression returns an empty set, the CROSS APPLY operator does not return the corresponding left row. For example, customers 22 and 57 did not place orders. In both cases the derived table is an empty set; therefore, those customers are not returned in the output. If you want to return rows from the left table for which the right table expression returns an empty set, use the OUTER APPLY operator instead of CROSS APPLY. The OUTER APPLY operator adds a second logical phase that identifies rows from the left side for which the right table expression returns an empty set, and adds those rows to the result table as outer rows with NULLs in the right side's attributes as place holders. In a sense, this phase is similar to the phase that adds outer rows in a left outer join.

For example, run the following code to return the three most recent orders for each customer, and include in the output customers with no orders as well:

```
SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
  OUTER APPLY
```

```
    (SELECT TOP(3) orderid, empid, orderdate, requireddate
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
     ORDER BY orderdate DESC, orderid DESC) AS A;
```

This time, customers 22 and 57, who did not place orders, are included in the output, which is shown here in abbreviated form:

```
custid      orderid     orderdate
----------- ----------- -----------------------
1           11011       2008-04-09 00:00:00.000
1           10952       2008-03-16 00:00:00.000
1           10835       2008-01-15 00:00:00.000
2           10926       2008-03-04 00:00:00.000
2           10759       2007-11-28 00:00:00.000
2           10625       2007-08-08 00:00:00.000
3           10856       2008-01-28 00:00:00.000
3           10682       2007-09-25 00:00:00.000
3           10677       2007-09-22 00:00:00.000
...
22          NULL        NULL
...
57          NULL        NULL
...

(265 row(s) affected)
```

For encapsulation purposes you may find it more convenient to work with inline TVFs instead of derived tables. This way your code will be simpler to follow and maintain. For example, the following code creates an inline TVF called *fn_TopOrders* that accepts as inputs a customer ID (*@custid*) and a number (*@n*), and returns the *@n* most recent orders for customer *@custid*:

```
IF OBJECT_ID('dbo.fn_TopOrders') IS NOT NULL
  DROP FUNCTION dbo.fn_TopOrders;
GO
CREATE FUNCTION dbo.fn_TopOrders
  (@custid AS INT, @n AS INT)
  RETURNS TABLE
AS
RETURN
  SELECT TOP(@n) orderid, empid, orderdate, requireddate
  FROM Sales.Orders
  WHERE custid = @custid
  ORDER BY orderdate DESC, orderid DESC;
GO
```

You can now substitute the use of the derived table from the previous examples with the new function:

```
SELECT
  C.custid, C.companyname,
  A.orderid, A.empid, A.orderdate, A.requireddate
FROM Sales.Customers AS C
  CROSS APPLY dbo.fn_TopOrders(C.custid, 3) AS A;
```

The code is much more readable and easier to maintain. In terms of physical processing, nothing really changed because, as I stated earlier, the definition of table expressions is expanded, and SQL Server will in any case end up querying the underlying objects directly.

Conclusion

Table expressions can help you simplify your code, improve its maintainability, and encapsulate querying logic. When you need to use table expressions and are not planning to reuse their definitions, use derived tables or CTEs. CTEs have a couple of advantages over derived tables; you do not nest CTEs as you do derived tables, making CTEs more modular and easier to maintain. Also, you can refer to multiple instances of the same CTE, which you cannot do with derived tables.

When you need to define reusable table expressions, use views or inline TVFs. When you do not need to support input parameters, use views; otherwise, use inline TVFs.

Use the APPLY operator when you want to apply a table expression to each row from a source table, and unify all result sets into one result table.

Exercises

This section provides exercises to help you familiarize yourself with the subjects discussed in this chapter. All the exercises in this chapter require your session to be connected to the database TSQLFundamentals2008.

1-1

Write a query that returns the maximum order date for each employee.

Tables involved: TSQLFundamentals2008 database, Sales.Orders table.

Desired output:

```
empid       maxorderdate
----------- -----------------------
3           2008-04-30 00:00:00.000
6           2008-04-23 00:00:00.000
9           2008-04-29 00:00:00.000
7           2008-05-06 00:00:00.000
1           2008-05-06 00:00:00.000
4           2008-05-06 00:00:00.000
2           2008-05-05 00:00:00.000
5           2008-04-22 00:00:00.000
8           2008-05-06 00:00:00.000

(9 row(s) affected)
```

1-2

Encapsulate the query from Exercise 1-1 in a derived table. Write a join query between the derived table and the Orders table to return the orders with the maximum order date for each employee.

Tables involved: Sales.Orders.

Desired output:

```
empid       orderdate               orderid     custid
----------- ----------------------- ----------- -----------
9           2008-04-29 00:00:00.000 11058       6
8           2008-05-06 00:00:00.000 11075       68
7           2008-05-06 00:00:00.000 11074       73
6           2008-04-23 00:00:00.000 11045       10
5           2008-04-22 00:00:00.000 11043       74
4           2008-05-06 00:00:00.000 11076       9
3           2008-04-30 00:00:00.000 11063       37
2           2008-05-05 00:00:00.000 11073       58
2           2008-05-05 00:00:00.000 11070       44
1           2008-05-06 00:00:00.000 11077       65

(10 row(s) affected)
```

2-1

Write a query that calculates a row number for each order based on orderdate, orderid ordering.

Tables involved: Sales.Orders.

Desired output (abbreviated):

```
orderid     orderdate               custid      empid       rownum
----------- ----------------------- ----------- ----------- -------
10248       2006-07-04 00:00:00.000 85          5           1
10249       2006-07-05 00:00:00.000 79          6           2
10250       2006-07-08 00:00:00.000 34          4           3
10251       2006-07-08 00:00:00.000 84          3           4
10252       2006-07-09 00:00:00.000 76          4           5
10253       2006-07-10 00:00:00.000 34          3           6
10254       2006-07-11 00:00:00.000 14          5           7
10255       2006-07-12 00:00:00.000 68          9           8
10256       2006-07-15 00:00:00.000 88          3           9
10257       2006-07-16 00:00:00.000 35          4           10
...

(830 row(s) affected)
```

2-2

Write a query that returns rows with row numbers 11 through 20 based on the row number definition in Exercise 2-1. Use a CTE to encapsulate the code from Exercise 2-1.

Tables involved: Sales.Orders.

Desired output:

```
orderid     orderdate               custid      empid       rownum
----------- ----------------------- ----------- ----------- -------
10258       2006-07-17 00:00:00.000 20          1           11
10259       2006-07-18 00:00:00.000 13          4           12
10260       2006-07-19 00:00:00.000 56          4           13
10261       2006-07-19 00:00:00.000 61          4           14
10262       2006-07-22 00:00:00.000 65          8           15
10263       2006-07-23 00:00:00.000 20          9           16
10264       2006-07-24 00:00:00.000 24          6           17
10265       2006-07-25 00:00:00.000 7           2           18
10266       2006-07-26 00:00:00.000 87          3           19
10267       2006-07-29 00:00:00.000 25          4           20

(10 row(s) affected)
```

3

Write a solution using a recursive CTE that returns the management chain leading to Zoya Dolgopyatova (employee ID 9).

Tables involved: HR.Employees.

Desired output:

```
empid       mgrid       firstname  lastname
----------- ----------- ---------- --------------------
9           5           Zoya       Dolgopyatova
5           2           Sven       Buck
2           1           Don        Funk
1           NULL        Sara       Davis

(4 row(s) affected)
```

4-1

Create a view that returns the total quantity for each employee and year.

Tables involved: Sales.Orders and Sales.OrderDetails.

When running the following code:

```
SELECT * FROM Sales.VEmpOrders ORDER BY empid, orderyear;
```

The desired output is:

```
empid       orderyear   qty
----------- ----------- -----------
1           2006        1620
1           2007        3877
1           2008        2315
2           2006        1085
2           2007        2604
2           2008        2366
3           2006        940
3           2007        4436
3           2008        2476
4           2006        2212
4           2007        5273
4           2008        2313
5           2006        778
5           2007        1471
5           2008        787
6           2006        963
6           2007        1738
6           2008        826
7           2006        485
7           2007        2292
7           2008        1877
8           2006        923
8           2007        2843
8           2008        2147
9           2006        575
9           2007        955
9           2008        1140

(27 row(s) affected)
```

4-2 (Optional, Advanced)

Write a query against Sales.VEmpOrders that returns the running total quantity for each employee and year.

Tables involved: Sales.VEmpOrders view.

Desired output:

```
empid       orderyear   qty         runqty
----------- ----------- ----------- -----------
1           2006        1620        1620
1           2007        3877        5497
1           2008        2315        7812
2           2006        1085        1085
2           2007        2604        3689
2           2008        2366        6055
3           2006        940         940
```

```
3           2007        4436        5376
3           2008        2476        7852
4           2006        2212        2212
4           2007        5273        7485
4           2008        2313        9798
5           2006        778         778
5           2007        1471        2249
5           2008        787         3036
6           2006        963         963
6           2007        1738        2701
6           2008        826         3527
7           2006        485         485
7           2007        2292        2777
7           2008        1877        4654
8           2006        923         923
8           2007        2843        3766
8           2008        2147        5913
9           2006        575         575
9           2007        955         1530
9           2008        1140        2670

(27 row(s) affected)
```

5-1

Create an inline function that accepts as inputs a supplier ID (@*supid* AS *INT*) and a requested number of products (@*n* AS *INT*). The function should return @*n* products with the highest unit prices that are supplied by the given supplier ID.

Tables involved: Production.Products.

When issuing the following query:

```
SELECT * FROM Production.fn_TopProducts(5, 2);
```

Desired output:

```
productid   productname        unitprice
----------- ------------------ ---------------
12          Product OSFNS      38.00
11          Product QMVUN      21.00

(2 row(s) affected)
```

5-2

Using the CROSS APPLY operator and the function you created in Exercise 4-1, return, for each supplier, the two most expensive products.

Desired output:

```
supplierid  companyname       productid   productname     unitprice
----------- ----------------- ----------- --------------- ----------
8           Supplier BWGYE    20          Product QHFFP   81.00
8           Supplier BWGYE    68          Product TBTBL   12.50
20          Supplier CIYNM    43          Product ZZZHR   46.00
20          Supplier CIYNM    44          Product VJIEO   19.45
23          Supplier ELCRN    49          Product FPYPN   20.00
23          Supplier ELCRN    76          Product JYGFE   18.00
5           Supplier EQPNC    12          Product OSFNS   38.00
5           Supplier EQPNC    11          Product QMVUN   21.00
...

(55 row(s) affected)
```

Solutions

This section provides solutions to the exercises in the preceding section.

1-1

This exercise is just a preliminary step to the next exercise. This step involves writing a query that returns the maximum order date for each employee:

```
USE TSQLFundamentals2008;

SELECT empid, MAX(orderdate) AS maxorderdate
FROM Sales.Orders
GROUP BY empid;
```

1-2

This exercise requires you to use the query from the previous step to define a derived table, and join this derived table with the Orders table to return the orders with the maximum order date for each employee, like so:

```
SELECT O.empid, O.orderdate, O.orderid, O.custid
FROM Sales.Orders AS O
  JOIN (SELECT empid, MAX(orderdate) AS maxorderdate
        FROM Sales.Orders
        GROUP BY empid) AS D
    ON O.empid = D.empid
    AND O.orderdate = D.maxorderdate;
```

2-1

This exercise is a preliminary step to the next exercise. It requires you to query the Orders table and calculate row numbers based on orderdate, orderid ordering, like so:

```
SELECT orderid, orderdate, custid, empid,
  ROW_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum
FROM Sales.Orders;
```

2-2

This exercise requires you to define a CTE based on the query from the previous step, and filter only rows with row numbers in the range 11 through 20 from the CTE, like so:

```
WITH OrdersRN AS
(
  SELECT orderid, orderdate, custid, empid,
    ROW_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum
  FROM Sales.Orders
)
SELECT * FROM OrdersRN WHERE rownum BETWEEN 11 AND 20;
```

You might wonder why you need a table expression here. Remember that calculations based on the OVER clause (such as the *ROW_NUMBER* function) are only allowed in the SELECT and ORDER BY clauses of a query, and not directly in the WHERE clause. By using a table expression you can invoke the *ROW_NUMBER* function in the SELECT clause, assign an alias to the result column, and refer to the result column in the WHERE clause of the outer query.

3

You can think of this exercise as the inverse of the request to return an employee and all subordinates in all levels. Here, the anchor member is a query that returns the row for employee 9. The recursive member joins the CTE (call it C)—representing the subordinate/child from the previous level—with the Employees table (call it P)—representing the manager/parent in the next level. This way, each invocation of the recursive member returns the manager from the next level, until no next level manager is found (in the case of the CEO).

Here's the complete solution query:

```
WITH EmpsCTE AS
(
  SELECT empid, mgrid, firstname, lastname
  FROM HR.Employees
  WHERE empid = 9

  UNION ALL
```

```
  SELECT P.empid, P.mgrid, P.firstname, P.lastname
  FROM EmpsCTE AS C
    JOIN HR.Employees AS P
      ON C.mgrid = P.empid
)
SELECT empid, mgrid, firstname, lastname
FROM EmpsCTE;
```

4-1

This exercise is a preliminary step to the next exercise. Here you are required to define a view based on a query that joins the Orders and OrderDetails tables, group the rows by employee ID and order year, and return the total quantity for each group. The view definition should look like this:

```
USE TSQLFundamentals2008;
IF OBJECT_ID('Sales.VEmpOrders') IS NOT NULL
  DROP VIEW Sales.VEmpOrders;
GO
CREATE VIEW  Sales.VEmpOrders
AS

SELECT
  empid,
  YEAR(orderdate) AS orderyear,
  SUM(qty) AS qty
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY
  empid,
  YEAR(orderdate);
GO
```

4-2

In this exercise, you query the VEmpOrders view and return the running total quantity for each employee and order year. To achieve this, you can write a query against the VEmpOrders view (call it V1) that returns from each row the employee ID, order year, and quantity. In the SELECT list you can incorporate a subquery against a second instance of VEmpOrders (call it V2), that returns the sum of all quantities from the rows where the employee ID is equal to the one in V1, and the order year is smaller than or equal to the one in V1. The complete solution query looks like this:

```
SELECT empid, orderyear, qty,
  (SELECT SUM(qty)
   FROM  Sales.VEmpOrders AS V2
   WHERE V2.empid = V1.empid
     AND V2.orderyear <= V1.orderyear) AS runqty
FROM  Sales.VEmpOrders AS V1
ORDER BY empid, orderyear;
```

5-1

This exercise requires you to define a function called *fn_TopProducts* that accepts a supplier ID (*@supid*) and a number (*@n*), and is supposed to return the *@n* most expensive products supplied by the input supplier ID. Here's how the function definition should look:

```
USE TSQLFundamentals2008;
IF OBJECT_ID('Production.fn_TopProducts') IS NOT NULL
  DROP FUNCTION Production.fn_TopProducts;
GO
CREATE FUNCTION Production.fn_TopProducts
  (@supid AS INT, @n AS INT)
  RETURNS TABLE
AS
RETURN
  SELECT TOP(@n) productid, productname, unitprice
  FROM Production.Products
  WHERE supplierid = @supid
  ORDER BY unitprice DESC;
GO
```

5-2

In this exercise, you write a query against the Production.Suppliers table, and use the CROSS APPLY operator to apply the function you defined by the previous step to each supplier. Your query is supposed to return the two most expensive products for each supplier. Here's the solution query:

```
SELECT S.supplierid, S.companyname, P.productid, P.productname, P.unitprice
FROM Production.Suppliers AS S
  CROSS APPLY Production.fn_TopProducts(S.supplierid, 2) AS P;
```

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Itzik Ben-Gan

Itzik is a mentor and cofounder of Solid Quality Mentors. A SQL Server Microsoft MVP (Most Valuable Professional) since 1999, Itzik has delivered numerous training events around the world 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 Magazine* as well as articles and white papers for MSDN. Itzik's speaking engagements include Tech Ed, DevWeek, PASS, SQL Server Magazine Connections, presentations to various user groups around the world, and Solid Quality Mentors events.