Microsoft

Microsoft® SQL Server® 2012 T-SQL Fundamentals

Itzik Ben-Gan

SolidQ

Microsoft® SQL Server® 2012 T-SQL Fundamentals

Gain a solid understanding of T-SQL—and write better queries

Master the fundamentals of Transact-SQL—and develop your own code for querying and modifying data in Microsoft SQL Server 2012. Led by a SQL Server expert, you'll learn the concepts behind T-SQL querying and programming, and then apply your knowledge with exercises in each chapter. Once you understand the logic behind T-SQL, you'll quickly learn how to write effective code—whether you're a programmer or database administrator.

Discover how to:

- Work with programming practices unique to T-SQL
- Create database tables and define data integrity
- Query multiple tables using joins and subqueries
- Simplify code and improve maintainability with table expressions
- Implement insert, update, delete, and merge data modification strategies
- Tackle advanced techniques such as window functions, pivoting, and grouping sets
- Control data consistency using isolation levels, and mitigate deadlocks and blocking
- Take T-SQL to the next level with programmable objects

Get code samples on the web
Ready to download at
http://go.microsoft.com/FWLink/?Linkid=248717

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

About the Author

Itzik Ben-Gan, Microsoft MVP for SQL Server since 1999, is cofounder of SolidQ, where he teaches and consults internationally on T-SQL querying, programming, and query tuning. He's a frequent contributor to *SQL Server Pro* and *MSDN Magazine*, and speaks at industry events such as Microsoft TechEd, DevTeach, PASS, and SQL Server Connections.

microsoft.com/mspress

ISBN: 978-0-7356-58141

U.S.A. $49.99
Canada $52.99
[Recommended]

Databases/Microsoft SQL Server

Microsoft

Microsoft®

Microsoft® SQL Server® 2012 T-SQL Fundamentals

Itzik Ben-Gan

ISBN: 978-0-735-65814-1

Fifth Printing: March 2015

Printed and bound in the United States of America.

Acquisitions and Developmental Editor: Russell Jones

Production Editor: Kristen Borg

Editorial Production and Illustration: Online Training Solutions, Inc.

Technical Reviewer: Gianluca Hotz and Herbert Albert

Copyeditor: Kathy Krause

Indexer: Allegro Technical Indexing

Cover Design: Twist Creative • Seattle

Cover Composition: Karen Montgomery

To Dato

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

—Thomas Campbell

This page intentionally left blank

Contents at a Glance

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

I'm very happy that Itzik has managed to find the time and energy to produce a book about T-SQL fundamentals. For many years, Itzik has been using his great Microsoft SQL Server teaching, mentoring, and consulting experience to write books on *advanced* programming subjects, leaving a significant gap not only for the novice and less experienced users but also for the many experts working with SQL Server in roles where T-SQL programming is not a high priority.

When it comes to T-SQL, Itzik is one of the most knowledgeable people in the world. In fact, we (members of the SQL Server development team), turn to Itzik for expert advice on most of the new language extensions we plan to implement. His feedback and consultations have become an important part of our SQL Server development process.

It is never an easy task for a person who is a subject matter expert to write an introductory book; however, Itzik has the advantage of having taught both introductory and advanced programming classes for many years. Such experience is a great asset when differentiating the fundamental T-SQL information from the more advanced topics. But in this book, Itzik is not simply avoiding anything considered *advanced*; he is not afraid to take on inherently complex subjects such as set theory, predicate logic, and the relational model, introducing them in simple terms, and providing just enough information for readers to understand their importance to the SQL language. The result is a book that rewards readers with an understanding of not only *what* and *how* T-SQL works, but also *why*.

In programming manuals and books, there is no better way to convey the subject under discussion than with a good example. This book includes many examples—and you can download them all from Itzik's website, *http://tsql.solidq.com*. T-SQL is a dialect of the official ISO and ANSI standards for the SQL language, but it has numerous extensions that can improve the expressiveness and brevity of your T-SQL code. Many of Itzik's examples show the T-SQL dialect solution and the equivalent ANSI SQL solution to the same exercise side by side. This is a great advantage for readers who are familiar with the ANSI version of SQL but who are new to T-SQL, as well as for programmers who need to write SQL code that can be deployed easily across several different database platforms.

Itzik's deep connection to the SQL Server team shows in his explanation of the Appliance, Box, Cloud (ABC) flavors of SQL Server in Chapter 1, "Background to T-SQL Querying and Programming." So far, I have seen the term "ABC" used only internally within the Microsoft SQL Server team, but I'm sure it is only a matter of time until the term spreads around. Itzik developed and tested the examples in the book against both the "B" (box) and "C" (cloud) flavors of SQL Server. And the Appendix points out where you can get started with the cloud version of SQL Server, known as Windows Azure SQL Database. Therefore, you can use this book as a starting point for your own cloud experiences. The Azure website shows how to start your free subscription to the Azure services, so you can then execute the examples in the book.

The cloud extension of SQL Server is an extremely important point that you should not miss. I consider it to be *so* important that I'm doing something here that never should be done in a Foreword—advertising another book (sorry, Itzik, I have to do this!). My own interest and belief in cloud computing skyrocketed after reading Nicholas G. Carr's *The Big Switch* (W.W. Norton and Company, 2009), and I want to share that experience. It is a great book that compares the advancement of cloud computing to electrification in the early 1900s. My certainty in the future of cloud computing was further cemented by watching James Hamilton's "Cloud Computing Economies of Scale" presentation at the MIX10 conference (the recording is available at *http://channel9.msdn.com/events/MIX/MIX10/EX01*).

Itzik mentions one more cloud-related change that you should be aware of. We were used to multi-year gaps between SQL Server releases, but that pattern is changing significantly with the cloud; you should instead be prepared for several smaller cloud releases (called Service Updates) deployed in the Microsoft Data Centers around the world every year. Therefore, Itzik wisely documents the discrepancies between SQL Server and Windows Azure SQL Database T-SQL on his *http://tsql.solidq.com* website rather than in the book, so he can easily keep the information up to date.

Enjoy the book—and even more—enjoy the new insights into T-SQL that this book will bring to you.

Lubor Kollar, SQL Server development team, Microsoft

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 ISO and ANSI standards for SQL. You'll learn the theory behind T-SQL querying and programming and how to develop T-SQL code to query and modify data, and you'll get an overview of programmable objects.

Although this book is intended for beginners, it is not merely a set of procedures for readers to follow. 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 might 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 2012. When I discuss language elements that were introduced recently, I specify the version in which they were added.

Besides being available in an on-premises flavor, SQL Server is also available as a cloud-based service called Windows Azure SQL Database (formerly called SQL Azure). The code samples in this book were tested against both on-premises SQL Server and SQL Database. The book's companion website (*http://tsql.solidq.com*) provides information about compatibility issues between the flavors—for example, features that are available in SQL Server 2012 but not yet in SQL Database.

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 Should Read This Book

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

Assumptions

To get the most out of this book, you should have working experience with Windows and with applications based on Windows. You should also be familiar with basic concepts concerning relational database management systems.

Who Should Not Read This Book

Not every book is aimed at every possible audience. This book covers fundamentals. It is mainly aimed at T-SQL practitioners with little or no experience. With that said, several readers of the previous edition of this book have mentioned that—even though they already had years of experience—they still found the book useful for filling gaps in their knowledge.

Organization of This Book

This 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 of 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 by using the *SELECT* statement.

- Chapter 3, "Joins," covers querying multiple tables by 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, common table expressions (CTEs), views, inline table-valued functions, and the *APPLY* operator.
- Chapter 6, "Set Operators," covers the set operators *UNION*, *INTERSECT*, and *EXCEPT*.
- Chapter 7, "Beyond the Fundamentals of Querying," covers window functions, pivoting, unpivoting, 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 of 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 *TSQL2012* sample database, start writing code against SQL Server, and learn how to get help by working with SQL Server Books Online.

System Requirements

The Appendix, "Getting Started," explains which editions of SQL Server 2012 you can use to work with the code samples included with this book. Each edition of SQL Server might 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 2012." The Appendix also explains how to work with SQL Server Books Online.

If you're connecting to SQL Database, hardware and server software are handled by Microsoft, so those requirements are irrelevant in this case.

Code Samples

This book features a companion website that makes available to you all the code used in the book, the errata, and additional resources.

http://tsql.solidq.com

Refer to the Appendix, "Getting Started," for details about the source code.

Acknowledgments

Many people contributed to making this book a reality, whether directly or indirectly, and deserve thanks and recognition.

To Lilach, for giving reason to everything I do, and for not complaining about the endless hours I spend on SQL.

To my parents Mila and Gabi and to my siblings Mickey and Ina, thanks for the constant support. Thanks for accepting the fact that I'm away, which is now harder than ever. Mom, we're all counting on you to be well and are encouraged by your strength and determination. Dad, thanks for being so supportive.

To members of the Microsoft SQL Server development team; Lubor Kollar, Tobias Ternstrom, Umachandar Jayachandran (UC), and I'm sure many others. Thanks for the great effort, and thanks for all the time you spent meeting me and responding to my email messages, addressing my questions and requests for clarification. I think that SQL Server 2012 and SQL Database show great investment in T-SQL, and I hope this will continue.

To the editorial team at O'Reilly Media and Microsoft Press; to Ken Jones, thanks for all the Itzik hours you spent, and thanks for initiating the project. To Russell Jones, thanks for your efforts in taking over the project and running it from the O'Reilly side. Also thanks to Kristen Borg, Kathy Krause, and all others who worked on the book.

To Herbert Albert and Gianluca Hotz, thanks for your work as the technical editors of the book. Your edits were excellent and I'm sure they improved the book's quality and accuracy.

To SolidQ, my company for the last decade: it's gratifying to be part of such a great company that evolved to what it is today. The members of this company are much more than colleagues to me; they are partners, friends, and family. Thanks to Fernando G. Guerrero, Douglas McDowell, Herbert Albert, Dejan Sarka, Gianluca Hotz, Jeanne Reeves,

Glenn McCoin, Fritz Lechnitz, Eric Van Soldt, Joelle Budd, Jan Taylor, Marilyn Templeton, Berry Walker, Alberto Martin, Lorena Jimenez, Ron Talmage, Andy Kelly, Rushabh Mehta, Eladio Rincón, Erik Veerman, Jay Hackney, Richard Waymire, Carl Rabeler, Chris Randall, Johan Åhlén, Raoul Illyés, Peter Larsson, Peter Myers, Paul Turley, and so many others.

To members of the *SQL Server Pro* editorial team, Megan Keller, Lavon Peters, Michele Crockett, Mike Otey, and I'm sure many others; I've been writing for the magazine for more than a decade and am grateful for the opportunity to share my knowledge with the magazine's readers.

To SQL Server MVPs Alejandro Mesa, Erland Sommarskog, Aaron Bertrand, Tibor Karaszi, Paul White, and many others, and to the MVP lead, Simon Tien; this is a great program that I'm grateful and proud to be part of. The level of expertise of this group is amazing and I'm always excited when we all get to meet, both to share ideas and just to catch up at a personal level over beer. I believe that, in great part, Microsoft's inspiration to add new T-SQL capabilities in SQL Server is thanks to the efforts of SQL Server MVPs, and more generally the SQL Server community. It is great to see this synergy yielding such a meaningful and important outcome.

To Q2, Q3, and Q4, thanQ.

Finally, to my students: teaching SQL is what drives me. It's my passion. Thanks for allowing me to fulfill my calling, and for all the great questions that make me seek more knowledge.

Errata & Book Support

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

http://www.microsoftpressstore.com/title/ 9780735658141

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

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

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

We Want to Hear from You

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

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

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

Stay in Touch

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

CHAPTER 4

Subqueries

SQL supports writing queries within queries, or *nesting* queries. The outermost query is a query whose result set is returned to the caller and is known as the *outer query*. The inner query is a query whose result is used by the outer query and is known as a *subquery*. The inner query acts in place of an expression that is based on constants or variables and is evaluated at run time. Unlike the results of expressions that use constants, the result of a subquery can change, because of changes in the queried tables. When you use subqueries, you avoid the need for separate steps in your solutions that store intermediate query results in variables.

A subquery can be either self-contained or correlated. A self-contained subquery has no dependency on the outer query that it belongs to, whereas a correlated subquery does. A subquery can be single-valued, multivalued, or table-valued. That is, a subquery can return a single value (a scalar value), multiple values, or a whole table result.

This chapter focuses on subqueries that return a single value (scalar subqueries) and subqueries that return multiple values (multivalued subqueries). I'll cover subqueries that return whole tables (table subqueries) later in the book in Chapter 5, "Table Expressions."

Both self-contained and correlated subqueries can return a scalar or multiple values. I'll first describe self-contained subqueries and demonstrate both scalar and multivalued examples, and explicitly identify those as scalar or multivalued subqueries. Then I'll describe correlated subqueries, but I won't explicitly identify them as scalar or multivalued, assuming that you will already understand the difference.

Again, exercises at the end of the chapter can help you practice what you've learned.

Self-Contained Subqueries

Every subquery has an outer query that it belongs to. Self-contained subqueries are subqueries that are independent of the outer query that they belong to. Self-contained subqueries are very convenient to debug, because you can always highlight the subquery code, run it, and ensure that it does what it's supposed to do. Logically, it's as if the subquery code is evaluated only once before the outer query is evaluated, and then the outer query uses the result of the subquery. The following sections take a look at some concrete examples of self-contained subqueries.

Self-Contained Scalar Subquery Examples

A scalar subquery is a subquery that returns a single value—regardless of whether it is self-contained. Such a subquery can appear anywhere in the outer query where a single-valued expression can appear (such as *WHERE* or *SELECT*).

For example, suppose that you need to query the *Orders* table in the *TSQL2012* database and return information about the order that has the maximum order ID in the table. You could accomplish the task by using a variable. The code could retrieve the maximum order ID from the *Orders* table and store the result in a variable. Then the code could query the *Orders* table and filter the order where the order ID is equal to the value stored in the variable. The following code demonstrates this technique.

```
USE TSQL2012;

DECLARE @maxid AS INT = (SELECT MAX(orderid)
                         FROM Sales.Orders);

SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderid = @maxid;
```

This query returns the following output.

```
orderid      orderdate                   empid        custid
------------ --------------------------- ------------ -----------
11077        2008-05-06 00:00:00.000     1            65
```

You can substitute the technique that uses a variable with an embedded subquery. You achieve this by substituting the reference to the variable with a scalar self-contained subquery that returns the maximum order ID. This way, your solution has a single query instead of the two-step process.

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderid = (SELECT MAX(O.orderid)
                 FROM Sales.Orders AS O);
```

For a scalar subquery to be valid, it must return no more than one value. If a scalar subquery can return more than one value, it might fail at run time. The following query happens to run without failure.

```
SELECT orderid
FROM Sales.Orders
WHERE empid =
  (SELECT E.empid
   FROM HR.Employees AS E
   WHERE E.lastname LIKE N'B%');
```

The purpose of this query is to return the order IDs of orders placed by any employee whose last name starts with the letter *B*. The subquery returns employee IDs of all employees whose last names start with the letter *B*, and the outer query returns order IDs of orders where the employee ID is equal to the result of the subquery. Because an equality operator expects single-valued expressions from both sides, the subquery is considered scalar. Because the subquery can potentially return more than one value, the choices of using an equality operator and a scalar subquery here are wrong. If the subquery returns more than one value, the query fails.

This query happens to run without failure because currently the *Employees* table contains only one employee whose last name starts with *B* (Sven Buck with employee ID 5). This query returns the following output, shown here in abbreviated form.

```
orderid
-----------
10248
10254
10269
10297
10320
...
10874
10899
10922
10954
11043

(42 row(s) affected)
```

Of course, if the subquery returns more than one value, the query fails. For example, try running the query with employees whose last names start with *D*.

```
SELECT orderid
FROM Sales.Orders
WHERE empid =
  (SELECT E.empid
   FROM HR.Employees AS E
   WHERE E.lastname LIKE N'D%');
```

Apparently, two employees have a last name starting with *D* (Sara Davis and Zoya Dolgopyatova). Therefore, the query fails at run time with the following error.

```
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <,
<= , >, >= or when the subquery is used as an expression.
```

If a scalar subquery returns no value, it returns a *NULL*. Recall that a comparison with a *NULL* yields *UNKNOWN* and that query filters do not return a row for which the filter expression evaluates to *UNKNOWN*. For example, the *Employees* table currently has no employees whose last names start with *A*; therefore, the following query returns an empty set.

```
SELECT orderid
FROM Sales.Orders
WHERE empid =
  (SELECT E.empid
   FROM HR.Employees AS E
   WHERE E.lastname LIKE N'A%');
```

Self-Contained Multivalued Subquery Examples

A multivalued subquery is a subquery that returns multiple values as a single column, regardless of whether the subquery is self-contained. Some predicates, such as the *IN* predicate, operate on a multivalued subquery.

Note There are other predicates that operate on a multivalued subquery; those are *SOME*, *ANY*, and *ALL*. They are very rarely used and therefore not covered in this book.

The form of the *IN* predicate is:

<scalar_expression> IN (<multivalued subquery>)

The predicate evaluates to *TRUE* if *scalar_expression* is equal to any of the values returned by the subquery. Recall the last request discussed in the previous section—returning order IDs of orders that were handled by employees with a last name starting with a certain letter. Because more than one employee can have a last name starting with the same letter, this request should be handled with the *IN* predicate and a multivalued subquery, and not with an equality operator and a scalar subquery. For example, the following query returns order IDs of orders placed by employees with a last name starting with *D*.

```
SELECT orderid
FROM Sales.Orders
WHERE empid IN
  (SELECT E.empid
   FROM HR.Employees AS E
   WHERE E.lastname LIKE N'D%');
```

Because it uses the *IN* predicate, this query is valid with any number of values returned—none, one, or more. This query returns the following output, shown here in abbreviated form.

```
orderid
-----------
10258
10270
10275
10285
10292
...
10978
11016
11017
11022
11058

(166 row(s) affected)
```

You might wonder why you wouldn't implement this task by using a join instead of subqueries, like this.

```
SELECT O.orderid
FROM HR.Employees AS E
  JOIN Sales.Orders AS O
    ON E.empid = O.empid
WHERE E.lastname LIKE N'D%';
```

Similarly, you are likely to stumble into many other querying problems that you can solve with either subqueries or joins. In my experience, there's no reliable rule of thumb that says that a subquery is better than a join. In some cases, the database engine interprets both types of queries the same way. Sometimes joins perform better than subqueries, and sometimes the opposite is true. My approach is to first write the solution query for the specified task in an intuitive form, and if performance is not satisfactory, one of my tuning approaches is to try query revisions. Such query revisions might include using joins instead of subqueries or using subqueries instead of joins.

As another example of using multivalued subqueries, suppose that you need to write a query that returns orders placed by customers from the United States. You can write a query against the *Orders* table that returns orders where the customer ID is in the set of customer IDs of customers from the United States. You can implement the last part in a self-contained, multivalued subquery. Here's the complete solution query.

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders
WHERE custid IN
  (SELECT C.custid
   FROM Sales.Customers AS C
   WHERE C.country = N'USA');
```

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

```
custid      orderid     orderdate                   empid
----------- ----------- --------------------------- -----------
65          10262       2006-07-22 00:00:00.000     8
89          10269       2006-07-31 00:00:00.000     5
75          10271       2006-08-01 00:00:00.000     6
65          10272       2006-08-02 00:00:00.000     6
65          10294       2006-08-30 00:00:00.000     4
...
32          11040       2008-04-22 00:00:00.000     4
32          11061       2008-04-30 00:00:00.000     4
71          11064       2008-05-01 00:00:00.000     1
89          11066       2008-05-01 00:00:00.000     7
65          11077       2008-05-06 00:00:00.000     1

(122 row(s) affected)
```

As with any other predicate, you can negate the *IN* predicate with the *NOT* logical operator. For example, the following query returns customers who did not place any orders.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN
  (SELECT O.custid
   FROM Sales.Orders AS O);
```

Note that best practice is to qualify the subquery to exclude *NULL* marks. Here, to keep the example simple, I didn't exclude *NULL* marks, but later in the chapter, in the "*NULL* Trouble" section, I explain this recommendation.

The self-contained, multivalued subquery returns all customer IDs that appear in the *Orders* table. Naturally, only IDs of customers who did place orders appear in the *Orders* table. The outer query returns customers from the *Customers* table where the customer ID is not in the set of values returned by the subquery—in other words, customers who did not place orders. This query returns the following output.

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

You might wonder whether specifying a *DISTINCT* clause in the subquery can help performance, because the same customer ID can occur more than once in the *Orders* table. The database engine is smart enough to consider removing duplicates without you asking it to do so explicitly, so this isn't something you need to worry about.

The last example in this section demonstrates the use of multiple self-contained subqueries in the same query—both single-valued and multivalued. Before I describe the task at hand, run the following code to create a table called *dbo.Orders* in the *TSQL2012* database (for test purposes), and populate it with order IDs from the *Sales.Orders* table that have even-numbered order IDs.

```
USE TSQL2012;
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
CREATE TABLE dbo.Orders(orderid INT NOT NULL CONSTRAINT PK_Orders PRIMARY KEY);

INSERT INTO dbo.Orders(orderid)
  SELECT orderid
  FROM Sales.Orders
  WHERE orderid % 2 = 0;
```

I describe the *INSERT* statement in more detail in Chapter 8, "Data Modification," so don't worry if you're not familiar with it yet.

The task at hand is to return all individual order IDs that are missing between the minimum and maximum in the table. It can be quite complicated to solve this problem with a query without any helper tables. You might find the *Nums* table introduced in Chapter 3, "Joins," very useful here. Remember that the *Nums* table contains a sequence of integers, starting with 1, with no gaps. To return all missing order IDs from the *Orders* table, query the *Nums* table and filter only numbers that are between the minimum and maximum in the *dbo.Orders* table and that do not appear in the set of order IDs in the *Orders* table. You can use scalar self-contained subqueries to return the minimum and maximum order IDs and a multivalued self-contained subquery to return the set of all existing order IDs. Here's the complete solution query.

```
SELECT n
FROM dbo.Nums
WHERE n BETWEEN (SELECT MIN(O.orderid) FROM dbo.Orders AS O)
            AND (SELECT MAX(O.orderid) FROM dbo.Orders AS O)
  AND n NOT IN (SELECT O.orderid FROM dbo.Orders AS O);
```

Because the code that populated the *dbo.Orders* table filtered only even-numbered order IDs, this query returns all odd-numbered values between the minimum and maximum order IDs in the *Orders* table. The output of this query is shown here in abbreviated form.

```
n
-----------
10249
10251
10253
10255
10257
...
11067
11069
11071
11073
11075

(414 row(s) affected)
```

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

```
DROP TABLE dbo.Orders;
```

Correlated Subqueries

Correlated subqueries are subqueries that refer to attributes from the table that appears in the outer query. This means that the subquery is dependent on the outer query and cannot be invoked independently. Logically, it's as if the subquery is evaluated separately for each outer row. For example, the query in Listing 4-1 returns orders with the maximum order ID for each customer.

LISTING 4-1 Correlated Subquery

```
USE TSQL2012;

SELECT custid, orderid, orderdate, empid
FROM Sales.Orders AS O1
WHERE orderid =
  (SELECT MAX(O2.orderid)
   FROM Sales.Orders AS O2
   WHERE O2.custid = O1.custid);
```

The outer query is against an instance of the *Orders* table called *O1*; it filters orders where the order ID is equal to the value returned by the subquery. The subquery filters orders from a second instance of the *Orders* table called *O2*, where the inner customer ID is equal to the outer customer ID, and returns the maximum order ID from the filtered orders. In simpler terms, for each row in *O1*, the subquery is in charge of returning the maximum order ID for the current customer. If the order ID in *O1* and the order ID returned by the subquery match, the order ID in *O1* is the maximum for the current customer, in which case the row from *O1* is returned by the query. This query returns the following output, shown here in abbreviated form.

```
custid      orderid     orderdate                   empid
----------- ----------- --------------------------- -----------
91          11044       2008-04-23 00:00:00.000     4
90          11005       2008-04-07 00:00:00.000     2
89          11066       2008-05-01 00:00:00.000     7
88          10935       2008-03-09 00:00:00.000     4
87          11025       2008-04-15 00:00:00.000     6
...
5           10924       2008-03-04 00:00:00.000     3
4           11016       2008-04-10 00:00:00.000     9
3           10856       2008-01-28 00:00:00.000     3
2           10926       2008-03-04 00:00:00.000     4
1           11011       2008-04-09 00:00:00.000     3

(89 row(s) affected)
```

Correlated subqueries are usually much harder to figure out than self-contained subqueries. To better understand the concept of correlated subqueries, I find it useful to focus attention on a single row in the outer table and understand the logical processing that takes place for that row. For example, focus your attention on the order in the *Orders* table with order ID 10248.

```
custid      orderid     orderdate                   empid
----------- ----------- --------------------------- -----------
85          10248       2006-07-04 00:00:00.000     5
```

With respect to this outer row, when the subquery is evaluated, the correlation or reference to *O1.custid* means *85*. After substituting the correlation with *85*, you get the following.

```
SELECT MAX(O2.orderid)
FROM Sales.Orders AS O2
WHERE O2.custid = 85;
```

This query returns the order ID 10739. The outer row's order ID—10248—is compared with the inner one—10739—and because there's no match in this case, the outer row is filtered out. The subquery returns the same value for all rows in *O1* with the same customer ID, and only in one case is there a match—when the outer row's order ID is the maximum for the current customer. Thinking in such terms will make it easier for you to grasp the concept of correlated subqueries.

The fact that correlated subqueries are dependent on the outer query makes them harder to debug than self-contained subqueries. You can't just highlight the subquery portion and run it. For example, if you try to highlight and run the subquery portion in Listing 4-1, you get the following error.

```
Msg 4104, Level 16, State 1, Line 1
The multi-part identifier "O1.custid" could not be bound.
```

This error indicates that the identifier *O1.custid* cannot be bound to an object in the query, because *O1* is not defined in the query. It is only defined in the context of the outer query. To debug correlated subqueries you need to substitute the correlation with a constant, and after ensuring that the code is correct, substitute the constant with the correlation.

As another example of a correlated subquery, suppose that you need to query the *Sales.OrderValues* view and return for each order the percentage that the current order value is of the total values of all of the customer's orders. In Chapter 7, "Beyond the Fundamentals of Querying," I provide a solution to this problem that uses window functions; here I'll explain how to solve the problem by using subqueries. It's always a good idea to try to come up with several solutions to each problem, because the different solutions will usually vary in complexity and performance.

You can write an outer query against an instance of the *OrderValues* view called *O1*; in the *SELECT* list, divide the current value by the result of a correlated subquery that returns the total value from a second instance of *OrderValues* called *O2* for the current customer. Here's the complete solution query.

```
SELECT orderid, custid, val,
  CAST(100. * val / (SELECT SUM(O2.val)
                     FROM Sales.OrderValues AS O2
                     WHERE O2.custid = O1.custid)
       AS NUMERIC(5,2)) AS pct
FROM Sales.OrderValues AS O1
ORDER BY custid, orderid;
```

The *CAST* function is used to convert the datatype of the expression to *NUMERIC* with a precision of 5 (the total number of digits) and a scale of 2 (the number of digits after the decimal point).

This query returns the following output.

```
orderid     custid      val        pct
----------- ----------- ---------- ------
10643       1           814.50     19.06
10692       1           878.00     20.55
10702       1           330.00     7.72
10835       1           845.80     19.79
10952       1           471.20     11.03
11011       1           933.50     21.85
10308       2           88.80      6.33
10625       2           479.75     34.20
10759       2           320.00     22.81
10926       2           514.40     36.67
...

(830 row(s) affected)
```

The *EXISTS* Predicate

T-SQL supports a predicate called *EXISTS* that accepts a subquery as input and returns *TRUE* if the subquery returns any rows and *FALSE* otherwise. For example, the following query returns customers from Spain who placed orders.

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE country = N'Spain'
  AND EXISTS
    (SELECT * FROM Sales.Orders AS O
     WHERE O.custid = C.custid);
```

The outer query against the *Customers* table filters only customers from Spain for whom the *EXISTS* predicate returns *TRUE*. The *EXISTS* predicate returns *TRUE* if the current customer has related orders in the *Orders* table.

One of the benefits of using the *EXISTS* predicate is that it allows you to intuitively phrase English-like queries. For example, this query can be read just as you would say it in ordinary English: select the customer ID and company name attributes from the *Customers* table, where the country is equal to Spain, and at least one order exists in the *Orders* table with the same customer ID as the customer's customer ID.

This query returns the following output.

```
custid      companyname
----------- ----------------
8           Customer QUHWH
29          Customer MDLWA
30          Customer KSLQF
69          Customer SIUIH
```

As with other predicates, you can negate the *EXISTS* predicate with the *NOT* logical operator. For example, the following query returns customers from Spain who did not place orders.

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE country = N'Spain'
  AND NOT EXISTS
    (SELECT * FROM Sales.Orders AS O
     WHERE O.custid = C.custid);
```

This query returns the following output.

```
custid      companyname
----------- ----------------
22          Customer DTDMN
```

Even though this book's focus is on logical query processing and not performance, I thought you might be interested to know that the *EXISTS* predicate lends itself to good optimization. That is, the Microsoft SQL Server engine knows that it is enough to determine whether the subquery returns at least one row or none, and it doesn't need to process all qualifying rows. You can think of this capability as a kind of short-circuit evaluation.

Unlike most other cases, in this case it's logically not a bad practice to use an asterisk (*) in the *SELECT* list of the subquery in the context of the *EXISTS* predicate. The *EXISTS* predicate only cares about the existence of matching rows regardless of the attributes specified in the *SELECT* list, as if the whole *SELECT* clause were superfluous. The SQL Server database engine knows this, and in terms of optimization, ignores the subquery's *SELECT* list. So in terms of optimization, specifying the column wildcard * in this case has no negative impact when compared to alternatives such as specifying a constant. However, some minor extra cost might be involved in the resolution process of expanding the wildcard against metadata info. But this extra resolution cost is so minor that you will probably barely notice it. My opinion on this matter is that queries should be natural and intuitive, unless there's a very compelling reason to sacrifice this aspect of the code. I find the form *EXISTS(SELECT * FROM . . .)* much more intuitive than *EXISTS(SELECT 1 FROM . . .)*. Saving the minor extra cost associated with the resolution of * is something that is not worth the cost of sacrificing the readability of the code.

Finally, another aspect of the *EXISTS* predicate that is interesting to note is that unlike most predicates in T-SQL, *EXISTS* uses two-valued logic and not three-valued logic. If you think about it, there's no situation where it is unknown whether a query returns rows.

Beyond the Fundamentals of Subqueries

This section covers aspects of subqueries that you might consider to be beyond the fundamentals. I provide it as optional reading in case you feel very comfortable with the material covered so far in this chapter.

Returning Previous or Next Values

Suppose that you need to query the *Orders* table in the *TSQL2012* database and return, for each order, information about the current order and also the previous order ID. The concept of "previous" implies logical ordering, but because you know that the rows in a table have no order, you need to come up with a logical equivalent to the concept of "previous" that can be phrased with a T-SQL expression. One example of such a logical equivalent is "the maximum value that is smaller than the current value." This phrase can be expressed in T-SQL with a correlated subquery like this.

```
SELECT orderid, orderdate, empid, custid,
  (SELECT MAX(O2.orderid)
   FROM Sales.Orders AS O2
   WHERE O2.orderid < O1.orderid) AS prevorderid
FROM Sales.Orders AS O1;
```

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

```
orderid     orderdate                   empid       custid      prevorderid
----------- --------------------------- ----------- ----------- -----------
10248       2006-07-04 00:00:00.000     5           85          NULL
10249       2006-07-05 00:00:00.000     6           79          10248
10250       2006-07-08 00:00:00.000     4           34          10249
10251       2006-07-08 00:00:00.000     3           84          10250
10252       2006-07-09 00:00:00.000     4           76          10251
...
11073       2008-05-05 00:00:00.000     2           58          11072
11074       2008-05-06 00:00:00.000     7           73          11073
11075       2008-05-06 00:00:00.000     8           68          11074
11076       2008-05-06 00:00:00.000     4           9           11075
11077       2008-05-06 00:00:00.000     1           65          11076

(830 row(s) affected)
```

Notice that because there's no order before the first, the subquery returned a *NULL* for the first order.

Similarly, you can phrase the concept of "next" as "the minimum value that is greater than the current value." Here's the T-SQL query that returns for each order the next order ID.

```
SELECT orderid, orderdate, empid, custid,
  (SELECT MIN(O2.orderid)
   FROM Sales.Orders AS O2
   WHERE O2.orderid > O1.orderid) AS nextorderid
FROM Sales.Orders AS O1;
```

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

```
orderid     orderdate                   empid       custid      nextorderid
----------- --------------------------- ----------- ----------- -----------
10248       2006-07-04 00:00:00.000     5           85          10249
10249       2006-07-05 00:00:00.000     6           79          10250
10250       2006-07-08 00:00:00.000     4           34          10251
10251       2006-07-08 00:00:00.000     3           84          10252
10252       2006-07-09 00:00:00.000     4           76          10253
...
11073       2008-05-05 00:00:00.000     2           58          11074
11074       2008-05-06 00:00:00.000     7           73          11075
11075       2008-05-06 00:00:00.000     8           68          11076
11076       2008-05-06 00:00:00.000     4           9           11077
11077       2008-05-06 00:00:00.000     1           65          NULL

(830 row(s) affected)
```

Notice that because there's no order after the last, the subquery returned a *NULL* for the last order.

Note that SQL Server 2012 introduces new window functions called *LAG* and *LEAD* that allow the return of an element from a "previous" or "next" row based on specified ordering. I will cover these and other window functions in Chapter 7.

Using Running Aggregates

Running aggregates are aggregates that accumulate values over time. In this section, I use the *Sales.OrderTotalsByYear* view to demonstrate the technique for calculating running aggregates. The view shows total order quantities by year. Query the view to examine its contents.

```
SELECT orderyear, qty
FROM Sales.OrderTotalsByYear;
```

You get the following output.

```
orderyear   qty
----------- -----------
2007        25489
2008        16247
2006        9581
```

Suppose you need to return for each year the order year, quantity, and running total quantity over the years. That is, for each year, return the sum of the quantity up to that year. So for the earliest year recorded in the view (2006), the running total is equal to that year's quantity. For the second year (2007), the running total is the sum of the first year plus the second year, and so on.

You can complete this task by querying one instance of the view (call it *O1*) to return for each year the order year and quantity, and then by using a correlated subquery against a second instance of the view (call it *O2*) to calculate the running-total quantity. The subquery should filter all years in *O2* that are smaller than or equal to the current year in *O1*, and sum the quantities from *O2*. Here's the solution query.

```
SELECT orderyear, qty,
  (SELECT SUM(O2.qty)
   FROM Sales.OrderTotalsByYear AS O2
   WHERE O2.orderyear <= O1.orderyear) AS runqty
FROM Sales.OrderTotalsByYear AS O1
ORDER BY orderyear;
```

This query returns the following output.

```
orderyear   qty         runqty
----------- ----------- -----------
2006        9581        9581
2007        25489       35070
2008        16247       51317
```

Note that SQL Server 2012 enhances the capabilities of window aggregate functions, allowing new, highly efficient solutions for running totals needs. As mentioned, I will discuss window functions in Chapter 7.

Dealing with Misbehaving Subqueries

This section introduces cases in which subqueries might behave counter to your expectations, and provides best practices that you can follow to avoid logical bugs in your code that are associated with those cases.

NULL Trouble

Remember that T-SQL uses three-valued logic. In this section, I will demonstrate problems that can evolve with subqueries when *NULL* marks are involved and you do not take into consideration the three-valued logic.

Consider the following apparently intuitive query that is supposed to return customers who did not place orders.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN(SELECT O.custid
                    FROM Sales.Orders AS O);
```

With the current sample data in the *Orders* table in the *TSQL2012* database, the query seems to work the way you expect it to; and indeed, it returns two rows for the two customers who did not place orders.

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

Next, run the following code to insert a new order to the *Orders* table with a *NULL* customer ID.

```
INSERT INTO Sales.Orders
  (custid, empid, orderdate, requireddate, shippeddate, shipperid,
   freight, shipname, shipaddress, shipcity, shipregion,
   shippostalcode, shipcountry)
  VALUES(NULL, 1, '20090212', '20090212',
         '20090212', 1, 123.00, N'abc', N'abc', N'abc',
         N'abc', N'abc', N'abc');
```

Run the query that is supposed to return customers who did not place orders again.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN(SELECT O.custid
                    FROM Sales.Orders AS O);
```

This time, the query returns an empty set. Keeping in mind what you've read in the section about *NULL* marks in Chapter 2, "Single-Table Queries," try to explain why the query returns an empty set. Also try to think of ways to get customers 22 and 57 in the output, and in general, to figure out best practices you can follow to avoid such problems, assuming that there is a problem here.

Obviously, the culprit in this story is the *NULL* customer ID that was added to the *Orders* table and is now returned among the known customer IDs by the subquery.

Let's start with the part that behaves the way you expect it to. The *IN* predicate returns *TRUE* for a customer who placed orders (for example, customer 85), because such a customer is returned by the subquery. The *NOT* operator is used to negate the *IN* predicate; hence, the *NOT TRUE* becomes *FALSE*, and the customer is not returned by the outer query. This means that when a customer ID appears in the *Orders* table, you can tell for sure that the customer placed orders, and therefore you don't want to see it in the output. However, when you have a *NULL* customer ID in the *Orders* table, you can't tell for sure whether a certain customer ID does not appear in *Orders*, as explained shortly.

The *IN* predicate returns *UNKNOWN* (the truth value *UNKNOWN* like the truth values *TRUE* and *FALSE*) for a customer such as 22 that does not appear in the set of known customer IDs in *Orders*. The *IN* predicate returns *UNKNOWN* for such a customer, because comparing it with all known customer IDs yields *FALSE*, and comparing it with the *NULL* in the set yields *UNKNOWN*. *FALSE OR UNKNOWN* yields *UNKNOWN*. As a more tangible example, consider the expression *22 NOT IN (1, 2, NULL)*. This expression can be rephrased as *NOT 22 IN (1, 2, NULL*). You can expand the last expression to *NOT (22 = 1 OR 22 = 2 OR 22 = NULL).* Evaluate each individual expression in the parentheses to its truth value and you get *NOT (FALSE OR FALSE OR UNKNOWN*), which translates to *NOT UNKNOWN*, which evaluates to *UNKNOWN*.

The logical meaning of *UNKNOWN* here before you apply the *NOT* operator is that it can't be determined whether the customer ID appears in the set, because the *NULL* could represent that customer ID as well as anything else. The tricky part is that negating the *UNKNOWN* with the *NOT* operator still yields *UNKNOWN*, and *UNKNOWN* in a query filter is filtered out. This means that in a case where it is unknown whether a customer ID appears in a set, it is also unknown whether it doesn't appear in the set.

In short, when you use the *NOT IN* predicate against a subquery that returns at least one *NULL*, the outer query always returns an empty set. Values from the outer table that are known to appear in the set are not returned because the outer query is supposed to return values that do not appear in the set. Values that do not appear in the set of known values are not returned because you can never tell for sure that the value is not in the set that includes the *NULL*.

So, what practices can you follow to avoid such trouble?

First, when a column is not supposed to allow *NULL* marks, it is important to define it as *NOT NULL*. Enforcing data integrity is much more important than many people realize.

Second, in all queries that you write, you should consider all three possible truth values of three-valued logic (*TRUE*, *FALSE*, and *UNKNOWN*). Think explicitly about whether the query might process *NULL* marks, and if so, whether the default treatment of *NULL* marks is suitable for your needs. When it isn't, you need to intervene. For example, in the example we've been working with, the outer query returns an empty set because of the comparison with *NULL*. If you want to check whether a customer ID appears in the set of known values and ignore the *NULL* marks, you should exclude the *NULL* marks—either explicitly or implicitly. One way to explicitly exclude the *NULL* marks is to add the predicate *O.custid IS NOT NULL* to the subquery, like this.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN(SELECT O.custid
                    FROM Sales.Orders AS O
                    WHERE O.custid IS NOT NULL);
```

You can also exclude the *NULL* marks implicitly by using the *NOT EXISTS* predicate instead of *NOT IN*, like this.

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE NOT EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid);
```

Recall that unlike *IN*, *EXISTS* uses two-valued predicate logic. *EXISTS* always returns *TRUE* or *FALSE* and never *UNKNOWN*. When the subquery stumbles into a *NULL* in *O.custid*, the expression evaluates to *UNKNOWN* and the row is filtered out. As far as the *EXISTS* predicate is concerned, the *NULL* cases are eliminated naturally, as though they weren't there. So *EXISTS* ends up handling only known customer IDs. Therefore, it's safer to use *NOT EXISTS* than *NOT IN*.

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

```
DELETE FROM Sales.Orders WHERE custid IS NULL;
```

Substitution Errors in Subquery Column Names

Logical bugs in your code can sometimes be very elusive. In this section, I describe an elusive bug that has to do with an innocent substitution error in a subquery column name. After explaining the bug, I provide best practices that can help you avoid such bugs in the future.

The examples in this section query a table called *MyShippers* in the *Sales* schema. Run the following code to create and populate this table.

```
IF OBJECT_ID('Sales.MyShippers', 'U') IS NOT NULL
  DROP TABLE Sales.MyShippers;

CREATE TABLE Sales.MyShippers
(
  shipper_id  INT          NOT NULL,
  companyname NVARCHAR(40) NOT NULL,
  phone       NVARCHAR(24) NOT NULL,
  CONSTRAINT PK_MyShippers PRIMARY KEY(shipper_id)
);

INSERT INTO Sales.MyShippers(shipper_id, companyname, phone)
  VALUES(1, N'Shipper GVSUA', N'(503) 555-0137'),
        (2, N'Shipper ETYNR', N'(425) 555-0136'),
        (3, N'Shipper ZHISN', N'(415) 555-0138');
```

Consider the following query, which is supposed to return shippers who shipped orders to customer 43.

```
SELECT shipper_id, companyname
FROM Sales.MyShippers
WHERE shipper_id IN
  (SELECT shipper_id
   FROM Sales.Orders
   WHERE custid = 43);
```

This query produces the following output.

```
shipper_id  companyname
----------- ---------------
1           Shipper GVSUA
2           Shipper ETYNR
3           Shipper ZHISN
```

Apparently, only shippers 2 and 3 shipped orders to customer 43, but for some reason, this query returned all shippers from the *MyShippers* table. Examine the query carefully and also the schemas of the tables involved, and see if you can explain why.

It turns out that the column name in the *Orders* table holding the shipper ID is not called *shipper_id*; it is called *shipperid* (no underscore). The column in the *MyShippers* table is called *shipper_id* with an underscore. The resolution of nonprefixed column names works in the context of a subquery from the current/inner scope outward. In our example, SQL Server first looks for the column *shipper_id* in the *Orders* table. Such a column is not found there, so SQL Server looks for it in the outer table in the query, *MyShippers*. Because one is found, it is the one used.

You can see that what was supposed to be a self-contained subquery unintentionally became a correlated subquery. As long as the *Orders* table has at least one row, all rows from the *MyShippers* table find a match when comparing the outer shipper ID with a query that returns the very same outer shipper ID for each row from the *Orders* table.

Some might argue that this behavior is a design flaw in standard SQL. However, it's not that the designers of this behavior in the ANSI SQL committee thought that it would be difficult to detect the "error;" rather, it's an intentional behavior designed to allow you to refer to column names from the outer table without needing to prefix them with the table name, as long as those column names are unambiguous (that is, as long as they appear only in one of the tables).

This problem is more common in environments that do not use consistent attribute names across tables. Sometimes the names are only slightly different, as in this case—*shipperid* in one table and *shipper_id* in another. That's enough for the bug to manifest itself.

You can follow a couple of best practices to avoid such problems—one to implement in the long run, and one that you can implement in the short run.

In the long run, your organization should as a policy not underestimate the importance of using consistent attribute names across tables. In the short run, of course, you don't want to start changing existing column names, which could break application code.

In the short run, you can adopt a very simple practice—prefix column names in subqueries with the source table alias. This way, the resolution process only looks for the column in the specified table, and if no such column is there, you get a resolution error. For example, try running the following code.

```
SELECT shipper_id, companyname
FROM Sales.MyShippers
WHERE shipper_id IN
  (SELECT O.shipper_id
   FROM Sales.Orders AS O
   WHERE O.custid = 43);
```

You get the following resolution error.

```
Msg 207, Level 16, State 1, Line 4
Invalid column name 'shipper_id'.
```

After getting this error, you of course can identify the problem and correct the query.

```
SELECT shipper_id, companyname
FROM Sales.MyShippers
WHERE shipper_id IN
  (SELECT O.shipperid
   FROM Sales.Orders AS O
   WHERE O.custid = 43);
```

This time, the query returns the expected result.

```
shipper_id  companyname
----------- ---------------
2           Shipper ETYNR
3           Shipper ZHISN
```

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

```
IF OBJECT_ID('Sales.MyShippers', 'U') IS NOT NULL
  DROP TABLE Sales.MyShippers;
```

Conclusion

This chapter covered subqueries. It discussed self-contained subqueries, which are independent of their outer queries, and correlated subqueries, which are dependent on their outer queries. Regarding the results of subqueries, I discussed scalar and multivalued subqueries. I also provided a more advanced section as optional reading, in which I covered returning previous and next values, using running aggregates, and dealing with misbehaving subqueries. Remember to always think about the three-valued logic and the importance of prefixing column names in subqueries with the source table alias.

The next chapter focuses on table subqueries, also known as table expressions.

Exercises

This section provides exercises to help you familiarize yourself with the subjects discussed in this chapter. The sample database *TSQL2012* is used in all exercises in this chapter.

1

Write a query that returns all orders placed on the last day of activity that can be found in the *Orders* table.

- Tables involved: *Sales.Orders*
- Desired output:

```
orderid     orderdate                   custid      empid
----------- --------------------------- ----------- -----------
11077       2008-05-06 00:00:00.000     65          1
11076       2008-05-06 00:00:00.000     9           4
11075       2008-05-06 00:00:00.000     68          8
11074       2008-05-06 00:00:00.000     73          7
```

2 (Optional, Advanced)

Write a query that returns all orders placed by the customer(s) who placed the highest number of orders. Note that more than one customer might have the same number of orders.

- Tables involved: *Sales.Orders*
- Desired output (abbreviated):

```
custid      orderid     orderdate                   empid
----------- ----------- --------------------------- -----------
71          10324       2006-10-08 00:00:00.000     9
71          10393       2006-12-25 00:00:00.000     1
71          10398       2006-12-30 00:00:00.000     2
71          10440       2007-02-10 00:00:00.000     4
71          10452       2007-02-20 00:00:00.000     8
71          10510       2007-04-18 00:00:00.000     6
71          10555       2007-06-02 00:00:00.000     6
71          10603       2007-07-18 00:00:00.000     8
71          10607       2007-07-22 00:00:00.000     5
71          10612       2007-07-28 00:00:00.000     1
71          10627       2007-08-11 00:00:00.000     8
71          10657       2007-09-04 00:00:00.000     2
71          10678       2007-09-23 00:00:00.000     7
71          10700       2007-10-10 00:00:00.000     3
71          10711       2007-10-21 00:00:00.000     5
71          10713       2007-10-22 00:00:00.000     1
71          10714       2007-10-22 00:00:00.000     5
71          10722       2007-10-29 00:00:00.000     8
71          10748       2007-11-20 00:00:00.000     3
71          10757       2007-11-27 00:00:00.000     6
71          10815       2008-01-05 00:00:00.000     2
71          10847       2008-01-22 00:00:00.000     4
71          10882       2008-02-11 00:00:00.000     4
71          10894       2008-02-18 00:00:00.000     1
71          10941       2008-03-11 00:00:00.000     7
71          10983       2008-03-27 00:00:00.000     2
71          10984       2008-03-30 00:00:00.000     1
71          11002       2008-04-06 00:00:00.000     4
71          11030       2008-04-17 00:00:00.000     7
71          11031       2008-04-17 00:00:00.000     6
71          11064       2008-05-01 00:00:00.000     1

(31 row(s) affected)
```

3

Write a query that returns employees who did not place orders on or after May 1, 2008.

- Tables involved: *HR.Employees* and *Sales.Orders*
- Desired output:

```
empid       FirstName     lastname
----------- ------------- --------------------
3           Judy          Lew
5           Sven          Buck
6           Paul          Suurs
9           Zoya          Dolgopyatova
```

4

Write a query that returns countries where there are customers but not employees.

- Tables involved: *Sales.Customers* and *HR.Employees*
- Desired output:

```
country
---------------
Argentina
Austria
Belgium
Brazil
Canada
Denmark
Finland
France
Germany
Ireland
Italy
Mexico
Norway
Poland
Portugal
Spain
Sweden
Switzerland
Venezuela

(19 row(s) affected)
```

5

Write a query that returns for each customer all orders placed on the customer's last day of activity.

- Tables involved: *Sales.Orders*
- Desired output:

```
custid      orderid     orderdate               empid
----------- ----------- ----------------------- -----------
1           11011       2008-04-09 00:00:00.000 3
2           10926       2008-03-04 00:00:00.000 4
3           10856       2008-01-28 00:00:00.000 3
4           11016       2008-04-10 00:00:00.000 9
5           10924       2008-03-04 00:00:00.000 3
...
87          11025       2008-04-15 00:00:00.000 6
88          10935       2008-03-09 00:00:00.000 4
89          11066       2008-05-01 00:00:00.000 7
90          11005       2008-04-07 00:00:00.000 2
91          11044       2008-04-23 00:00:00.000 4

(90 row(s) affected)
```

6

Write a query that returns customers who placed orders in 2007 but not in 2008.

- Tables involved: *Sales.Customers* and *Sales.Orders*
- Desired output:

```
custid      companyname
----------- ----------------
21          Customer KIDPX
23          Customer WVFAF
33          Customer FVXPQ
36          Customer LVJSO
43          Customer UISOJ
51          Customer PVDZC
85          Customer ENQZT

(7 row(s) affected)
```

7 (Optional, Advanced)

Write a query that returns customers who ordered product 12.

- Tables involved: *Sales.Customers*, *Sales.Orders*, and *Sales.OrderDetails*
- Desired output:

```
custid      companyname
----------- ----------------
48          Customer DVFMB
39          Customer GLLAG
71          Customer LCOUJ
65          Customer NYUHS
44          Customer OXFRU
51          Customer PVDZC
86          Customer SNXOJ
20          Customer THHDP
90          Customer XBBVR
46          Customer XPNIK
31          Customer YJCBX
87          Customer ZHYOS

(12 row(s) affected)
```

8 (Optional, Advanced)

Write a query that calculates a running-total quantity for each customer and month.

- Tables involved: *Sales.CustOrders*
- Desired output:

```
custid      ordermonth                  qty         runqty
----------- --------------------------- ----------- -----------
1           2007-08-01 00:00:00.000     38          38
1           2007-10-01 00:00:00.000     41          79
1           2008-01-01 00:00:00.000     17          96
1           2008-03-01 00:00:00.000     18          114
1           2008-04-01 00:00:00.000     60          174
2           2006-09-01 00:00:00.000     6           6
2           2007-08-01 00:00:00.000     18          24
2           2007-11-01 00:00:00.000     10          34
2           2008-03-01 00:00:00.000     29          63
3           2006-11-01 00:00:00.000     24          24
3           2007-04-01 00:00:00.000     30          54
3           2007-05-01 00:00:00.000     80          134
3           2007-06-01 00:00:00.000     83          217
3           2007-09-01 00:00:00.000     102         319
3           2008-01-01 00:00:00.000     40          359
...

(636 row(s) affected)
```

Solutions

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

1

You can write a self-contained subquery that returns the maximum order date from the *Orders* table. You can refer to the subquery in the *WHERE* clause of the outer query to return all orders that were placed on the last day of activity. Here's the solution query.

```
USE TSQL2012;

SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate =
  (SELECT MAX(O.orderdate) FROM Sales.Orders AS O);
```

2

This problem is best solved in multiple steps. First, you can write a query that returns the customer or customers who placed the highest number of orders. You can achieve this by grouping the orders by customer, ordering the customers by *COUNT(*)* descending, and using the *TOP(1) WITH TIES* option to return the IDs of the customers who placed the highest number of orders. If you don't remember how to use the *TOP* option, refer to Chapter 2. Here's the query that solves the first step.

```
SELECT TOP (1) WITH TIES O.custid
FROM Sales.Orders AS O
GROUP BY O.custid
ORDER BY COUNT(*) DESC;
```

This query returns the value 71, which is the customer ID of the customer who placed the highest number of orders, 31. With the sample data stored in the *Orders* table, only one customer placed the maximum number of orders. But the query uses the *WITH TIES* option to return all IDs of customers who placed the maximum number of orders, in case there are more than one.

The next step is to write a query against the *Orders* table returning all orders where the customer ID is in the set of customer IDs returned by the solution query for the first step.

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders
WHERE custid IN
  (SELECT TOP (1) WITH TIES O.custid
   FROM Sales.Orders AS O
   GROUP BY O.custid
   ORDER BY COUNT(*) DESC);
```

3

You can write a self-contained subquery against the *Orders* table that filters orders placed on or after May 1, 2008 and returns only the employee IDs from those orders. Write an outer query against the *Employees* table returning employees whose IDs do not appear in the set of employee IDs returned by the subquery. Here's the complete solution query.

```
SELECT empid, FirstName, lastname
FROM HR.Employees
WHERE empid NOT IN
  (SELECT O.empid
   FROM Sales.Orders AS O
   WHERE O.orderdate >= '20080501');
```

4

You can write a self-contained subquery against the *Employees* table returning the country attribute from each employee row. Write an outer query against the *Customers* table that filters only customer rows where the country does not appear in the set of countries returned by the subquery. In the *SELECT* list of the outer query, specify *DISTINCT country* to return only distinct occurrences of countries, because the same country can have more than one customer. Here's the complete solution query.

```
SELECT DISTINCT country
FROM Sales.Customers
WHERE country NOT IN
  (SELECT E.country FROM HR.Employees AS E);
```

5

This exercise is similar to Exercise 1, except that in that exercise, you were asked to return orders placed on the last day of activity in general; in this exercise, you were asked to return orders placed on the last day of activity for the customer. The solutions for both exercises are similar, but here you need to correlate the subquery to match the inner customer ID with the outer customer ID, like this.

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders AS O1
WHERE orderdate =
  (SELECT MAX(O2.orderdate)
   FROM Sales.Orders AS O2
   WHERE O2.custid = O1.custid)
ORDER BY custid;
```

You're not comparing the outer row's order date with the general maximum order date, but instead with the maximum order date for the current customer.

6

You can solve this problem by querying the *Customers* table and using *EXISTS* and *NOT EXISTS* predicates with correlated subqueries to ensure that the customer placed orders in 2007 but not in 2008. The *EXISTS* predicate returns *TRUE* only if at least one row exists in the *Orders* table with the same customer ID as in the outer row, within the date range representing the year 2007. The *NOT EXISTS* predicate returns *TRUE* only if no row exists in the *Orders* table with the same customer ID as in the outer row, within the date range representing the year 2008. Here's the complete solution query.

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
     AND O.orderdate >= '20070101'
     AND O.orderdate < '20080101')
  AND NOT EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
     AND O.orderdate >= '20080101'
     AND O.orderdate < '20090101');
```

7

You can solve this exercise by nesting *EXISTS* predicates with correlated subqueries. You write the outermost query against the *Customers* table. In the *WHERE* clause of the outer query, you can use the *EXISTS* predicate with a correlated subquery against the *Orders* table to filter only the current customer's orders. In the filter of the subquery against the *Orders* table, you can use a nested *EXISTS* predicate with a subquery against the *OrderDetails* table that filters only order details with product ID 12. This way, only customers who placed orders that contain product 12 in their order details are returned. Here's the complete solution query.

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
     AND EXISTS
       (SELECT *
        FROM Sales.OrderDetails AS OD
        WHERE OD.orderid = O.orderid
          AND OD.ProductID = 12));
```

8

When I need to solve querying problems, I often find it useful to rephrase the original request in a more technical way so that it will be more convenient to translate the request to a T-SQL query. To solve the current exercise, you can first try to express the request "return a running total quantity for each customer and month" differently—in a more technical manner. For each customer, return the customer ID, month, the sum of the quantity for that month, and the sum of all months less than or equal to the current month. The rephrased request can be translated to the following T-SQL query quite literally.

```
SELECT custid, ordermonth, qty,
  (SELECT SUM(O2.qty)
   FROM Sales.CustOrders AS O2
   WHERE O2.custid = O1.custid
     AND O2.ordermonth <= O1.ordermonth) AS runqty
FROM Sales.CustOrders AS O1
ORDER BY custid, ordermonth;
```

This page intentionally left blank

Index

About the Author

ITZIK BEN-GAN is a mentor with and co-founder of SolidQ. A SQL Server Microsoft MVP since 1999, Itzik has taught 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 Pro* as well as articles and white papers for MSDN and *The SolidQ Journal*. Itzik's speaking engagements include Tech-Ed, SQL PASS, SQL Server Connections, presentations to various SQL Server user groups, and SolidQ events. Itzik is a subject-matter expert within SolidQ for its T-SQL related activities. He authored SolidQ's Advanced T-SQL and T-SQL Fundamentals courses and delivers them regularly worldwide.