



MURACH

TRAINING & REFERENCE

murach's SQL Server 2019 for developers

BRYAN SYVERSON

JOEL MURACH

MASTER THE SQL STATEMENTS

that every application developer needs
to retrieve and update the data
in a Microsoft SQL Server database

DESIGN DATABASES LIKE A DBA

and implement them with SQL
statements or the Management Studio

GAIN PROFESSIONAL SKILLS

like working with views, scripts,
stored procedures, functions,
triggers, transactions, locking,
security, XML, and BLOBs

murach's
SQL Server
2019
for developers

Bryan Syverson
Joel Murach

TRAINING & REFERENCE

**murach's
SQL Server
2019
for developers**

**Bryan Syverson
Joel Murach**



MIKE MURACH & ASSOCIATES, INC.
4340 N. Knoll Ave. • Fresno, CA 93722
www.murach.com • murachbooks@murach.com

Authors: Bryan Syverson
Joel Murach

Editor: Anne Boehm

Production: Juliette Baylon

Books for database developers

Murach's SQL Server for Developers

Murach's Oracle SQL and PL/SQL for Developers

Murach's MySQL

Books for .NET developers

Murach's C#

Murach's Visual Basic

Murach's ASP.NET Web Programming with C#

Murach's ASP.NET Core MVC

Books for Python, C++, and Java developers

Murach's Python Programming

Murach's C++ Programming

Murach's Java Programming

Murach's Java Servlets and JSP

Books for web developers

Murach's HTML5 and CSS3

Murach's JavaScript and jQuery

Murach's PHP and MySQL

**For more on Murach books,
please visit us at www.murach.com**

© 2020, Mike Murach & Associates, Inc.
All rights reserved.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1
ISBN-13: 978-1-943872-57-2

Content

Introduction

xvii

Section 1 An introduction to SQL

Chapter 1	An introduction to relational databases and SQL	3
Chapter 2	How to use the Management Studio	49

Section 2 The essential SQL skills

Chapter 3	How to retrieve data from a single table	85
Chapter 4	How to retrieve data from two or more tables	125
Chapter 5	How to code summary queries	159
Chapter 6	How to code subqueries	183
Chapter 7	How to insert, update, and delete data	215
Chapter 8	How to work with data types	239
Chapter 9	How to use functions	261

Section 3 Database design and implementation

Chapter 10	How to design a database	303
Chapter 11	How to create a database and its tables with SQL statements	333
Chapter 12	How to create a database and its tables with the Management Studio	375

Section 4 Advanced SQL skills

Chapter 13	How to work with views	395
Chapter 14	How to code scripts	417
Chapter 15	How to code stored procedures, functions, and triggers	457
Chapter 16	How to manage transactions and locking	509
Chapter 17	How to manage database security	535
Chapter 18	How to work with XML	587
Chapter 19	How to work with BLOBs	619

Reference aids

Appendix A	How to set up your computer for this book	647
Index		659

Expanded contents

Section 1 An introduction to SQL

Chapter 1 An introduction to relational databases and SQL

An introduction to client/server systems.....	4
The hardware components of a client/server system	4
The software components of a client/server system.....	6
Other client/server system architectures	8
An introduction to the relational database model.....	10
How a database table is organized	10
How the tables in a relational database are related	12
How the columns in a table are defined	14
How relational databases compare to other data models	16
An introduction to SQL and SQL-based systems	18
A brief history of SQL	18
A comparison of Oracle, DB2, MySQL, and SQL Server.....	20
The Transact-SQL statements.....	22
An introduction to the SQL statements.....	22
Typical statements for working with database objects.....	24
How to query a single table.....	26
How to join data from two or more tables	28
How to add, update, and delete data in a table	30
SQL coding guidelines.....	32
How to work with other database objects.....	34
How to work with views	34
How to work with stored procedures, triggers, and user-defined functions.....	36
How to use SQL from an application program	38
Common data access models	38
How to use ADO.NET from a .NET application	40
Visual Basic code that retrieves data from a SQL Server database.....	42
C# code that retrieves data from a SQL Server database	44

Chapter 2 How to use the Management Studio

An introduction to SQL Server 2019	50
A summary of the SQL Server 2019 tools	50
How to start and stop the database engine	52
How to enable remote connections	52
An introduction to the Management Studio.....	54
How to connect to a database server	54
How to navigate through the database objects	56
How to manage the database files	58
How to attach a database	58
How to detach a database	58
How to back up a database	60
How to restore a database.....	60
How to set the compatibility level for a database	62

How to view and modify the database.....	64
How to create database diagrams.....	64
How to view the column definitions of a table.....	66
How to modify the column definitions.....	66
How to view the data of a table.....	68
How to modify the data of a table.....	68
How to work with queries.....	70
How to enter and execute a query	70
How to handle syntax errors.....	72
How to open and save queries.....	74
An introduction to the Query Designer.....	76
How to view the documentation for SQL Server	78
How to display the SQL Server documentation.....	78
How to look up information in the documentation	78

Section 2 The essential SQL skills

Chapter 3 How to retrieve data from a single table

An introduction to the SELECT statement.....	86
The basic syntax of the SELECT statement.....	86
SELECT statement examples.....	88
How to code the SELECT clause.....	90
How to code column specifications.....	90
How to name the columns in a result set.....	92
How to code string expressions.....	94
How to code arithmetic expressions.....	96
How to use functions.....	98
How to use the DISTINCT keyword to eliminate duplicate rows	100
How to use the TOP clause to return a subset of selected rows	102
How to code the WHERE clause	104
How to use comparison operators	104
How to use the AND, OR, and NOT logical operators	106
How to use the IN operator	108
How to use the BETWEEN operator	110
How to use the LIKE operator	112
How to use the IS NULL clause.....	114
How to code the ORDER BY clause.....	116
How to sort a result set by a column name	116
How to sort a result set by an alias, an expression, or a column number.....	118
How to retrieve a range of selected rows	120

Chapter 4

How to retrieve data from two or more tables

How to work with inner joins	126
How to code an inner join	126
When and how to use correlation names.....	128
How to work with tables from different databases.....	130
How to use compound join conditions.....	132
How to use a self-join.....	134
Inner joins that join more than two tables	136
How to use the implicit inner join syntax	138

How to work with outer joins	140
How to code an outer join	140
Outer join examples	142
Outer joins that join more than two tables	144
Other skills for working with joins	146
How to combine inner and outer joins	146
How to use cross joins.....	148
How to work with unions.....	150
The syntax of a union.....	150
Unions that combine data from different tables.....	150
Unions that combine data from the same table	152
How to use the EXCEPT and INTERSECT operators	154
Chapter 5 How to code summary queries	
How to work with aggregate functions.....	160
How to code aggregate functions	160
Queries that use aggregate functions.....	162
How to group and summarize data	164
How to code the GROUP BY and HAVING clauses	164
Queries that use the GROUP BY and HAVING clauses.....	166
How the HAVING clause compares to the WHERE clause.....	168
How to code complex search conditions.....	170
How to summarize data using SQL Server extensions	172
How to use the ROLLUP operator	172
How to use the CUBE operator.....	174
How to use the GROUPING SETS operator.....	176
How to use the OVER clause	178
Chapter 6 How to code subqueries	
An introduction to subqueries	184
How to use subqueries.....	184
How subqueries compare to joins	186
How to code subqueries in search conditions	188
How to use subqueries with the IN operator	188
How to compare the result of a subquery with an expression.....	190
How to use the ALL keyword	192
How to use the ANY and SOME keywords	194
How to code correlated subqueries	196
How to use the EXISTS operator	198
Other ways to use subqueries.....	200
How to code subqueries in the FROM clause	200
How to code subqueries in the SELECT clause.....	202
Guidelines for working with complex queries	204
A complex query that uses subqueries	204
A procedure for building complex queries.....	206
How to work with common table expressions.....	208
How to code a CTE	208
How to code a recursive CTE.....	210

Chapter 7 How to insert, update, and delete data

How to create test tables	216
How to use the SELECT INTO statement	216
How to use a copy of the database	216
How to insert new rows.....	218
How to insert a single row.....	218
How to insert multiple rows	218
How to insert default values and null values.....	220
How to insert rows selected from another table.....	222
How to modify existing rows	224
How to perform a basic update operation	224
How to use subqueries in an update operation.....	226
How to use joins in an update operation	228
How to delete existing rows.....	230
How to perform a basic delete operation	230
How to use subqueries and joins in a delete operation	232
How to merge rows.....	234
How to perform a basic merge operation	234
How to code more complex merge operations.....	234

Chapter 8 How to work with data types

A review of the SQL data types	240
Data type overview.....	240
The numeric data types	242
The string data types	244
The date/time data types.....	246
The large value data types	248
How to convert data.....	250
How data conversion works	250
How to convert data using the CAST function	252
How to convert data using the CONVERT function	254
How to use the TRY_CONVERT function	256
How to use other data conversion functions.....	258

Chapter 9 How to use functions

How to work with string data	262
A summary of the string functions.....	262
How to solve common problems that occur with string data.....	266
How to work with numeric data.....	268
A summary of the numeric functions.....	268
How to solve common problems that occur with numeric data.....	270
How to work with date/time data.....	272
A summary of the date/time functions	272
How to parse dates and times.....	276
How to perform operations on dates and times.....	278
How to perform a date search	280
How to perform a time search	282
Other functions you should know about.....	284
How to use the CASE function	284
How to use the IIF and CHOOSE functions	286

How to use the COALESCE and ISNULL functions	288
How to use the GROUPING function.....	290
How to use the ranking functions.....	292
How to use the analytic functions	296

Section 3 Database design and implementation

Chapter 10 How to design a database

How to design a data structure	304
The basic steps for designing a data structure.....	304
How to identify the data elements.....	306
How to subdivide the data elements.....	308
How to identify the tables and assign columns	310
How to identify the primary and foreign keys	312
How to enforce the relationships between tables.....	314
How normalization works	316
How to identify the columns to be indexed.....	318
How to normalize a data structure	320
The seven normal forms.....	320
How to apply the first normal form.....	322
How to apply the second normal form.....	324
How to apply the third normal form.....	326
When and how to denormalize a data structure	328

Chapter 11 How to create a database and its tables with SQL Statements

An introduction to DDL	334
The SQL statements for data definition	334
Rules for coding object names	336
How to create databases, tables, and indexes	338
How to create a database	338
How to create a table	340
How to create an index.....	342
How to use snippets to create database objects.....	344
How to use constraints	346
An introduction to constraints.....	346
How to use check constraints	348
How to use foreign key constraints	350
How to change databases and tables.....	352
How to delete an index, table, or database	352
How to alter a table	354
How to work with sequences.....	356
How to create a sequence	356
How to use a sequence	356
How to delete a sequence	358
How to alter a sequence	358
How to work with collations.....	360
An introduction to encodings	360
An introduction to collations.....	362
How to view collations.....	364
How to specify a collation.....	366

The script used to create the AP database	368
How the script works.....	368
How the DDL statements work.....	368
Chapter 12 How to create a database and its tables with the Management Studio	
 How to work with a database.....	376
How to create a database	376
How to delete a database	376
 How to work with tables	378
How to create, modify, or delete a table.....	378
How to work with foreign key relationships.....	380
How to work with indexes and keys.....	382
How to work with check constraints	384
How to examine table dependencies	386
 How to generate scripts	388
How to generate scripts for databases and tables.....	390
How to generate a change script when you modify a table.....	392

Section 4 Advanced SQL skills

Chapter 13 How to work with views

An introduction to views	396
How views work.....	396
Benefits of using views	398
How to create and manage views.....	400
How to create a view	400
Examples that create views	402
How to create an updatable view	404
How to delete or modify a view	406
How to use views	408
How to update rows through a view.....	408
How to insert rows through a view.....	410
How to delete rows through a view	410
How to use the catalog views.....	412
How to use the View Designer.....	414
How to create or modify a view	414
How to delete a view	414

Chapter 14 How to code scripts

An introduction to scripts	418
How to work with scripts	418
The Transact-SQL statements for script processing.....	420
How to work with variables and temporary tables	422
How to work with scalar variables	422
How to work with table variables.....	424
How to work with temporary tables.....	426
A comparison of the five types of Transact-SQL table objects.....	428
How to control the execution of a script	430
How to perform conditional processing	430
How to test for the existence of a database object	432

How to perform repetitive processing	434
How to use a cursor	436
How to handle errors	438
How to use surround-with snippets	440

Advanced scripting techniques **442**

How to use the system functions	442
How to change the session settings	444
How to use dynamic SQL	446
A script that summarizes the structure of a database	448
How to use the SQLCMD utility	452

Chapter 15 How to code stored procedures, functions, and triggers**Procedural programming options in Transact-SQL** **458**

Scripts	458
Stored procedures, user-defined functions, and triggers	458

How to code stored procedures **460**

An introduction to stored procedures	460
How to create a stored procedure	462
How to declare and work with parameters	464
How to call procedures with parameters	466
How to work with return values	468
How to validate data and raise errors	470
A stored procedure that manages insert operations	472
How to pass a table as a parameter	478
How to delete or change a stored procedure	480
How to work with system stored procedures	482

How to code user-defined functions **484**

An introduction to user-defined functions	484
How to create a scalar-valued function	486
How to create a simple table-valued function	488
How to create a multi-statement table-valued function	490
How to delete or change a function	492

How to code triggers **494**

How to create a trigger	494
How to use AFTER triggers	496
How to use INSTEAD OF triggers	498
How to use triggers to enforce data consistency	500
How to use triggers to work with DDL statements	502
How to delete or change a trigger	504

Chapter 16 How to manage transactions and locking**How to work with transactions** **510**

How transactions maintain data integrity	510
SQL statements for handling transactions	512
How to work with nested transactions	514
How to work with save points	516

An introduction to concurrency and locking **518**

How concurrency and locking are related	518
The four concurrency problems that locks can prevent	520
How to set the transaction isolation level	522

How SQL Server manages locking	524
Lockable resources and lock escalation	524
Lock modes and lock promotion.....	526
Lock mode compatibility	528
How to prevent deadlocks	530
Two transactions that deadlock	530
Coding techniques that prevent deadlocks	532

Chapter 17 How to manage database security

How to work with SQL Server login IDs	536
An introduction to SQL Server security.....	536
How to change the authentication mode	538
How to create login IDs	540
How to delete or change login IDs or passwords.....	542
How to work with database users.....	544
How to work with schemas	546
How to work with permissions	548
How to grant or revoke object permissions.....	548
The SQL Server object permissions.....	550
How to grant or revoke schema permissions.....	552
How to grant or revoke database permissions.....	554
How to grant or revoke server permissions	556
How to work with roles	558
How to work with the fixed server roles	558
How to work with user-defined server roles.....	560
How to display information about server roles and role members	562
How to work with the fixed database roles	564
How to work with user-defined database roles	566
How to display information about database roles and role members	568
How to deny permissions granted by role membership	570
How to work with application roles	572
How to manage security using the Management Studio	574
How to work with login IDs.....	574
How to work with the server roles for a login ID	576
How to assign database access and roles by login ID.....	578
How to assign user permissions to database objects	580
How to work with database permissions.....	582

Chapter 18 How to work with XML

An introduction to XML	588
An XML document	588
An XML schema	590
How to work with the xml data type.....	592
How to store data in the xml data type.....	592
How to work with the XML Editor.....	594
How to use the methods of the xml data type	596
An example that parses the xml data type.....	600
Another example that parses the xml data type	602
How to work with XML schemas	604
How to add an XML schema to a database.....	604
How to use an XML schema to validate the xml data type.....	606
How to view an XML schema.....	608
How to drop an XML schema	608

Two more skills for working with XML	610
How to use the FOR XML clause of the SELECT statement.....	610
How to use the OPENXML statement.....	614
Chapter 19 How to work with BLOBs	
An introduction to BLOBs.....	620
Pros and cons of storing BLOBs in files	620
Pros and cons of storing BLOBs in a column	620
When to use FILESTREAM storage for BLOBs.....	620
How to use SQL to work with a varbinary(max) column	622
How to create a table with a varbinary(max) column	622
How to insert, update, and delete binary data	622
How to retrieve binary data.....	622
A .NET application that uses a varbinary(max) column.....	624
The user interface for the application.....	624
The event handlers for the form	626
A data access class that reads and writes binary data	628
How to use FILESTREAM storage.....	634
How to enable FILESTREAM storage on the server.....	634
How to create a database with FILESTREAM storage.....	636
How to create a table with a FILESTREAM column	638
How to insert, update, and delete FILESTREAM data.....	638
How to retrieve FILESTREAM data.....	638
A data access class that uses FILESTREAM storage	640
Appendix A How to set up your computer for this book	
Three editions of SQL Server 2019 Express.....	648
The tool for working with all editions of SQL Server	648
How to install SQL Server 2019 Express.....	650
How to install SQL Server Management Studio	650
How to install the files for this book	652
How to create the databases for this book.....	654
How to restore the databases for this book	654
How to install Visual Studio 2019 Community.....	656

Introduction

If you want to learn SQL, you've picked the right book. And if you want to learn the specifics of SQL for SQL Server 2019, you've made an especially good choice. Along the way, you'll learn a lot about relational database management systems in general and about SQL Server in particular.

Why learn SQL? First, because most programmers would be better at database programming if they knew more about SQL. Second, because SQL programming is a valuable specialty in itself. And third, because knowing SQL is the first step toward becoming a database administrator. In short, knowing SQL makes you more valuable on the job.

Who this book is for

This book is the ideal book for application developers who need to work with a SQL Server database. It shows you how to code the SQL statements that you need for your applications. It shows you how to code these statements so they run efficiently. And it shows you how to take advantage of the most useful advanced features that SQL Server has to offer.

This book is also a good choice for anyone who wants to learn standard SQL. Since SQL is a standard language for accessing database data, most of the SQL code in this book will work with any database management system. As a result, once you use this book to learn how to use SQL to work with a SQL Server database, you can transfer most of what you have learned to another database management system such as Oracle, DB2, or MySQL.

This book is also the right *first* book for anyone who wants to become a database administrator. Although this book doesn't present all of the advanced skills that are needed by a top DBA, it will get you started. Then, when you have finished this book, you'll be prepared for more advanced books on the subject.

4 reasons why you'll learn faster with this book

- Unlike most SQL books, this one starts by showing you how to query an existing database rather than how to create a new database. Why? Because that's what you're most likely to need to do first on the job. Once you master those skills, you can learn how to design and implement a database, whenever you need to do that. Or, you can learn how to work with other database features like views and stored procedures, whenever you need to do that.

- Like all our books, this one includes hundreds of examples that range from the simple to the complex. That way, you can quickly get the idea of how a feature works from the simple examples, but you'll also see how the feature is used in the real world from the complex examples.
- Like most of our books, this one has exercises at the end of each chapter that give you hands-on experience by letting you practice what you've learned. These exercises also encourage you to experiment and to apply what you've learned in new ways.
- If you page through this book, you'll see that all of the information is presented in "paired pages," with the essential syntax, guidelines, and examples on the right page and the perspective and extra explanation on the left page. This helps you learn more with less reading, and it is the ideal reference format when you need to refresh your memory about how to do something.

What you'll learn in this book

- In section 1, you'll learn the concepts and terms you need for working with any database. You'll also learn how to use Microsoft SQL Server 2019 and the Management Studio to run SQL statements on your own PC.
- In section 2, you'll learn all the skills for retrieving data from a database and for adding, updating, and deleting that data. These chapters move from the simple to the complex so you won't have any trouble if you're a SQL novice. And they present skills like using outer joins, summary queries, and subqueries that will raise your SQL expertise if you already have SQL experience.
- In section 3, you'll learn how to design a database and how to implement that design by using either SQL statements or the Management Studio. When you're done, you'll be able to design and implement your own databases. But even if you're never called upon to do that, this section will give you perspective that will make you a better SQL programmer.
- In section 4, you'll learn the skills for working with database features like views, scripts, stored procedures, functions, triggers, and transactions. In addition, you'll learn the skills for working with database security, XML, and BLOBs. These are the features that give a database much of its power. So once you master them, you'll have a powerful set of SQL skills.

Prerequisites

Although you will progress through this book more quickly if you have some development experience, everything you need to know about databases and SQL is presented in this book. As a result, you don't need to have any programming background to use this book to learn SQL.

However, if you want to use C# or Visual Basic to work with a SQL Server database as described in chapter 19, you need to have some experience using C# or Visual Basic to write ADO.NET code. If you don't already have that experience, you can refer to the appropriate chapter in the current edition of *Murach's C#* or *Murach's Visual Basic*.

What software you need for this book

All of the software you need for this book is available from Microsoft's website for free. That includes:

- SQL Server 2019 Express (only runs on Windows 10 and later)
- SQL Server Management Studio
- Visual Studio Community (only necessary for chapter 19)

In appendix A, you'll find complete instructions for installing these items on your PC.

However, SQL Server 2019 only runs on Windows 10 and later. As a result, if you have an earlier version of Windows, such as Windows 8, you'll need to upgrade your operating system to Windows 10 or later.

What you can download from our website

You can download all the source code for this book from our website. That includes:

- Scripts that create the databases used by this book
- The source code for all examples in this book
- The solutions for all exercises in this book

In appendix A, you'll find complete instructions for installing these items on your PC.

Support materials for trainers and instructors

If you're a corporate trainer or a college instructor who would like to use this book for a course, we offer these supporting materials: (1) a complete set of PowerPoint slides that you can use to review and reinforce the content of this book; (2) instructional objectives that describe the skills a student should have upon completion of each chapter; (3) test banks that measure mastery of those skills; (4) additional exercises for each chapter that aren't in this book; and (5) solutions to those exercises.

To learn more about these materials, please go to our website at www.murachforinstructors.com if you're an instructor. If you're a trainer, please go to www.murach.com and click on the *Courseware for Trainers* link, or contact Kelly at 1-800-221-5528 or kelly@murach.com.

Please let us know how this book works for you

When we started working on this book, our goal was (1) to provide a SQL Server book for application developers that will help them work more effectively; (2) to cover the database design and implementation skills that application developers are most likely to use; and (3) to do both in a way that helps you learn faster and better than you can with any other SQL Server book.

Now, if you have any comments about this book, we would appreciate hearing from you. If you like this book, please tell a friend. And good luck with your SQL Server projects!



Joel Murach
Author
joel@murach.com

Section 1

An introduction to SQL

Before you begin to learn the fundamentals of programming in SQL, you need to understand the concepts and terms related to SQL and relational databases. That's what you'll learn in chapter 1. Then, in chapter 2, you'll learn about some of the tools you can use to work with a SQL Server database. That will prepare you for using the skills you'll learn in the rest of this book.

1

An introduction to relational databases and SQL

Before you can use SQL to work with a SQL Server database, you need to be familiar with the concepts and terms that apply to database systems. In particular, you need to understand what a relational database is. That's what you'll learn in the first part of this chapter. Then, you'll learn about some of the basic SQL statements and features provided by SQL Server.

An introduction to client/server systems	4
The hardware components of a client/server system	4
The software components of a client/server system.....	6
Other client/server system architectures	8
An introduction to the relational database model	10
How a database table is organized	10
How the tables in a relational database are related	12
How the columns in a table are defined	14
How relational databases compare to other data models	16
An introduction to SQL and SQL-based systems.....	18
A brief history of SQL.....	18
A comparison of Oracle, DB2, MySQL, and SQL Server	20
The Transact-SQL statements	22
An introduction to the SQL statements	22
Typical statements for working with database objects.....	24
How to query a single table	26
How to join data from two or more tables.....	28
How to add, update, and delete data in a table	30
SQL coding guidelines	32
How to work with other database objects	34
How to work with views	34
How to work with stored procedures, triggers, and user-defined functions.....	36
How to use SQL from an application program.....	38
Common data access models.....	38
How to use ADO.NET from a .NET application	40
Visual Basic code that retrieves data from a SQL Server database	42
C# code that retrieves data from a SQL Server database	44
Perspective	46

An introduction to client/server systems

In case you aren't familiar with client/server systems, the first two topics that follow introduce you to their essential hardware and software components. These are the types of systems that you're most likely to use SQL with. Then, the last topic gives you an idea of how complex client/server systems can be.

The hardware components of a client/server system

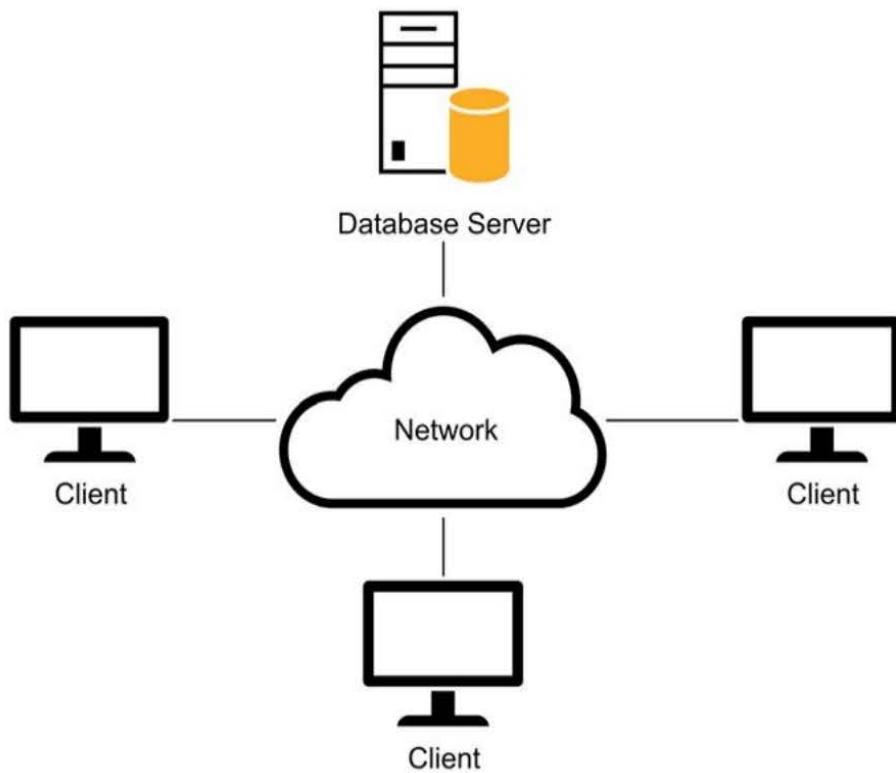
Figure 1-1 presents the three hardware components of a client/server system: the clients, the network, and the server. The *clients* are usually the PCs that are already available on the desktops throughout a company. And the *network* is the cabling, communication lines, network interface cards, hubs, routers, and other components that connect the clients and the server.

The *server*, commonly referred to as a *database server*, is a computer that has enough processor speed, internal memory (RAM), and disk storage to store the files and databases of the system and provide services to the clients of the system. This computer is usually a high-powered PC, but it can also be a midrange system like an IBM Power System or a Unix system, or even a mainframe system. When a system consists of networks, midrange systems, and mainframe systems, often spread throughout the country or world, it is commonly referred to as an *enterprise system*.

To back up the files of a client/server system, a server usually has a disk drive or some other form of offline storage. It often has one or more printers or specialized devices that can be shared by the users of the system. And it can provide programs or services like e-mail that can be accessed by all the users of the system.

In a simple client/server system, the clients and the server are part of a *local area network (LAN)*. However, two or more LANs that reside at separate geographical locations can be connected as part of a larger network such as a *wide area network (WAN)*. In addition, individual systems or networks can be connected over the Internet.

A simple client/server system



The three hardware components of a client/server system

- The *clients* are the PCs, Macs, or workstations of the system.
- The *server* is a computer that stores the files and databases of the system and provides services to the clients. When it stores databases, it's often referred to as a *database server*.
- The *network* consists of the cabling, communication lines, and other components that connect the clients and the servers of the system.

Client/server system implementations

- In a simple *client/server system* like the one shown above, the server is typically a high-powered PC that communicates with the clients over a *local area network (LAN)*.
- The server can also be a midrange system, like an IBM Power System or a Unix system, or it can be a mainframe system. Then, special hardware and software components are required to make it possible for the clients to communicate with the midrange and mainframe systems.
- A client/server system can also consist of one or more PC-based systems, one or more midrange systems, and a mainframe system in dispersed geographical locations. This type of system is commonly referred to as an *enterprise system*.
- Individual systems and LANs can be connected and share data over larger private networks, such as a *wide area network (WAN)*, or a public network like the Internet.

Figure 1-1 The hardware components of a client/server system

The software components of a client/server system

Figure 1-2 presents the software components of a typical client/server system. In addition to a *network operating system* that manages the functions of the network, the server requires a *database management system (DBMS)* like Microsoft SQL Server or Oracle. This DBMS manages the databases that are stored on the server.

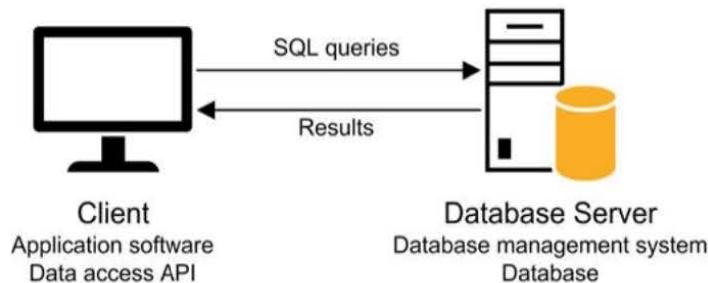
In contrast to a server, each client requires *application software* to perform useful work. This can be a purchased software package like a financial accounting package, or it can be custom software that's developed for a specific application.

Although the application software is run on the client, it uses data that's stored on the server. To do that, it uses a *data access API (application programming interface)* such as ADO.NET. Since the technique you use to work with an API depends on the programming language and API you're using, you won't learn those techniques in this book. Instead, you'll learn about a standard language called *SQL*, or *Structured Query Language*, that lets any application communicate with any DBMS. (In conversation, SQL is pronounced as either *S-Q-L* or *sequel*.)

Once the software for both client and server is installed, the client communicates with the server via *SQL queries* (or just *queries*) that are passed to the DBMS through the API. After the client sends a query to the DBMS, the DBMS interprets the query and sends the results back to the client.

As you can see in this figure, the processing done by a client/server system is divided between the clients and the server. In this case, the DBMS on the server is processing requests made by the application running on the client. Theoretically, at least, this balances the workload between the clients and the server so the system works more efficiently. By contrast, in a file-handling system, the clients do all of the work because the server is used only to store the files that are used by the clients.

Client software, server software, and the SQL interface



Server software

- To store and manage the databases of the client/server system, each server requires a *database management system (DBMS)* like Microsoft SQL Server.
- The processing that's done by the DBMS is typically referred to as *back-end processing*, and the database server is referred to as the *back end*.

Client software

- The *application software* does the work that the user wants to do. This type of software can be purchased or developed.
- The *data access API (application programming interface)* provides the interface between the application program and the DBMS. The current Microsoft API is ADO.NET, which can communicate directly with SQL Server. Older APIs required a data access model, such as ADO or DAO, plus a driver, such as OLE DB or ODBC.
- The processing that's done by the client software is typically referred to as *front-end processing*, and the client is typically referred to as the *front end*.

The SQL interface

- The application software communicates with the DBMS by sending *SQL queries* through the data access API. When the DBMS receives a query, it provides a service like returning the requested data (the *query results*) to the client.
- *SQL* stands for *Structured Query Language*, which is the standard language for working with a relational database.

Client/server versus file-handling systems

- In a client/server system, the processing done by an application is typically divided between the client and the server.
- In a file-handling system, all of the processing is done on the clients. Although the clients may access data that's stored in files on the server, none of the processing is done by the server. As a result, a file-handling system isn't a client/server system.

Figure 1-2 The software components of a client/server system

Other client/server system architectures

In its simplest form, a client/server system consists of a single database server and one or more clients. Many client/server systems today, though, include additional servers. In figure 1-3, for example, you can see two client/server systems that include an additional server between the clients and the database server.

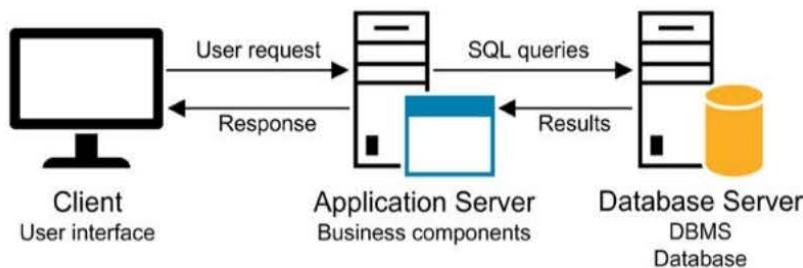
The first illustration is for a simple Windows-based system. With this system, only the user interface for an application runs on the client. The rest of the processing that's done by the application is stored in one or more *business components* on the *application server*. Then, the client sends requests to the application server for processing. If the request involves accessing data in a database, the application server formulates the appropriate query and passes it on to the database server. The results of the query are then sent back to the application server, which processes the results and sends the appropriate response back to the client.

Similar processing is done by a web-based system, as illustrated by the second example in this figure. In this case, though, a *web browser* running on the client is used to send requests to a *web application* running on a *web server* somewhere on the Internet. The web application, in turn, can use *web services* to perform some of its processing. Then, the web application or web service can pass requests for data on to the database server.

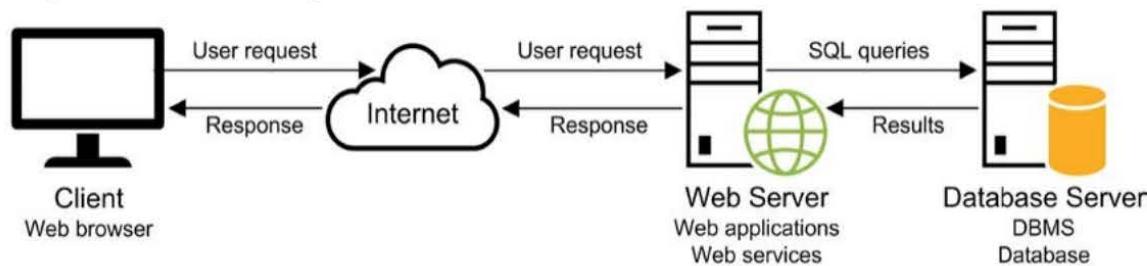
Although this figure should give you an idea of how client/server systems can be configured, you should realize that they can be much more complicated than what's shown here. In a Windows-based system, for example, business components can be distributed over any number of application servers, and those components can communicate with databases on any number of database servers. Similarly, the web applications and services in a web-based system can be distributed over numerous web servers that access numerous database servers. In most cases, though, it's not necessary for you to know how a system is configured to use SQL.

Before I go on, you should know that client/server systems aren't the only systems that support SQL. For example, traditional mainframe systems and newer *thin client* systems also use SQL. Unlike client/server systems, though, most of the processing for these types of systems is done by a mainframe or another high-powered machine. The terminals or PCs that are connected to the system do little or no work.

A Windows-based system that uses an application server



A simple web-based system



Description

- In addition to a database server and clients, a client/server system can include additional servers, such as *application servers* and *web servers*.
- Application servers are typically used to store *business components* that do part of the processing of the application. In particular, these components are used to process database requests from the user interface running on the client.
- Web servers are typically used to store *web applications* and *web services*. Web applications are applications that are designed to run on a web server. Web services are like business components, except that, like web applications, they are designed to run on a web server.
- In a web-based system, a *web browser* running on a client sends a request to a web server over the Internet. Then, the web server processes the request and passes any requests for data on to the database server.
- More complex system architectures can include two or more application servers, web servers, and database servers.

Figure 1-3 Other client/server system architectures

An introduction to the relational database model

In 1970, Dr. E. F. Codd developed a model for a new type of database called a *relational database*. This type of database eliminated some of the problems that were associated with standard files and other database designs. By using the relational model, you can reduce data redundancy, which saves disk storage and leads to efficient data retrieval. You can also view and manipulate data in a way that is both intuitive and efficient. Today, relational databases are the de facto standard for database applications.

How a database table is organized

The model for a relational database states that data is stored in one or more *tables*. It also states that each table can be viewed as a two-dimensional matrix consisting of *rows* and *columns*. This is illustrated by the relational table in figure 1-4. Each row in this table contains information about a single vendor.

In practice, the rows and columns of a relational database table are often referred to by the more traditional terms, *records* and *fields*. In fact, some software packages use one set of terms, some use the other, and some use a combination. This book uses the terms *rows* and *columns* because those are the terms used by SQL Server.

In general, each table is modeled after a real-world entity such as a vendor or an invoice. Then, the columns of the table represent the attributes of the entity such as name, address, and phone number. And each row of the table represents one instance of the entity. A value is stored at the intersection of each row and column, sometimes called a *cell*.

If a table contains one or more columns that uniquely identify each row in the table, you can define these columns as the *primary key* of the table. For instance, the primary key of the Vendors table in this figure is the VendorID column. In this example, the primary key consists of a single column. However, a primary key can also consist of two or more columns, in which case it's called a *composite primary key*.

In addition to primary keys, some database management systems let you define additional keys that uniquely identify each row in a table. If, for example, the VendorName column in the Vendors table contains unique data, it can be defined as a *non-primary key*. In SQL Server, this is called a *unique key*.

Indexes provide an efficient way of accessing the rows in a table based on the values in one or more columns. Because applications typically access the rows in a table by referring to their key values, an index is automatically created for each key you define. However, you can define indexes for other columns as well. If, for example, you frequently need to sort the Vendor rows by zip code, you can set up an index for that column. Like a key, an index can include one or more columns.

The Vendors table in an Accounts Payable database

		Columns			
	VendorID	VendorName	VendorAddress1	VendorAddress2	VendorCity
1	1	US Postal Service	Attn: Supt. Window Services	PO Box 7005	Madison
2	2	National Information Data Ctr	PO Box 96621	NULL	Washington
3	3	Register of Copyrights	Library Of Congress	NULL	Washington
4	4	Jobtrak	1990 Westwood Blvd Ste 260	NULL	Los Angeles
5	5	Newbridge Book Clubs	3000 Cindel Drive	NULL	Washington
6	6	California Chamber Of Commerce	3255 Ramos Cir	NULL	Sacramento
7	7	Towne Advertiser's Mailing Svcs	Kevin Minder	3441 W Macarthur Blvd	Santa Ana
8	8	BFI Industries	PO Box 9369	NULL	Fresno
9	9	Pacific Gas & Electric	Box 52001	NULL	San Francisco
10	10	Robbins Mobile Lock And Key	4669 N Fresno	NULL	Fresno
11	11	Bill Marvin Electric Inc	4583 E Home	NULL	Fresno
12	12	City Of Fresno	PO Box 2069	NULL	Fresno
13	13	Golden Eagle Insurance Co	PO Box 85826	NULL	San Diego
14	14	Expedata Inc	4420 N. First Street, Suite 108	NULL	Fresno
15	15	ASC Signs	1528 N Sierra Vista	NULL	Fresno
16	16	Internal Revenue Service	NULL	NULL	Fresno

Concepts

- A *relational database* consists of *tables*. Tables consist of *rows* and *columns*, which can also be referred to as *records* and *fields*.
- A table is typically modeled after a real-world entity, such as an invoice or a vendor.
- A column represents some attribute of the entity, such as the amount of an invoice or a vendor's address.
- A row contains a set of values for a single instance of the entity, such as one invoice or one vendor.
- The intersection of a row and a column is sometimes called a *cell*. A cell stores a single value.
- Most tables have a *primary key* that uniquely identifies each row in the table. The primary key is usually a single column, but it can also consist of two or more columns. If a primary key uses two or more columns, it's called a *composite primary key*.
- In addition to primary keys, some database management systems let you define one or more *non-primary keys*. In SQL Server, these keys are called *unique keys*. Like a primary key, a non-primary key uniquely identifies each row in the table.
- A table can also be defined with one or more *indexes*. An index provides an efficient way to access data from a table based on the values in specific columns. An index is automatically created for a table's primary and non-primary keys.

Figure 1-4 How a database table is organized

How the tables in a relational database are related

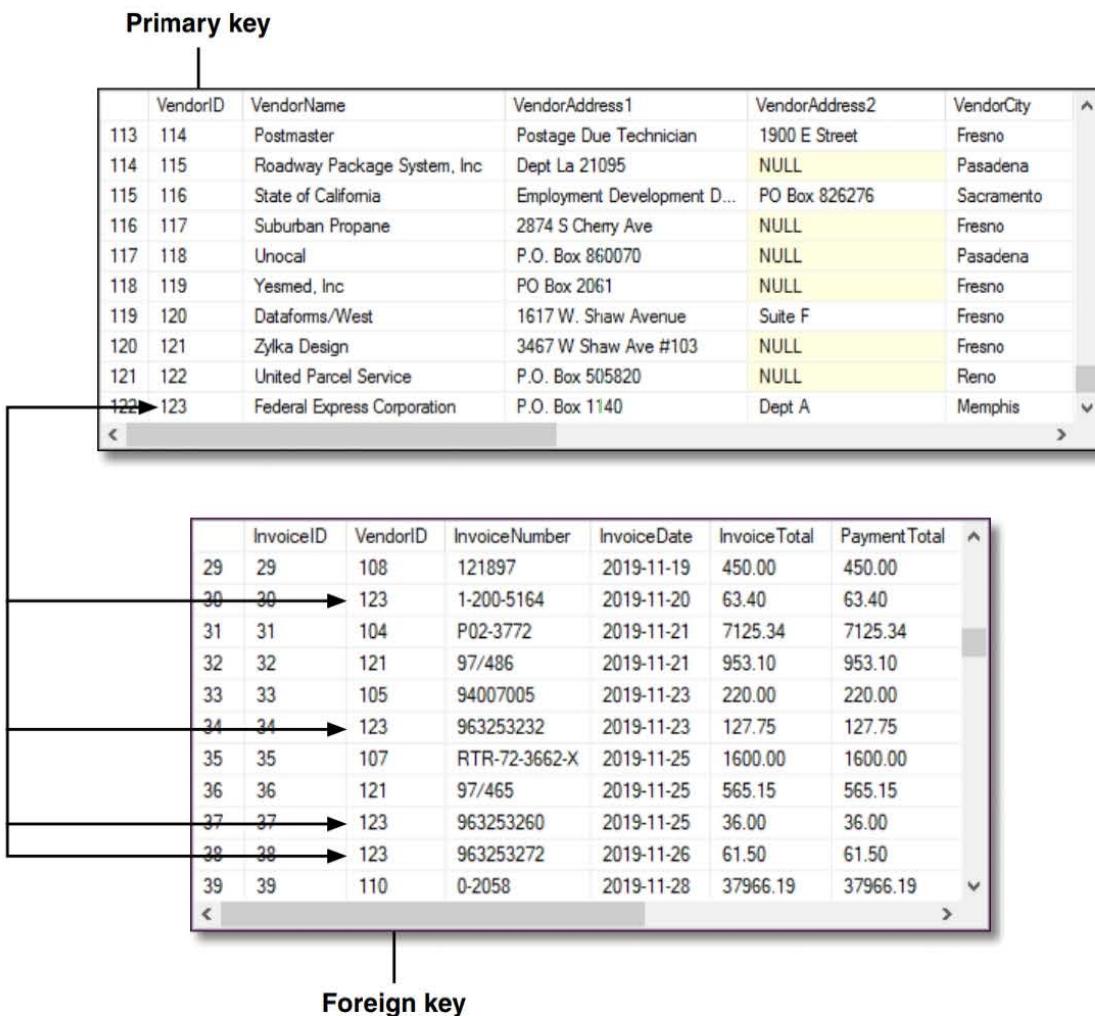
The tables in a relational database can be related to other tables by values in specific columns. The two tables shown in figure 1-5 illustrate this concept. Here, each row in the Vendors table is related to one or more rows in the Invoices table. This is called a *one-to-many relationship*.

Typically, relationships exist between the primary key in one table and the *foreign key* in another table. The foreign key is simply one or more columns in a table that refer to a primary key in another table. In SQL Server, relationships can also exist between a unique key in one table and a foreign key in another table.

Although one-to-many relationships are the most common, two tables can also have a one-to-one or many-to-many relationship. If a table has a *one-to-one relationship* with another table, the data in the two tables could be stored in a single table. However, it's often useful to store large objects such as images, sound, and videos in a separate table. Then, you can join the two tables with the one-to-one relationship only when the large objects are needed.

By contrast, a *many-to-many relationship* is usually implemented by using an intermediate table that has a one-to-many relationship with the two tables in the many-to-many relationship. In other words, a many-to-many relationship can usually be broken down into two one-to-many relationships.

The relationship between the Vendors and Invoices tables in the database



Concepts

- The tables in a relational database are related to each other through their key columns. For example, the VendorID column is used to relate the Vendors and Invoices tables above. The VendorID column in the Invoices table is called a *foreign key* because it identifies a related row in the Vendors table. A table may contain one or more foreign keys.
- When you define a foreign key for a table in SQL Server, you can't add rows to the table with the foreign key unless there's a matching primary key in the related table.
- The relationships between the tables in a database correspond to the relationships between the entities they represent. The most common type of relationship is a *one-to-many relationship* as illustrated by the Vendors and Invoices tables. A table can also have a *one-to-one relationship* or a *many-to-many relationship* with another table.

Figure 1-5 How the tables in a relational database are related

How the columns in a table are defined

When you define a column in a table, you assign properties to it as indicated by the design of the Invoices table in figure 1-6. The most critical property for a column is its data type, which determines the type of information that can be stored in the column. With SQL Server 2019, you typically use one of the *data types* listed in this figure. As you define each column in a table, you generally try to assign the data type that will minimize the use of disk storage because that will improve the performance of the queries later.

In addition to a data type, you must identify whether the column can store a *null value*. A null represents a value that's unknown, unavailable, or not applicable. If you don't allow null values, then you must provide a value for the column or you can't store the row in the table.

You can also assign a *default value* to each column. Then, that value is assigned to the column if another value isn't provided. You'll learn more about how to work with nulls and default values later in this book.

Each table can also contain a numeric column whose value is generated automatically by the DBMS. In SQL Server, a column like this is called an *identity column*, and you establish it using the Is Identity, Identity Seed, and Identity Increment properties. You'll learn more about these properties in chapter 11. For now, just note that the primary key of both the Vendors and the Invoices tables—VendorID and InvoiceID—are identity columns.

The columns of the Invoices table

The screenshot shows the Microsoft SQL Server Management Studio interface. On the left, the Object Explorer displays the database structure under 'localhost\SQLEXPRESS (SQL Server 15.0.2000 - murach\Anne)'. Under the 'AP' database, the 'Tables' node is expanded, showing the 'dbo.Invoices' table. The 'Columns' node under 'dbo.Invoices' lists the columns: InvoiceID, VendorID, InvoiceNumber, InvoiceDate, InvoiceTotal, PaymentTotal, CreditTotal, TermsID, InvoiceDueDate, and PaymentDate. The 'InvoiceID' column is highlighted. On the right, the 'MMA17\SQLEXPRESS.AP - dbo.Invoices' tab is active in the 'Table Designer'. It shows a grid of columns with their names, data types, and 'Allow Nulls' settings. Below the grid, the 'Column Properties' window is open, specifically showing the 'Identity Specification' section. The 'Is Identity' property is set to 'Yes', 'Identity Increment' is 1, and 'Identity Seed' is 1. The 'Indexable' property is set to 'Yes'.

Common SQL Server data types

Type	Description
bit	A value of 1 or 0 that represents a True or False value.
int, bigint, smallint, tinyint	Integer values of various sizes.
money, smallmoney	Monetary values that are accurate to four decimal places.
decimal, numeric	Decimal values that are accurate to the least significant digit. The values can contain an integer portion and a decimal portion.
float, real	Floating-point values that contain an approximation of a decimal value.
date, time, datetime2	Dates and times.
char, varchar	A string of letters, symbols, and numbers in the ASCII character set.
nchar, nvarchar	A string of letters, symbols, and numbers in the Unicode character set.

Description

- The *data type* that's assigned to a column determines the type and size of the information that can be stored in the column.
- Each column definition also indicates whether or not it can contain *null values*. A null value indicates that the value of the column is unknown.
- A column can also be defined with a *default value*. Then, that value is used if another value isn't provided when a row is added to the table.
- A column can also be defined as an *identity column*. An identity column is a numeric column whose value is generated automatically when a row is added to the table.

Figure 1-6 How the columns in a table are defined

How relational databases compare to other data models

Now that you understand how a relational database is organized, you're ready to learn how relational databases differ from other data models.

Specifically, you should know how relational databases compare to conventional file systems, *hierarchical databases*, and *network databases*. Figure 1-7 presents the most important differences.

To start, you should realize that because the physical structure of a relational database is defined and managed by the DBMS, it's not necessary to define that structure within the programs that use the database. Instead, you can simply refer to the tables and columns you want to use by name and the DBMS will take care of the rest. By contrast, when you use a conventional file system, you have to define and control the files of the system within each application that uses them. That's because a conventional file system is just a collection of files that contain the data of the system. In addition, if you modify the structure of a file, you have to modify every program that uses it. That's not necessary with a relational database.

The hierarchical and network database models were predecessors to the relational database model. The hierarchical database model is limited in that it can only represent one-to-many relationships, also called *parent/child relationships*. The network database model is an extension of the hierarchical model that provides for all types of relationships.

Although hierarchical and network databases don't have the same drawbacks as conventional file systems, they still aren't as easy to use as relational databases. In particular, each program that uses a hierarchical or network database must navigate through the physical layout of the tables they use. By contrast, this navigation is automatically provided by the DBMS in a relational database system. In addition, programs can define ad hoc relationships between the tables of a relational database. In other words, they can use relationships that aren't defined by the DBMS. That's not possible with hierarchical and network databases.

Another type of database that's not mentioned in this figure is the *object database*. This type of database is designed to store and retrieve the objects that are used by applications written in an object-oriented programming language such as C#, C++, or Java. Although object databases have some advantages over relational databases, they also have some disadvantages. In general, object databases have not yet become widely used. However, they have acquired a niche in some areas such as engineering, telecommunications, financial services, high energy physics, and molecular biology.

A comparison of relational databases and conventional file systems

Feature	Conventional file system	Relational database
Definition	Each program that uses the file must define the file and the layout of the records within the file	Tables, rows, and columns are defined within the database and can be accessed by name
Maintenance	If the definition of a file changes, each program that uses the file must be modified	Programs can be used without modification when the definition of a table changes
Validity checking	Each program that updates a file must include code to check for valid data	Can include checks for valid data
Relationships	Each program must provide for and enforce relationships between files	Can enforce relationships between tables using foreign keys; ad hoc relationships can also be used
Data access	Each I/O operation targets a specific record in a file based on its relative position in the file or its key value	A program can use SQL to access selected data in one or more tables of a database

A comparison of relational databases and other database systems

Feature	Hierarchical database	Network database	Relational database
Supported relationships	One-to-many only	One-to-many, one-to-one, and many-to-many	One-to-many, one-to-one, and many-to-many; ad hoc relationships can also be used
Data access	Programs must include code to navigate through the physical structure of the database	Programs must include code to navigate through the physical structure of the database	Programs can access data without knowing its physical structure
Maintenance	New and modified relationships can be difficult to implement in application programs	New and modified relationships can be difficult to implement in application programs	Programs can be used without modification when the definition of a table changes

Description

- To work with any of the data models other than the relational database model, you must know the physical structure of the data and the relationships between the files or tables.
- Because relationships are difficult to implement in a conventional file system, redundant data is often stored in these types of files.
- The *hierarchical database* model provides only for one-to-many relationships, called *parent/child relationships*.
- The *network database* model can accommodate any type of relationship.
- Conventional files, hierarchical databases, and network databases are all more efficient than relational databases because they require fewer system resources. However, the flexibility and ease of use of relational databases typically outweigh this inefficiency.

Figure 1-7 How relational databases compare to other data models

An introduction to SQL and SQL-based systems

In the topics that follow, you'll learn how SQL and SQL-based database management systems evolved. In addition, you'll learn how some of the most popular SQL-based systems compare.

A brief history of SQL

Prior to the release of the first *relational database management system (RDBMS)*, each database had a unique physical structure and a unique programming language that the programmer had to understand. That all changed with the advent of SQL and the relational database management system.

Figure 1-8 lists the important events in the history of SQL. In 1970, Dr. E. F. Codd published an article that described the relational database model he had been working on with a research team at IBM. By 1978, the IBM team had developed a database system based on this model, called System/R, along with a query language called *SEQUEL (Structured English Query Language)*. Although the database and query language were never officially released, IBM remained committed to the relational model.

The following year, Relational Software, Inc. released the first relational database management system, called *Oracle*. This RDBMS ran on a minicomputer and used SQL as its query language. This product was widely successful, and the company later changed its name to Oracle to reflect that success.

In 1982, IBM released its first commercial SQL-based RDBMS, called *SQL/DS (SQL/Data System)*. This was followed in 1985 by *DB2 (Database 2)*. Both systems ran only on IBM mainframe computers. Later, DB2 was ported to other systems, including those that ran the Unix, Linux, and Windows operating systems. Today, it continues to be IBM's premier database system.

During the 1980s, other SQL-based database systems, including SQL Server, were developed. Although each of these systems used SQL as its query language, each implementation was unique. That began to change in 1989, when the *American National Standards Institute (ANSI)* published its first set of standards for a database query language. These standards have been revised a few times since then, most recently in 2016. As each database manufacturer has attempted to comply with these standards, their implementations of SQL have become more similar. However, each still has its own *dialect* of SQL that includes additions, or *extensions*, to the standards.

Although you should be aware of the SQL standards, they will have little effect on your job as a SQL programmer. The main benefit of the standards is that the basic SQL statements are the same in each dialect. As a result, once you've learned one dialect, it's relatively easy to learn another. On the other hand, porting applications that use SQL from one database to another isn't as easy as it should be. In fact, any non-trivial application will require at least modest modifications.

Important events in the history of SQL

Year	Event
1970	Dr. E. F. Codd developed the relational database model.
1978	IBM developed the predecessor to SQL, called Structured English Query Language (SEQUEL). This language was used on a database system called System/R, but neither the system nor the query language was ever released.
1979	Relational Software, Inc. (later renamed Oracle) released the first relational DBMS, Oracle.
1982	IBM released their first relational database system, SQL/DS (SQL/Data System).
1985	IBM released DB2 (Database 2).
1987	Microsoft released SQL Server.
1989	The American National Standards Institute (ANSI) published the first set of standards for a database query language, called ANSI/ISO SQL-89, or SQL1. Because they were not stringent standards, most commercial products could claim adherence.
1992	ANSI published revised standards (ANSI/ISO SQL-92, or SQL2) that were more stringent than SQL1 and incorporated many new features. These standards introduced levels of conformance that indicated the extent to which a dialect met the standards.
1999	ANSI published SQL3 (ANSI/ISO SQL:1999), which incorporated new features, including support for objects. Levels of conformance were dropped and were replaced by a core specification along with specifications for nine additional packages.
2003	ANSI published SQL:2003, which introduced XML-related features, standardized sequences, and identity columns.
2006	ANSI published SQL:2006, which defined how SQL can be used with XML. The standards also allowed applications to integrate XQuery into their SQL code.
2008	ANSI published SQL:2008, which introduced INSTEAD OF triggers and the TRUNCATE statement.
2011	ANSI published SQL:2011, which included improved support for temporal databases.
2016	ANSI published SQL:2016, which introduced polymorphic table functions, row pattern recognition, and support for JSON.

Description

- SQL-92 initially provided for three *levels of conformance*: entry, intermediate, and full. A transitional level was later added between the entry and intermediate levels.
- SQL:1999 includes a *core specification* that defines the essential elements for conformance, plus nine *packages*. Each package is designed to serve a specific market niche.
- Although SQL is a standard language, each vendor has its own *SQL dialect*, or *variant*, that may include extensions to the standards. SQL Server's SQL dialect is called *Transact-SQL*.

How knowing “standard SQL” helps you

- The most basic SQL statements are the same for all SQL dialects.
- Once you have learned one SQL dialect, you can easily learn other dialects.

How knowing “standard SQL” does not help you

- Any non-trivial application will require modification when moved from one SQL database to another.

Figure 1-8 A brief history of SQL

A comparison of Oracle, DB2, MySQL, and SQL Server

Although this book is about SQL Server, you may want to know about some of the other SQL-based relational database management systems. Figure 1-9 compares SQL Server with three of the most popular: Oracle, DB2, and MySQL.

Oracle has a huge installed base of customers and continues to dominate the marketplace, especially for servers running the Unix or Linux operating system. Oracle works well for large systems and has a reputation for being extremely reliable, but also has a reputation for being expensive and difficult to use.

DB2 was originally designed to run on IBM mainframe systems and continues to be the premier database for those systems. It also dominates in hybrid environments where IBM mainframes and newer servers must coexist. Although it has a reputation for being expensive, it also has a reputation for being reliable and easy to use.

MySQL runs on all major operating systems and is widely used for web applications. MySQL is an *open-source database*, which means that any developer can view and improve its source code. In addition, the MySQL Community Server is free for most users, although Oracle also sells other editions of MySQL that include customer support and advanced features.

SQL Server was designed by Microsoft to run on Windows and is widely used for small- to medium-sized departmental systems. It has a reputation for being inexpensive and easy to use.

Until 2016, SQL Server ran only under the Windows operating system. By contrast, Oracle and MySQL ran under most modern operating systems. As a result, if a company used Linux as the operating system for its database servers, it couldn't use SQL Server and had to use Oracle or MySQL. However, in 2016, Microsoft released a preview version of SQL Server that runs under Linux, and it released a final version in 2017. This should allow SQL Server to compete with Oracle and MySQL when a company prefers to use Linux, not Windows, for its database servers. Although this book focuses on using SQL Server with Windows, the database engine is basically the same for Linux.

If you search the Internet, you'll find that dozens of other relational database products are also available. These include proprietary databases like Informix, SQL Anywhere, and Teradata. And they include open-source databases like PostgreSQL.

A comparison of Oracle, DB2, MySQL, and SQL Server

	Oracle	DB2	MySQL	SQL Server
Released	1979	1985	2000	1987
Platforms	Unix/Linux z/OS Windows macOS	OS/390, z/OS, and AIX Unix/Linux Windows macOS	Unix/Linux Windows macOS	Windows Linux

Description

- Oracle is typically used for large, mission-critical systems that run on one or more Unix servers.
- DB2 is typically used for large, mission-critical systems that run on legacy IBM mainframe systems using the z/OS or OS/390 operating system.
- MySQL is a popular *open-source database* that runs on all major operating systems and is commonly used for web applications.
- SQL Server is typically used for small- to medium-sized systems that run on one or more Windows servers. However, SQL Server 2017 and later are designed to run on Linux as well as Windows.

Figure 1-9 A comparison of Oracle, DB2, MySQL, and SQL Server

The Transact-SQL statements

In the topics that follow, you'll learn about some of the SQL statements provided by SQL Server. As you'll see, you can use some of these statements to manipulate the data in a database, and you can use others to work with database objects. Although you may not be able to code these statements after reading these topics, you should have a good idea of how they work. Then, you'll be better prepared to learn the details of coding these statements when they're presented in sections 2 and 3 of this book.

An introduction to the SQL statements

Figure 1-10 summarizes some of the most common SQL statements. As you can see, these statements can be divided into two categories. The statements that work with the data in a database are called the *data manipulation language (DML)*. These four statements are the ones that application programmers use the most. You'll see how these statements work later in this chapter, and you'll learn the details of using them in section 2 of this book.

The statements that work with the objects in a database are called *the data definition language (DDL)*. On large systems, these statements are used exclusively by *database administrators*, or *DBAs*. It's the DBA's job to maintain existing databases, tune them for faster performance, and create new databases. On smaller systems, though, the SQL programmer may also be the DBA. You'll see examples of some of these statements in the next figure, and you'll learn how to use them in chapter 11.

SQL statements used to work with data (DML)

Statement	Description
SELECT	Retrieves data from one or more tables.
INSERT	Adds one or more new rows to a table.
UPDATE	Changes one or more existing rows in a table.
DELETE	Deletes one or more existing rows from a table.

SQL statements used to work with database objects (DDL)

Statement	Description
CREATE DATABASE	Creates a new database.
CREATE TABLE	Creates a new table in a database.
CREATE INDEX	Creates a new index for a table.
ALTER TABLE	Changes the structure of an existing table.
ALTER INDEX	Changes the structure of an existing index.
DROP DATABASE	Deletes an existing database.
DROP TABLE	Deletes an existing table.
DROP INDEX	Deletes an existing index.

Description

- The SQL statements can be divided into two categories: the *data manipulation language (DML)* that lets you work with the data in the database and the *data definition language (DDL)* that lets you work with the objects in the database.
- SQL programmers typically work with the DML statements, while *database administrators (DBAs)* use the DDL statements.

Figure 1-10 An introduction to the SQL statements

Typical statements for working with database objects

To give you an idea of how you use the DDL statements you saw in the previous figure, figure 1-11 presents five examples. The first statement creates an accounts payable database named AP. This is the database that's used in many of the examples throughout this book.

The second statement creates the Invoices table you saw earlier in this chapter. If you don't understand all of this code right now, don't worry. You'll learn how to code statements like this later in this book. For now, just realize that this statement defines each column in the table, including its data type, whether or not it allows null values, and its default value if it has one. In addition, it identifies identity columns, primary key columns, and foreign key columns.

The third statement in this figure changes the Invoices table by adding a column to it. Like the statement that created the table, this statement specifies all the attributes of the new column. Then, the fourth statement deletes the column that was just added.

The last statement creates an index on the Invoices table. In this case, the index is for the VendorID column, which is used frequently to access the table. Notice the name that's given to this index. This follows the standard naming conventions for indexes, which you'll learn about in chapter 11.

A statement that creates a new database

```
CREATE DATABASE AP;
```

A statement that creates a new table

```
CREATE TABLE Invoices
(
    InvoiceID      INT          NOT NULL IDENTITY PRIMARY KEY,
    VendorID       INT          NOT NULL REFERENCES Vendors(VendorID),
    InvoiceNumber  VARCHAR(50)  NOT NULL,
    InvoiceDate    DATE         NOT NULL,
    InvoiceTotal   MONEY        NOT NULL,
    PaymentTotal   MONEY        NOT NULL DEFAULT 0,
    CreditTotal    MONEY        NOT NULL DEFAULT 0,
    TermsID        INT          NOT NULL REFERENCES Terms(TermsID),
    InvoiceDueDate DATE         NOT NULL,
    PaymentDate    DATE         NULL
);
```

A statement that adds a new column to the table

```
ALTER TABLE Invoices
ADD BalanceDue MONEY NOT NULL;
```

A statement that deletes the new column

```
ALTER TABLE Invoices
DROP COLUMN BalanceDue;
```

A statement that creates an index on the table

```
CREATE INDEX IX_Invoices_VendorID
    ON Invoices (VendorID);
```

Description

- The REFERENCES clause for a column indicates that the column contains a foreign key, and it names the table and column that contains the primary key. Because the Invoices table includes foreign keys to the Vendors and Terms tables, these tables must be created before the Invoices table.
- Because default values are specified for the PaymentTotal and CreditTotal columns, these values don't need to be specified when a row is added to the table.
- Because the PaymentDate column accepts nulls, a null value is assumed if a value isn't specified for this column when a row is added to the table.

How to query a single table

Figure 1-12 shows how to use a SELECT statement to query a single table in a database. At the top of this figure, you can see some of the columns and rows of the Invoices table. Then, in the SELECT statement that follows, the SELECT clause names the columns to be retrieved, and the FROM clause names the table that contains the columns, called the *base table*. In this case, six columns will be retrieved from the Invoices table.

Notice that the last column, BalanceDue, is calculated from three other columns in the table. In other words, a column by the name of BalanceDue doesn't actually exist in the database. This type of column is called a *calculated value*, and it exists only in the results of the query.

In addition to the SELECT and FROM clauses, this SELECT statement includes a WHERE clause and an ORDER BY clause. The WHERE clause gives the criteria for the rows to be selected. In this case, a row is selected only if it has a balance due that's greater than zero. And the returned rows are sorted by the InvoiceDate column.

This figure also shows the *result table*, or *result set*, that's returned by the SELECT statement. A result set is a logical table that's created temporarily within the database. When an application requests data from a database, it receives a result set.

The Invoices base table

	InvoiceID	VendorID	InvoiceNumber	InvoiceDate	InvoiceTotal	PaymentTotal	CreditTotal	TermSID
1	1	122	989319-457	2019-10-08	3813.33	3813.33	0.00	3
2	2	123	263253241	2019-10-10	40.20	40.20	0.00	3
3	3	123	963253234	2019-10-13	138.75	138.75	0.00	3
4	4	123	2-000-2993	2019-10-16	144.70	144.70	0.00	3
5	5	123	963253251	2019-10-16	15.50	15.50	0.00	3
6	6	123	963253261	2019-10-16	42.75	42.75	0.00	3
7	7	123	963253237	2019-10-21	172.50	172.50	0.00	3
8	8	89	125520-1	2019-10-24	95.00	95.00	0.00	1
9	9	121	97/488	2019-10-24	601.95	601.95	0.00	3
10	10	123	263253250	2019-10-24	42.67	42.67	0.00	3
11	11	123	963253262	2019-10-25	42.50	42.50	0.00	3
12	12	96	I77271-001	2019-10-26	662.00	662.00	0.00	2
13	13	95	111-92R-10096	2019-10-30	16.33	16.33	0.00	2
14	14	115	25022117	2019-11-01	6.00	6.00	0.00	4
15	15	48	P02-88D77S7	2019-11-03	856.92	856.92	0.00	3

A SELECT statement that retrieves and sorts selected columns and rows from the Invoices table

```

SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
       PaymentTotal, CreditTotal,
       InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue
  FROM Invoices
 WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0
 ORDER BY InvoiceDate;

```

The result set defined by the SELECT statement

	InvoiceNumber	InvoiceDate	InvoiceTotal	PaymentTotal	CreditTotal	BalanceDue
1	39104	2020-01-10	85.31	0.00	0.00	85.31
2	963253264	2020-01-18	52.25	0.00	0.00	52.25
3	31361833	2020-01-21	579.42	0.00	0.00	579.42
4	263253268	2020-01-21	59.97	0.00	0.00	59.97
5	263253270	2020-01-22	67.92	0.00	0.00	67.92
6	263253273	2020-01-22	30.75	0.00	0.00	30.75

Concepts

- You use the SELECT statement to retrieve selected columns and rows from a *base table*. The result of a SELECT statement is a *result table*, or *result set*, like the one shown above.
- A result set can include *calculated values* that are calculated from columns in the table.
- The execution of a SELECT statement is commonly referred to as a *query*.

Figure 1-12 How to query a single table

How to join data from two or more tables

Figure 1-13 presents a SELECT statement that retrieves data from two tables. This type of operation is called a *join* because the data from the two tables is joined together into a single result set. For example, the SELECT statement in this figure joins data from the Invoices and Vendors tables.

An *inner join* is the most common type of join. When you use an inner join, rows from the two tables in the join are included in the result table only if their related columns match. These matching columns are specified in the FROM clause of the SELECT statement. In the SELECT statement in this figure, for example, rows from the Invoices and Vendors tables are included only if the value of the VendorID column in the Vendors table matches the value of the VendorID column in one or more rows in the Invoices table. If there aren't any invoices for a particular vendor, that vendor won't be included in the result set.

Although this figure shows only how to join data from two tables, you should know that you can extend this idea to join data from three or more tables. If, for example, you want to include line item data from a table named InvoiceLineItems in the results shown in this figure, you can code the FROM clause of the SELECT statement like this:

```
FROM Vendors
    INNER JOIN Invoices
        ON Vendors.VendorID = Invoices.VendorID
    INNER JOIN InvoiceLineItems
        ON Invoices.InvoiceID = InvoiceLineItems.InvoiceID
```

Then, in the SELECT clause, you can include any of the columns in the InvoiceLineItems table.

In addition to inner joins, SQL Server supports *outer joins* and *cross joins*. You'll learn more about the different types of joins in chapter 4.

A SELECT statement that joins data from the Vendors and Invoices tables

```
SELECT VendorName, InvoiceNumber, InvoiceDate, InvoiceTotal  
FROM Vendors INNER JOIN Invoices  
    ON Vendors.VendorID = Invoices.VendorID  
WHERE InvoiceTotal >= 500  
ORDER BY VendorName, InvoiceTotal DESC;
```

The result set defined by the SELECT statement

	VendorName	InvoiceNumber	InvoiceDate	InvoiceTotal
1	Bertelsmann Industry Svcs. Inc	509786	2019-12-18	6940.25
2	Cahners Publishing Company	587056	2019-12-28	2184.50
3	Computerworld	367447	2019-12-11	2433.00
4	Data Reproductions Corp	40318	2019-12-01	21842.00
5	Dean Witter Reynolds	75C-90227	2019-12-11	1367.50
6	Digital Dreamworks	P02-3772	2019-11-21	7125.34
7	Federal Express Corporation	963253230	2020-01-07	739.20
8	Ford Motor Credit Company	9982771	2020-01-24	503.20
9	Franchise Tax Board	RTR-72-366...	2019-11-25	1600.00
10	Fresno County Tax Collector	P02-88D77S7	2019-11-03	856.92
11	IBM	Q545443	2019-12-09	1083.58
12	Ingram	31359783	2019-12-03	1575.00
13	Ingram	31361833	2020-01-21	579.42
14	Malloy Lithographing Inc	0-2058	2019-11-28	37966.19
15	Malloy Lithographing Inc	P-0259	2020-01-19	26881.40
16	Malloy Lithographing Inc	0-2060	2020-01-24	23517.58
17	Malloy Lithographing Inc	P-0608	2020-01-23	20551.18

Concepts

- A *join* lets you combine data from two or more tables into a single result set.
- The most common type of join is an *inner join*. This type of join returns rows from both tables only if their related columns match.
- An *outer join* returns rows from one table in the join even if the other table doesn't contain a matching row.

Figure 1-13 How to join data from two or more tables

How to add, update, and delete data in a table

Figure 1-14 shows how you can use the INSERT, UPDATE, and DELETE statements to modify the data in a table. The first statement in this figure, for example, uses the INSERT statement to add a row to the Invoices table. To do that, the INSERT clause names the columns whose values are supplied in the VALUES clause. You'll learn more about specifying column names and values in chapter 7. For now, just realize that you have to specify a value for a column unless it's an identity column, a column that allows null values, or a column that's defined with a default value.

The two UPDATE statements in this figure illustrate how you can change the data in one or more rows of a table. The first statement, for example, assigns a value of 35.89 to the CreditTotal column of the invoice in the Invoices table with invoice number 367447. The second statement adds 30 days to the invoice due date for each row in the Invoices table whose TermsID column has a value of 4.

To delete rows from a table, you use the DELETE statement. The first DELETE statement in this figure, for example, deletes the invoice with invoice number 4-342-8069 from the Invoices table. The second DELETE statement deletes all invoices with a balance due of zero.

Before I go on, you should know that INSERT, UPDATE, and DELETE statements are often referred to as *action queries* because they perform an action on the database. By contrast, SELECT statements are referred to as *queries* since they simply query the database. When I use the term *query* in this book, then, I'm usually referring to a SELECT statement.

A statement that adds a row to the Invoices table

```
INSERT INTO Invoices (VendorID, InvoiceNumber, InvoiceDate,  
InvoiceTotal, TermsID, InvoiceDueDate)  
VALUES (12, '3289175', '2/18/2020', 165, 3, '3/18/2020');
```

**A statement that changes the value of the CreditTotal column
for a selected row in the Invoices table**

```
UPDATE Invoices  
SET CreditTotal = 35.89  
WHERE InvoiceNumber = '367447';
```

**A statement that changes the values in the InvoiceDueDate column
for all invoices with the specified TermsID**

```
UPDATE Invoices  
SET InvoiceDueDate = InvoiceDueDate + 30  
WHERE TermsID = 4;
```

A statement that deletes a selected invoice from the Invoices table

```
DELETE FROM Invoices  
WHERE InvoiceNumber = '4-342-8069';
```

A statement that deletes all paid invoices from the Invoices table

```
DELETE FROM Invoices  
WHERE InvoiceTotal - PaymentTotal - CreditTotal = 0;
```

Concepts

- You use the INSERT statement to add rows to a table.
- You use the UPDATE statement to change the values in one or more rows of a table based on the condition you specify.
- You use the DELETE statement to delete one or more rows from a table based on the condition you specify.
- The execution of an INSERT, UPDATE, or DELETE statement is often referred to as an *action query*.

Warning

- Until you read chapter 7 and understand the effect that these statements can have on the database, do not execute the statements shown above.

SQL coding guidelines

SQL is a freeform language. That means that you can include line breaks, spaces, and indentation without affecting the way the database interprets the code. In addition, SQL is not case-sensitive like some languages. That means that you can use uppercase or lowercase letters or a combination of the two without affecting the way the database interprets the code.

Although you can code SQL statements with a freeform style, we suggest that you follow the coding recommendations presented in figure 1-15. First, you should start each clause of a statement on a new line. In addition, you should continue long clauses onto multiple lines and you should indent the continued lines. You should also capitalize the first letter of each keyword in a statement to make them easier to identify, you should capitalize the first letter of each word in table and column names, and you should end each statement with a semicolon. Although the semicolon isn't currently required in most cases, it will be in a future version of SQL Server. So you should get used to coding it now. Finally, you should use *comments* to document code that's difficult to understand.

The examples at the top of this figure illustrate these coding recommendations. The first example presents an unformatted SELECT statement. As you can see, this statement is difficult to read. By contrast, this statement is much easier to read after our coding recommendations are applied, as you can see in the second example.

The third example illustrates how to code a *block comment*. This type of comment is typically coded at the beginning of a statement and is used to document the entire statement. Block comments can also be used within a statement to describe blocks of code, but that's not common.

The fourth example in this figure includes a *single-line comment*. This type of comment is typically used to document a single line of code. A single-line comment can be coded on a separate line as shown in this example, or it can be coded at the end of a line of code. In either case, the comment is delimited by the end of the line.

Although many programmers sprinkle their code with comments, that shouldn't be necessary if you write your code so it's easy to read and understand. Instead, you should use comments only to clarify portions of code that are hard to understand. Then, if you change the code, you should be sure to change the comments too. That way, the comments will always accurately represent what the code does.

A SELECT statement that's difficult to read

```
select invoicenumber, invoicedate, invoicetotal,  
    invoicetotal - paymenttotal - credittotal as balancedue  
from invoices where invoicetotal - paymenttotal -  
    credittotal > 0 order by invoicedate
```

A SELECT statement that's coded with a readable style

```
Select InvoiceNumber, InvoiceDate, InvoiceTotal,  
    InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue  
From Invoices  
Where InvoiceTotal - PaymentTotal - CreditTotal > 0  
Order By InvoiceDate;
```

A SELECT statement with a block comment

```
/*  
Author: Joel Murach  
Date: 1/22/2020  
*/  
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,  
    InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue  
FROM Invoices;
```

A SELECT statement with a single-line comment

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,  
    InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue  
    -- The fourth column calculates the balance due for each invoice  
FROM Invoices;
```

Coding recommendations

- Start each new clause on a new line.
- Break long clauses into multiple lines and indent continued lines.
- Capitalize the first letter of each keyword and each word in column and table names.
- End each statement with a semicolon (;).
- Use *comments* only for portions of code that are difficult to understand.

How to code a comment

- To code a *block comment*, type /* at the start of the block and */ at the end.
- To code a *single-line comment*, type -- followed by the comment.

Description

- Line breaks, white space, indentation, and capitalization have no effect on the operation of a statement.
- Comments can be used to document what a statement does or what specific parts of a statement do. They are not executed by the system.

Note

- Throughout this book, SQL keywords are capitalized so they're easier to identify. However, it's not necessary or customary to capitalize SQL keywords in your own code.

How to work with other database objects

In addition to the tables you've already learned about, relational databases can contain other objects. In the two topics that follow, you'll be introduced to four of those objects: views, stored procedures, triggers, and user-defined functions. Then, in section 4, you'll learn more about how to code and use these objects.

How to work with views

A *view* is a predefined query that's stored in a database. To create a view, you use the CREATE VIEW statement as shown in figure 1-16. This statement causes the SELECT statement you specify to be stored with the database. In this case, the CREATE VIEW statement creates a view named VendorsMin that retrieves three columns from the Vendors table.

Once you've created the view, you can refer to it instead of a table in most SQL statements. For this reason, a view is sometimes referred to as a *viewed table*. For example, the SELECT statement in this figure refers to the VendorsMin view rather than to the Vendors table. Notice that this SELECT statement makes use of the * operator, which causes all three of the columns defined by the view to be returned.

If you choose to, you can let a user query certain views but not query the tables on which the views are based. In this way, views can be used to restrict the columns and rows of a table that the user can see. In addition, you can simplify a user's access to one or more tables by coding complex SELECT queries as views.

A CREATE VIEW statement for a view named VendorsMin

```
CREATE VIEW VendorsMin AS
    SELECT VendorName, VendorState, VendorPhone
    FROM Vendors;
```

The virtual table that's represented by the view

	VendorName	VendorState	VendorPhone
1	US Postal Service	WI	(800) 555-1205
2	National Information Data Ctr	DC	(301) 555-8950
3	Register of Copyrights	DC	NULL
4	Jobtrak	CA	(800) 555-8725
5	Newbridge Book Clubs	NJ	(800) 555-9980
6	California Chamber Of Commerce	CA	(916) 555-6670
7	Towne Advertiser's Mailing Svcs	CA	NULL
8	BFI Industries	CA	(559) 555-1551
9	Pacific Gas & Electric	CA	(800) 555-6081
10	Robbins Mobile Lock And Key	CA	(559) 555-9375

A SELECT statement that uses the VendorsMin view

```
SELECT * FROM VendorsMin
WHERE VendorState = 'CA'
ORDER BY VendorName;
```

The result set that's returned by the SELECT statement

	VendorName	VendorState	VendorPhone
1	Abbey Office Furnishings	CA	(559) 555-8300
2	American Express	CA	(800) 555-3344
3	ASC Signs	CA	NULL
4	Aztek Label	CA	(714) 555-9000
5	Bertelsmann Industry Svcs. Inc	CA	(805) 555-0584
6	BFI Industries	CA	(559) 555-1551
7	Bill Jones	CA	NULL
8	Bill Marvin Electric Inc	CA	(559) 555-5106
9	Blanchard & Johnson Associates	CA	(214) 555-3647

Description

- A *view* consists of a SELECT statement that's stored with the database. Because views are stored as part of the database, they can be managed independently of the applications that use them.
- A view behaves like a virtual table. Since you can code a view name anywhere you'd code a table name, a view is sometimes called a *viewed table*.
- Views can be used to restrict the data that a user is allowed to access or to present data in a form that's easy for the user to understand. In some databases, users may be allowed to access data only through views.

Figure 1-16 How to work with views

How to work with stored procedures, triggers, and user-defined functions

A *stored procedure* is a set of one or more SQL statements that are stored together in a database. To create a stored procedure, you use the CREATE PROCEDURE statement as shown in figure 1-17. Here, the stored procedure contains a single SELECT statement. To use the stored procedure, you send a request for it to be executed. One way to do that is to use the Transact-SQL EXEC statement as shown in this figure. You can also execute a stored procedure from an application program by issuing the appropriate statement. How you do that depends on the programming language and the API you’re using to access the database.

When the server receives the request, it executes the stored procedure. If the stored procedure contains a SELECT statement like the one in this figure, the result set is sent back to the calling program. If the stored procedure contains INSERT, UPDATE, or DELETE statements, the appropriate processing is performed.

Notice that the stored procedure in this figure accepts an *input parameter* named @State from the calling program. The value of this parameter is then substituted for the parameter in the WHERE clause so that only vendors in the specified state are included in the result set. When it’s done with its processing, a stored procedure can also pass *output parameters* back to the calling program. In addition, stored procedures can include *control-of-flow language* that determines the processing that’s done based on specific conditions. You’ll learn more about how to code stored procedures in chapter 15.

A *trigger* is a special type of stored procedure that’s executed automatically when an insert, update, or delete operation is executed on a table or when a DDL statement is executed on a database. Triggers are used most often to validate data before a row is added or updated, but they can also be used to maintain the relationships between tables or to provide information about changes to the definition of a database.

A *user-defined function*, or *UDF*, is also a special type of procedure. After it performs its processing, a UDF can return a single value or an entire table to the calling program. You’ll learn how to code and use user-defined functions and triggers in chapter 15.

A CREATE PROCEDURE statement for a procedure named spVendorsByState

```
CREATE PROCEDURE spVendorsByState @StateVar char(2) AS
    SELECT VendorName, VendorState, VendorPhone
    FROM Vendors
    WHERE VendorState = @StateVar
    ORDER BY VendorName;
```

A statement that executes the spVendorsByState stored procedure

```
EXEC spVendorsByState 'CA';
```

The result set that's created when the stored procedure is executed

	VendorName	VendorState	VendorPhone
1	Abbey Office Furnishings	CA	(559) 555-8300
2	American Express	CA	(800) 555-3344
3	ASC Signs	CA	NULL
4	Aztek Label	CA	(714) 555-9000
5	Bertelsmann Industry Svcs. Inc	CA	(805) 555-0584
6	BFI Industries	CA	(559) 555-1551
7	Bill Jones	CA	NULL
8	Bill Marvin Electric Inc	CA	(559) 555-5106

Concepts

- A *stored procedure* is one or more SQL statements that have been compiled and stored with the database. A stored procedure can be started by application code on the client.
- Stored procedures can improve database performance because the SQL statements in each procedure are only compiled and optimized the first time they're executed. By contrast, SQL statements that are sent from a client to the server have to be compiled and optimized every time they're executed.
- In addition to SELECT statements, a stored procedure can contain other SQL statements such as INSERT, UPDATE, and DELETE. It can also contain *control-of-flow language*, which lets you perform conditional processing within the stored procedure.
- A *trigger* is a special type of procedure that's executed when rows are inserted, updated, or deleted from a table or when the definition of a database is changed. Triggers are typically used to check the validity of the data in a row that's being updated or added to a table.
- A *user-defined function (UDF)* is a special type of procedure that can return a value or a table.

Figure 1-17 How to use stored procedures, triggers, and user-defined functions

How to use SQL from an application program

This book teaches you how to use SQL from within the SQL Server environment. However, SQL is commonly used from application programs too. So in the topics that follow, you'll get a general idea of how that works. And you'll see that it's easy to recognize the SQL statements in an application program because they're coded just as they would be if they were running on their own.

Common data access models

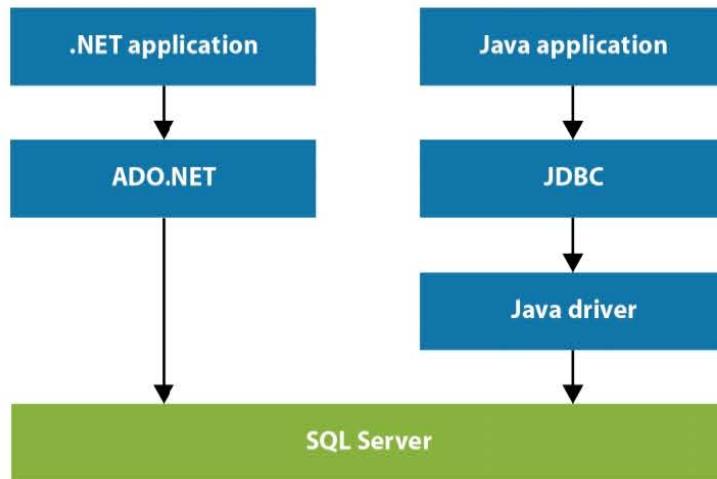
Figure 1-18 shows two common ways for an application to access a SQL Server database. First, you can access a SQL Server database from a .NET application written using a .NET language such as C# or Visual Basic. To do that, you can use *ADO.NET*. This is a *data access model* developed by Microsoft, and it can communicate directly with SQL Server.

Because ADO.NET uses a *disconnected data architecture*, its use has become widespread, particularly for web-based applications. That's because web-based applications by necessity work with *disconnected data*. That means that once an application has sent a response to the client, it doesn't maintain its connection to the database.

The second data access model in this figure is JDBC, which is used by Java applications. Unlike ADO.NET, JDBC requires additional software, called a *driver*, to communicate with SQL Server.

Although it's not shown here, another option for accessing SQL Server data that's becoming more and more popular is using an object-relational mapping (ORM) framework. When you use an ORM framework, the data in a relational database is mapped to the objects used by an object-oriented programming language such as C# or Java. Then, the application can make requests against the object model, and those requests are translated into ones that can be executed by the database. This simplifies the code that's needed to communicate with the database. Before you start developing an application that works with a database, then, you should consider using an ORM framework such as Entity Framework (EF) for .NET applications and Hibernate for Java applications.

Two common options for accessing SQL Server data



Description

- To work with the data in a SQL Server database, an application uses a *data access model*. For an application written in a .NET language such as C# or Visual Basic that model is typically *ADO.NET*. For an application written in Java, that model is typically *JDBC (Java Database Connectivity)*.
- Each data access model defines a set of objects you can use to connect to and work with a SQL Server database. For example, both of the models shown above include a connection object that you can use to specify the information for connecting to a database.
- Some of the data access models require additional software, called *drivers*, to communicate with SQL Server. For example, JDBC requires a Java driver.
- ADO.NET, a data access model developed by Microsoft, includes its own driver so it can communicate directly with SQL Server.
- In addition to working with ADO.NET and JDBC directly, you can use an object-relational mapping (ORM) framework. An ORM framework works by mapping the data in a relational database to the objects used by an object-oriented programming language. Then, the ORM translates requests to retrieve, insert, update, and delete data so the DBMS will understand them.
- The most popular ORM framework for use with .NET applications is Entity Framework (EF), which was developed by Microsoft. The most popular ORM framework for use with Java is Hibernate. Both of these frameworks are open-source.

Figure 1-18 Common data access models

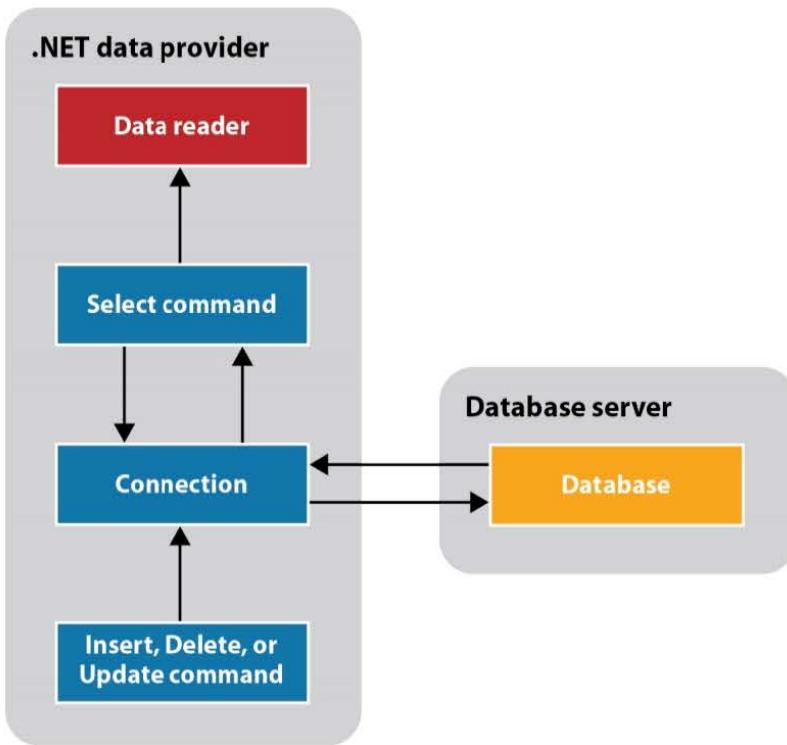
How to use ADO.NET from a .NET application

To illustrate how you use a data access model, figure 1-19 introduces you to the basic ADO.NET objects that you use in a .NET application. Then, in the next two figures, you'll see some actual code that creates and uses these objects. Keep in mind, though, that there's a lot more you need to know about ADO.NET than what's presented here.

When you develop a .NET application, you can choose from several languages, but the most popular are Visual Basic and C#. Although C# has become more popular than Visual Basic over the past few years, the language you choose is largely a matter of personal preference.

To access a database using the objects shown here, you execute *command* objects. Then, a *connection* object is used to connect to the database, perform the requested operation, and return the result. If you execute a command that contains a SELECT statement, the result is a result set that contains the rows you requested. Then, you can read the rows in the result set using a *data reader* object. If you execute a command that contains an INSERT, UPDATE, or DELETE statement, the result is a value that indicates if the operation was successful.

Basic ADO.NET objects in a .NET application



Description

- To work with the data in a SQL Server database from a .NET application, you can use ADO.NET objects like the ones shown above.
- A *.NET data provider* provides the classes that let you create the objects that you use to retrieve data from a database and to store data in a database.
- To retrieve data from a database, you execute a *command* object that contains a SELECT statement. Then, the command object uses a *connection* object to connect to the database and retrieve the data. You can then read the results one row at a time using a *data reader* object.
- To insert, update, or delete data in a database, you execute a command object that contains an INSERT, UPDATE, or DELETE statement. Then, the command object uses a connection to connect to the database and update the data. You can then check the value that's returned to determine if the operation was successful.
- After data is retrieved from a database or updated in a database, the connection is closed and the resources used by the connection are released. This is referred to as a *disconnected data architecture*.

Figure 1-19 How to use ADO.NET from a .NET application

Visual Basic code that retrieves data from a SQL Server database

Figure 1-20 presents a Visual Basic function that uses the ADO.NET objects shown in the previous figure. This function is from a simple application that accepts a vendor ID from the user, retrieves the information for the vendor with that ID from the Vendors table, and then displays that information. Although I don't expect you to understand this code, I hope it will give you a feel for how you use SQL from an application program.

This function starts by creating a new Vendor object. Although it's not shown here, this object contains properties that correspond to the columns in the Vendors table. Then, this function creates the connection object that will be used to connect to the database and sets the connection string for that object. The connection string provides ADO.NET with the information it needs to connect to the database.

Once the connection string is set, the next group of statements defines the Command object that will be executed to retrieve the data from the database. The first statement in this group creates the command object. Then, the next statement assigns the connection object to the command object. That means that when the statement that this object will contain is executed, it will use the connection string in the connection object to connect to the database.

The next statement in this group specifies the SELECT statement to be executed. If you review this statement, you'll see that the WHERE clause includes a parameter named @VendorID that will contain the value of the vendor ID. This value is set by the last statement in this group.

The next statement opens the connection to the database. Then, the next group of statements retrieves the vendor row and stores it in a Vendor object. To do that, it starts by executing the command to create a data reader. Then, if the vendor is found, it assigns the values of the columns in the row to the properties of the Vendor object. Otherwise, the Vendor object is set to Nothing.

After all of the rows are processed, the data reader and connection are closed. Then, the Vendor object is returned to the calling procedure.

Now that you've reviewed this code, you can see that there's a lot involved in accessing a SQL Server database from an application program. However, you can also see that only one statement in this figure actually involves using SQL. That's the statement that specifies the SELECT statement to be executed. Of course, if the program also provided for updating the data in the Vendors table, it would include INSERT, UPDATE, and DELETE statements. With the skills that you'll learn in this book, though, you won't have any trouble coding the SQL statements you need for your applications.

A Visual Basic function that uses ADO.NET objects to retrieve data from a SQL Server database

```
Public Shared Function GetVendor(vendorID As Integer) As Vendor
    Dim vendor As New Vendor

    ' Create the connection object
    Dim connection As New SqlConnection()
    connection.ConnectionString = "Data Source=localhost\SqlExpress;" &
        "Initial Catalog=AP;Integrated Security=True"

    ' Create the command object and set the connection,
    ' SELECT statement, and parameter value
    Dim selectCommand As New SqlCommand
    selectCommand.Connection = connection
    selectCommand.CommandText = "SELECT VendorID, " &
        "VendorName, VendorAddress1, VendorAddress2, " &
        "VendorCity, VendorState, VendorZipCode " &
        "FROM Vendors WHERE VendorID = @VendorID"
    selectCommand.Parameters.AddWithValue("@VendorID", vendorID)

    ' Open the connection to the database
    connection.Open()

    ' Retrieve the row specified by the SELECT statement
    ' and load it into the Vendor object
    Dim reader As SqlDataReader = selectCommand.ExecuteReader
    If reader.Read Then
        vendor.VendorID = CInt(reader("VendorID"))
        vendor.VendorName = reader("VendorName").ToString
        vendor.VendorAddress1 = reader("VendorAddress1").ToString
        vendor.VendorAddress2 = reader("VendorAddress2").ToString
        vendor.VendorCity = reader("VendorCity").ToString
        vendor.VendorState = reader("VendorState").ToString
        vendor.VendorZipCode = reader("VendorZipCode").ToString
    Else
        vendor = Nothing
    End If
    reader.Close()

    ' Close the connection to the database
    connection.Close()

    Return vendor
End Function
```

Description

- To issue a SQL statement from a Visual Basic program, you can create ADO.NET objects like the ones shown above.
- After you create the ADO.NET objects, you have to set the properties of those objects that define how they work. For example, the ConnectionString property of a connection object contains the information ADO.NET needs to connect to a database.

Figure 1-20 Visual Basic code that retrieves data from a SQL Server database

C# code that retrieves data from a SQL Server database

Figure 1-21 presents C# code that uses the ADO.NET objects to retrieve data from a SQL Server database. This code provides the same functionality as the Visual Basic code presented in figure 1-20. If you compare the code presented in these two figures, you'll see that both Visual Basic and C# use the same ADO.NET objects that are provided as part of the .NET Framework.

The main difference is that the C# language uses a different syntax than Visual Basic. This syntax is similar to the syntax that's used by C++ and Java. As a result, if you already know C++ or Java, it should be relatively easy for you to learn C#. Conversely, once you learn C#, it's easier to learn C++ or Java.

A C# method that uses ADO.NET objects to retrieve data from a SQL Server database

```
public static Vendor GetVendor(int vendorID)
{
    Vendor vendor = new Vendor();

    // Create the connection object
    SqlConnection connection = new SqlConnection();
    connection.ConnectionString = "Data Source=localhost\\SqlExpress;" +
        "Initial Catalog=AP;Integrated Security=True";

    // Create the command object and set the connection,
    // SELECT statement, and parameter value
    SqlCommand selectCommand = new SqlCommand();
    selectCommand.Connection = connection;
    selectCommand.CommandText = "SELECT VendorID, " +
        "VendorName, VendorAddress1, VendorAddress2, " +
        "VendorCity, VendorState, VendorZipCode " +
        "FROM Vendors WHERE VendorID = @VendorID";
    selectCommand.Parameters.AddWithValue("@VendorID", vendorID);

    // Open the connection to the database
    connection.Open();

    // Retrieve the row specified by the SELECT statement
    // and load it into the Vendor object
    SqlDataReader reader = selectCommand.ExecuteReader();
    if (reader.Read())
    {
        vendor.VendorID = (int)reader["VendorID"];
        vendor.VendorName = reader["VendorName"].ToString();
        vendor.VendorAddress1 = reader["VendorAddress1"].ToString();
        vendor.VendorAddress2 = reader["VendorAddress2"].ToString();
        vendor.VendorCity = reader["VendorCity"].ToString();
        vendor.VendorState = reader["VendorState"].ToString();
        vendor.VendorZipCode = reader["VendorZipCode"].ToString();
    }
    else
    {
        vendor = null;
    }
    reader.Close();

    // Close the connection to the database
    connection.Close();

    return vendor;
}
```

Description

- To issue a SQL statement from a C# application, you can use ADO.NET objects like the ones shown above.

Figure 1-21 C# code that retrieves data from a SQL Server database

Perspective

To help you understand how SQL is used from an application program, this chapter has introduced you to the hardware and software components of a client/server system. It has also described how relational databases are organized and how you use some of the basic SQL statements to work with the data in a relational database. With that as background, you're ready to start using SQL Server. In the next chapter, then, you'll learn how to use some of the tools provided by SQL Server.

Terms

client	primary key
server	composite primary key
database server	non-primary key
network	unique key
client/server system	index
local area network (LAN)	foreign key
enterprise system	one-to-many relationship
wide area network (WAN)	one-to-one relationship
network operating system	many-to-many relationship
database management system (DBMS)	data type
back end	null value
application software	default value
data access API (application programming interface)	identity column
front end	hierarchical database
SQL (Structured Query Language)	parent/child relationship
SQL query	network database
query results	object database
application server	relational database management system (RDBMS)
web server	Oracle
business component	DB2 (Database 2)
web application	ANSI (American National Standards Institute)
web service	levels of conformance
web browser	core specification
thin client	package
relational database	SQL dialect
table	extension
row	SQL variant
column	Transact-SQL
record	open-source database
field	data manipulation language (DML)
cell	data definition language (DDL)

database administrator (DBA)	stored procedure
base table	input parameter
result table	output parameter
result set	control-of-flow language
calculated value	user-defined function (UDF)
join	trigger
inner join	data access model
outer join	ADO.NET
cross join	JDBC (Java Database Connectivity)
action query	disconnected data architecture
comment	driver
block comment	command
single-line comment	connection
view	data source
viewed table	

How to use the Management Studio

In the last chapter, you learned about some of the SQL statements you can use to work with the data in a relational database. Before you learn the details of coding these statements, however, you need to become familiar with a tool that you can use to execute these statements against a relational database. Since this book is about SQL Server 2019, this chapter will teach you about the primary tool for working with SQL Server 2019, the SQL Server Management Studio.

An introduction to SQL Server 2019.....	50
A summary of the SQL Server 2019 tools	50
How to start and stop the database engine	52
How to enable remote connections.....	52
An introductionto the Management Studio	54
How to connect to a database server	54
How to navigate through the database objects	56
How to manage the database files	58
How to attach a database	58
How to detach a database	58
How to back up a database	60
How to restore a database.....	60
How to set the compatibility level for a database.....	62
How to view and modify the database	64
How to create database diagrams	64
How to view the column definitions of a table	66
How to modify the column definitions.....	66
How to view the data of a table	68
How to modify the data of a table	68
How to work with queries	70
How to enter and execute a query.....	70
How to handle syntax errors.....	72
How to open and save queries	74
An introduction to the Query Designer.....	76
How to view the documentationfor SQL Server	78
How to display the SQL Server documentation	78
How to look up information in the documentation	78
Perspective	80

An introduction to SQL Server 2019

The current version of Microsoft SQL Server, SQL Server 2019, is a complete database management system. It consists of a *database server* that provides the services for managing SQL Server databases and *client tools* that provide an interface for working with the databases. Of these client tools, the Management Studio is the primary tool for working with a database server.

Before I go on, you should know that you can follow along with the skills presented in this chapter if you have access to SQL Server 2019. If that's not the case, you can refer to appendix A of this book to learn how to download and install it. In addition, you can refer to appendix A to download all of the database and source code files used in this book. Once you do that, you can work along with the book examples.

If you install SQL Server as described in appendix A of this book, a free edition of SQL Server, called the SQL Server 2019 Express, will be installed on your machine. Although the Express Edition restricts the number of processors, the amount of memory, and the amount of data that SQL Server can manage, it provides a realistic testing environment that is 100% compatible with the other versions of SQL Server 2019. In fact, SQL Server Express is adequate for many small and medium sized applications. And since it's free and easy to use, it's perfect for learning about SQL Server. For example, I used SQL Server Express to create and test all of the statements presented in this book.

Note, however, that SQL Server Express is strictly a database server, or *database engine*. In other words, it doesn't provide some of the client tools you may need. In particular, it doesn't include the Management Studio. That's why appendix A describes how to install the Management Studio, which is also available for free.

A summary of the SQL Server 2019 tools

Figure 2-1 summarizes the SQL Server 2019 client tools that you'll learn how to use in this book: the Management Studio and the Configuration Manager. Although other tools exist for working with a SQL Server 2019 database, they are commonly used by database administrators and other specialists, not application developers. That's why they aren't presented in this book.

A summary of the SQL Server 2019 tools

Tool	Description
SQL Server Management Studio	The primary graphical tool that a developer uses to work with a SQL Server 2019 database. You can use this tool to work directly with database objects and to develop and test SQL statements.
SQL Server Configuration Manager	A graphical tool that you can use to start and stop the database server.

Description

- To work with a SQL Server database and the data it contains, you can use the SQL Server 2019 tools described above.
- After you install SQL Server 2019, you can access the Management Studio from the Start→Microsoft SQL Server Tools 18 program group, and you can access the Configuration Manager from the Start→Microsoft SQL Server 2019 program group. To start one of these tools, just select it from its group.
- To make either of these tools easier to access, you can pin it to the Start menu by right-clicking on it and selecting Pin to Start. Then, it will appear as a tile in the Start menu and you can click on it to start it.

How to start and stop the database engine

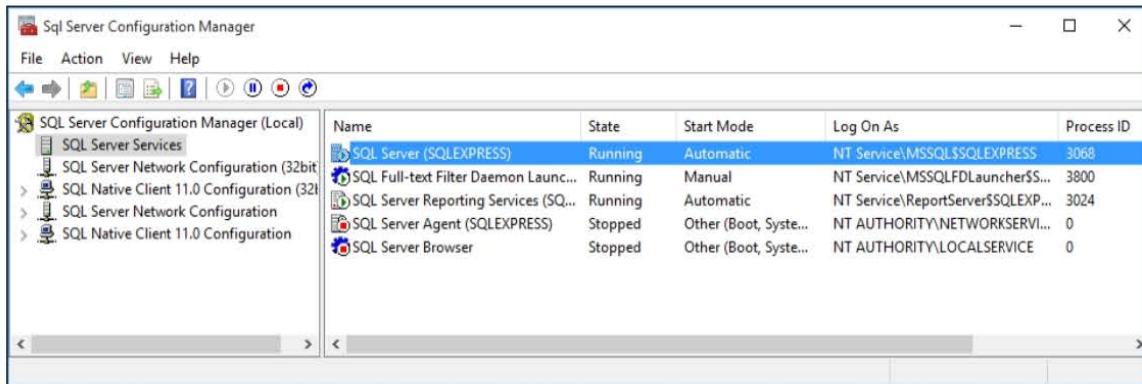
If you've installed SQL Server Express on your own system, you can use the SQL Server Configuration Manager to start and stop the database engine as described in figure 2-2. By default, the database engine starts automatically when the operating system starts, which is usually what you want. However, you may occasionally need to stop and then restart the engine. For example, some changes you make to the database server won't go into effect until you restart the engine.

By the way, if you simply want to find out if the database engine is running, you can do that by selecting SQL Server 2019 Services in the left pane. Then, you can look at the State column in the right pane. In this figure, for example, the Configuration Manager shows that the SQL Server Express database engine is running.

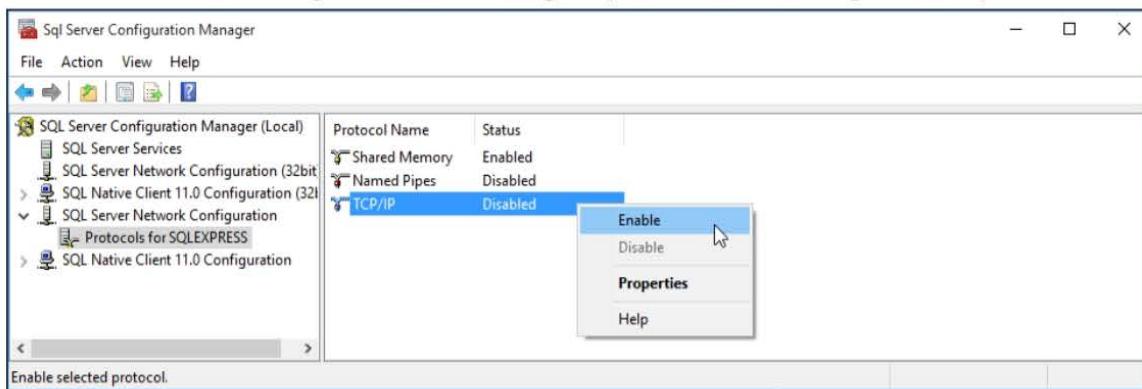
How to enable remote connections

When you install SQL Server 2019, remote connections are disabled by default. This is a security precaution that prevents other computers from connecting to this instance of SQL Server before it has been properly secured. As a result, if you have installed SQL Server 2019 and you want to allow other computers to be able to access this instance of SQL Server, you must enable remote connections. To do that, you can use the SQL Server Configuration Manager tool as described in figure 2-2. Of course, if databases that contain sensitive data are running under this instance of SQL Server, you'll want to secure the database properly before you enable remote connections.

The SQL Server Configuration Manager (Services)



The SQL Server Configuration Manager (Network Configuration)



Description

- After you install SQL Server Express, the database server will start automatically each time you start your PC by default.
- To display the Configuration Manager, select Start→Microsoft SQL Server 2019→SQL Server 2019 Configuration Manager.
- To start or stop a service, select the service in the right pane, and use the buttons in the toolbar to start or stop the service.
- To change the start mode for a service, right-click on the service in the right pane, select the Properties command to display the properties for the service, select the Service tab, and select the start mode you want from the Start Mode combo box.
- By default, remote connections are disabled for SQL Server 2019. To enable them, expand the SQL Server Network Configuration node and select the Protocols node for the server. Then, right-click on the protocol you want to enable and select the Enable command.

Figure 2-2 How to work with the database server

An introduction to the Management Studio

Once the SQL Server database engine is installed, configured, and running, you can use the Management Studio to connect to an instance of the SQL Server database engine. Then, you can use the Management Studio to work with the SQL Server database engine as described throughout this chapter.

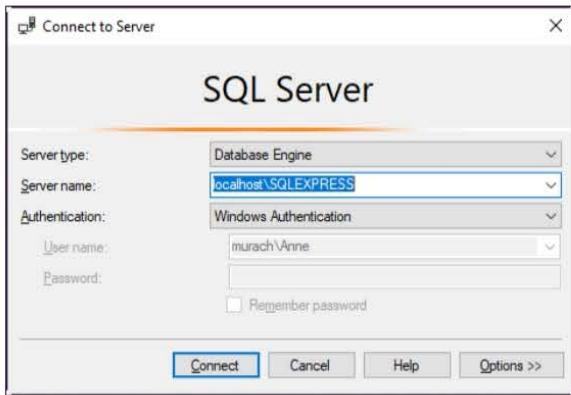
How to connect to a database server

When you start the Management Studio, a dialog box like the one in figure 2-3 is displayed. This dialog box lets you select the instance of SQL Server you want to connect to, and it lets you enter the required connection information.

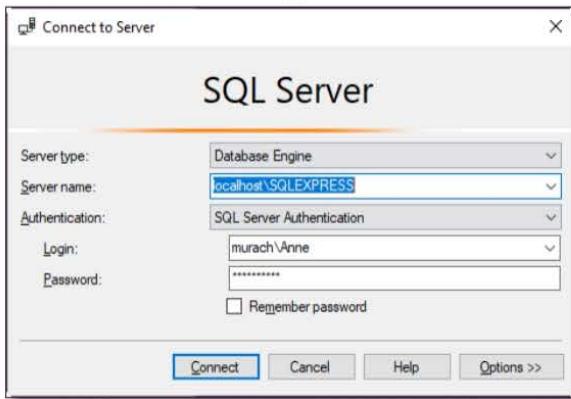
As you can see in this figure, you can use one of two types of authentication to connect to a server. In most cases, you can select the Windows Authentication option to let Windows supply the appropriate login name and password for you.

However, if you aren't able to use Windows Authentication, you can use SQL Server authentication. For example, you may need to use SQL Server authentication if you're accessing your school's or company's server. In that case, you can contact the database administrator to get an appropriate SQL Server login name and password. For more information about both types of authentication, please refer to chapter 17.

How to connect using Windows authentication



How to connect using SQL Server authentication



Description

- When you start the Management Studio, it displays a dialog box that allows you to specify the information that's needed to connect to the appropriate database server.
- To connect to a database server, you use the Server Name combo box to enter or select a path that specifies the database server. You begin by entering the name of the computer, followed by a backslash, followed by the name of the SQL Server database server.
- To connect to the SQL Server Express database engine when it's running on your PC, you can use the localhost keyword to specify your computer as the host machine, and you can use "SqlExpress" to specify SQL Server Express as the database engine.
- If you select Windows authentication, SQL Server will use the login name and password that you use for your computer to verify that you are authorized to connect to the database server.
- If you select SQL Server authentication, you'll need to enter an appropriate login name and password. This type of authentication is typically used only with non-Windows clients.

Figure 2-3 How to connect to a database server

How to navigate through the database objects

Figure 2-4 shows how to use the Management Studio to navigate through the database objects that are available from the current database server. By default, the Object Explorer window is displayed on the left side of the Management Studio window. If it isn't displayed, you can use the View menu to display it.

This window displays the instance of SQL Server that the Management Studio is connected to, all of the databases that are attached to this instance of SQL Server, and all objects within each database. These objects include tables, columns, keys, constraints, triggers, indexes, views, stored procedures, functions, and so on.

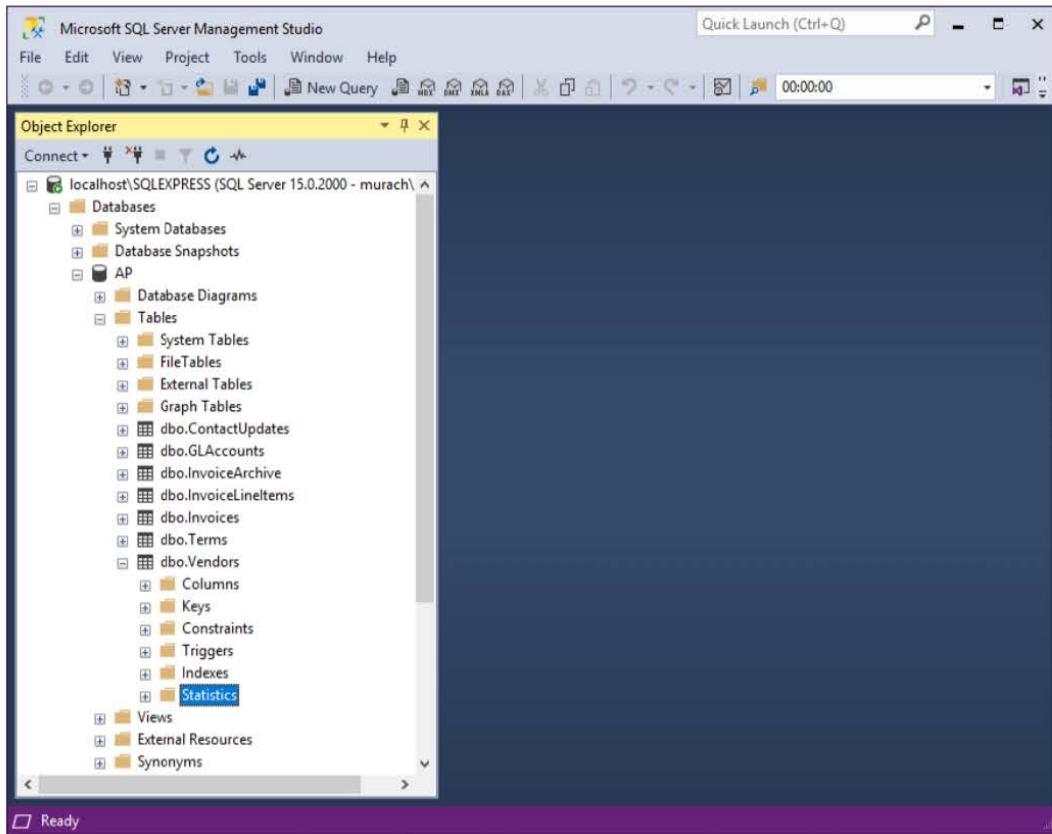
To navigate through the database objects displayed in the Object Explorer, you can click on the plus (+) and minus (-) signs to the left of each node to expand or collapse the node. In this figure, for example, I expanded the Databases node. That way, all databases on the server are shown. Then, I expanded the node for the database named AP to browse through all of the objects for this database, and I expanded the Tables node for the AP database to view all of the tables for this database. Finally, I expanded the Vendors table node to show the types of database objects that are available for a table.

To work with a specific object, you can right-click on it to display a shortcut menu. To view or modify the design of a table, for example, you can right-click on the table and select the Design command. You'll learn how to use this command later in this chapter.

When you're working with the Management Studio, you may occasionally want to free up more space for the pane that's displayed to the right of the Object Explorer. To do that, you can click on the AutoHide button that's displayed in the top right of the Object Explorer. This button looks like a pushpin, and it automatically hides the Object Explorer when you click on it. Then, a tab for the Object Explorer is displayed on the left side of the Management Studio, and you can display the window by pointing to this tab. You can turn off the AutoHide feature by displaying the window and clicking on the AutoHide button again.

Before I go on, I want to point out the qualifier that's used on all of the table names in this figure: dbo. This qualifier indicates the schema that the tables belong to. In SQL Server, a *schema* is a container that holds objects. If you don't specify a schema when you create an object, it's stored in the default schema, dbo. As you'll learn in chapter 17, you can use schemas to make it easier to implement the security for a database. For now, you can assume that all the objects you work with are stored in the dbo schema.

The SQL Server Management Studio



Description

- The Management Studio is a graphical tool that you can use to work with the objects in a SQL Server database.
- If the Object Explorer isn't displayed, you can use the View menu to display it. You can close this window by clicking the Close button at the top of the window.
- To navigate through the database objects displayed in the Object Explorer, click on the plus (+) and minus (-) signs to the left of each node to expand or collapse the node.
- To display a menu of commands for working with an object, right-click on the object.
- If you want to automatically hide the Object Explorer, you can click on the AutoHide button at the top of the Object Explorer. Then, you can display the window by pointing to the Object Explorer tab that's displayed along the left side of the Management Studio window.

Figure 2-4 How to navigate through the database objects

How to manage the database files

Before you can work with the objects that are stored within a database, you need to create the database and its objects. If you have the files for an existing SQL Server database, the easiest way to create the database is to *attach* those files to the database server.

How to attach a database

Figure 2-5 shows how to use the Management Studio to *attach* the database files for a SQL Server database to an instance of the server. A SQL Server database consists of two types of files. The first file is the main data file, and it has an extension of mdf. The second file is the log file, and it has an extension of ldf.

If you have the data file for a database, the easiest way to attach the database is to use the existing data file. To do that, you right-click on the Databases folder and select the Attach Database command to display the Attach Databases dialog box. Then, you can click on the Add button and use the resulting dialog box to select the mdf file for the database. This should add both the data file and the log file for the database to the Database Details pane at the bottom of the dialog box. In this figure, for example, the Attach Databases dialog box shows both the mdf and ldf files for the database named AP. Finally, click OK to attach the database.

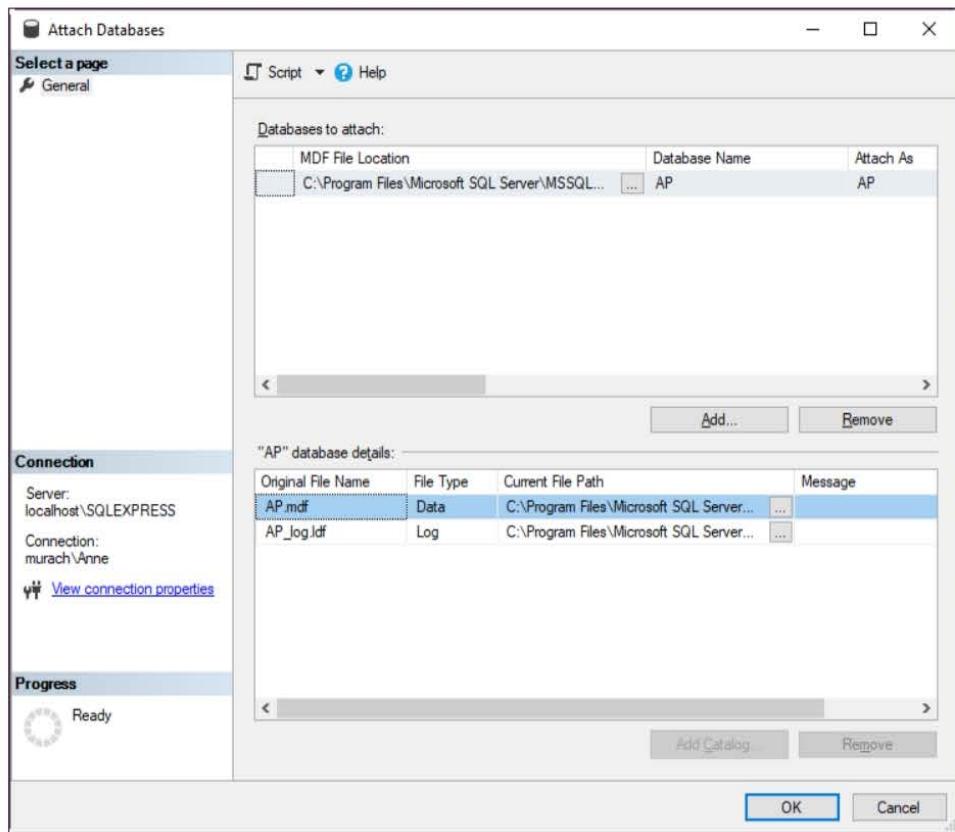
If you want to attach a database that doesn't have a log file, or if you want to create a new log file for a database, you can remove the log file for the database before you click the OK button. To do that, select the log file in the Database Details pane and click the Remove button. Then, when you click the OK button, the Management Studio will create a new log file for the database.

Before you attach database files, you need to decide where you'd like to store them. In most cases, you'll store them in the default directory shown in this figure. This is the directory where SQL Server 2019 stores the database files for databases you create from scratch. For example, when you run the script that creates the databases for this book as described in appendix A, SQL Server will store the database files in this directory. If you want to store the files for a database in a different location, though, you can do that too. You just need to remember where you store them.

How to detach a database

After you attach a database file, you will sometimes need to *detach* it. If, for example, you try to move a database file that's attached to a server, you'll get an error message that indicates that the file is in use. To get around this, you can detach the database file as described in figure 2-5. Then, you can move the database file and reattach it to the server later.

The Attach Databases dialog box



The default directory for SQL Server 2019 databases

C:\Program Files\Microsoft SQL Server\MSSQL15.SQLEXPRESS\MSSQL\DATA

Description

- To attach a database, right-click on the Databases folder and select the Attach command to display the Attach Databases dialog box shown above. Then, click on the Add button and use the resulting dialog box to select the mdf file for the database. This should add both the data file and the log file for the database to the Database Details pane at the bottom of the dialog box. Finally, click OK to attach the database.
- If you want to attach a database that doesn't have a log file, or if you want to create a new log file for a database, you can remove the log file for the database before you click the OK button. To do that, select the log file in the Database Details pane and click the Remove button.
- To detach a database, right-click on its icon and select the Tasks→Detach command to display the Detach Database dialog box. If the Message column indicates that there are active connections, you can usually select the Drop option to drop the connections. Then, click on the OK button.

Figure 2-5 How to attach or detach a database

How to back up a database

Whenever you're working with a database, and especially before you begin experimenting with new features, it's a good idea to *back up* the database as shown in figure 2-6. Then, if you accidentally modify or delete data, you can easily restore it.

By default, the Management Studio creates a full backup of the database and it stores the file for this database in the Backup directory shown in this figure. The file for the backup is the name of the database with an extension of bak. In this figure, for example, the backup file for the AP database is named AP.bak.

By default, the backup is set to expire in zero days, which means that the backup file will be saved on disk until the backup is run again. Then, the old backup file will be replaced by the new backup.

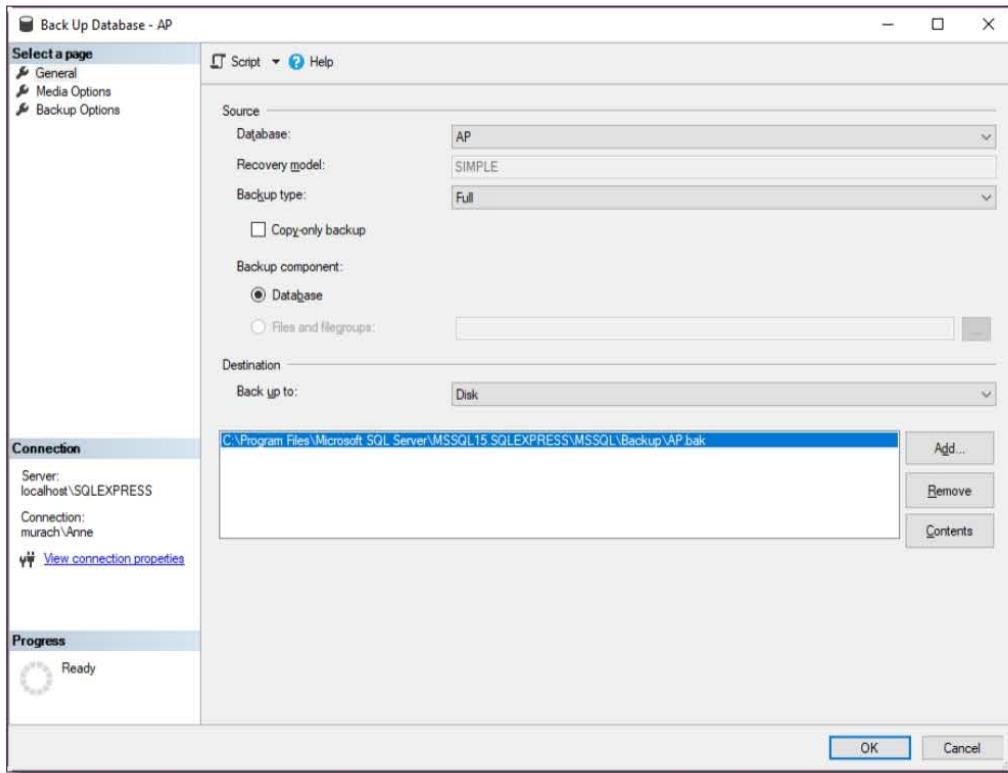
For the purposes of this book, those settings are usually adequate. However, if they aren't, you can use an incremental backup, or you can set the number of expiration days for the backup.

How to restore a database

If you need to *restore* a database from the backup copy, you can use the procedure described in figure 2-6. Although the Restore Database dialog box isn't shown in this figure, you shouldn't have any trouble using it.

By default, the Restore Database dialog box restores the current database to the most recent backup of the database, which is usually what you want. However, if you want to restore the database to a specific point in time, you can use the Restore Database dialog box to specify a date and time. Then, when you click OK, SQL Server will use the log files to restore the database to the specific point in time.

The Back Up Database dialog box



The default directory for SQL Server 2019 database backups

C:\Program Files\Microsoft SQL Server\MSSQL15.SQLEXPRESS\MSSQL\Backup

Description

- To *back up* a database, right-click on the database and select the Tasks→Back Up command to display the Back Up Database dialog box. For the purposes of this book, the default settings are usually adequate for backing up the database. As a result, you can usually click OK to back up the database.
- To *restore* a database, right-click on the database and select the Tasks→Restore→Database command to display the Restore Database dialog box. Then, click OK to restore the database. This replaces the current database with the most recent backup of the database.

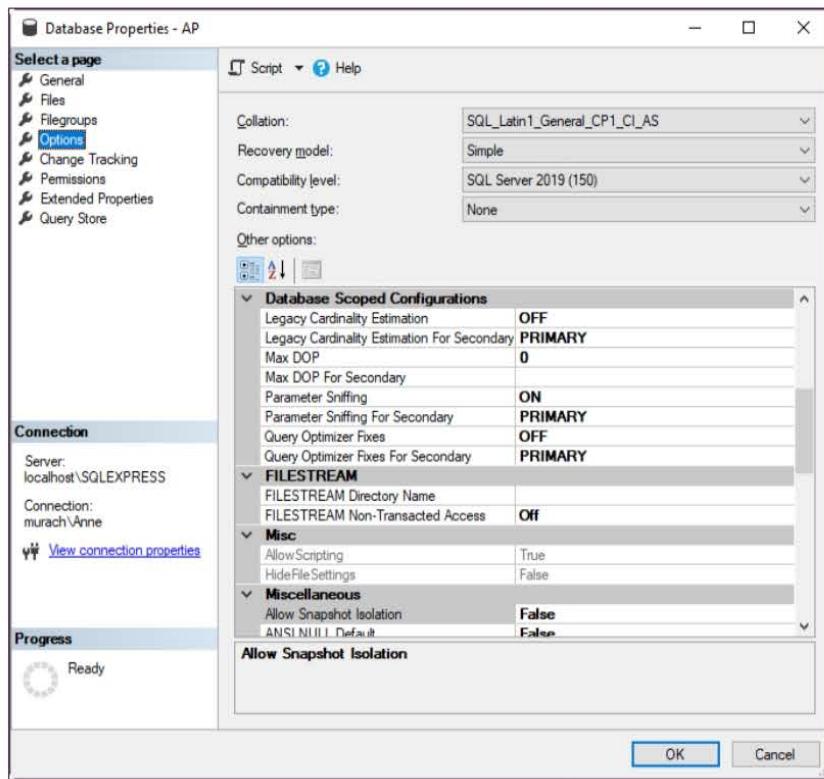
Figure 2-6 How to back up and restore a database

How to set the compatibility level for a database

The SQL Server 2019 database engine is backwards compatible and can run older versions of SQL Server databases as if they were running under an older version of the SQL Server database engine. As a result, after you attach a database, you may want to change the compatibility level for the database so it's appropriate for your purposes as described in figure 2-7.

For example, if you attach database files that were originally created under SQL Server 2016 to the SQL Server 2019 database engine, the compatibility level will remain set to SQL Server 2016. As a result, you will still be able to use most SQL Server 2016 features, even ones that have been deprecated, and you won't be able to use new SQL Server 2019 features. If that's what you want, you can leave the compatibility level set as it is. However, if you want to try using new SQL Server 2019 features with this database, you need to change the compatibility level to SQL Server 2019.

The Database Properties dialog box with the compatibility level options



Description

- The SQL Server 2019 database engine is backwards compatible and can run older versions of SQL Server databases just as if they were running under an older version of the SQL Server database engine.
- To set the compatibility level for a database to SQL Server 2019 , right-click the database, select the Properties command, click on the Options item, and select SQL Server 2019 from the Compatibility Level drop-down list.

Figure 2-7 How to set the compatibility level for a database

How to view and modify the database

Before you use SQL to query a database, you need to know how the database is defined. In particular, you need to know how the columns in each table are defined and how the tables are related. In addition, you may need to modify the database definition so it works the way you want.

How to create database diagrams

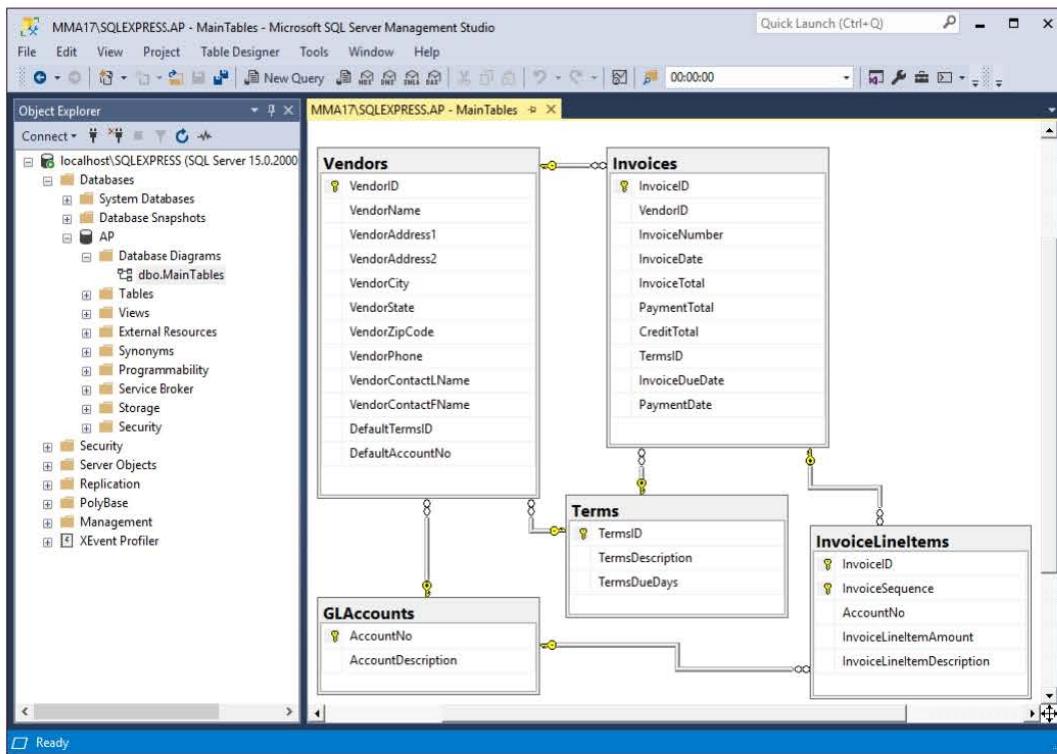
The easiest way to view the relationships between the tables in a database is to create a *database diagram* for the database as described in figure 2-8. In this figure, for example, the database diagram shows the relationships between five of the tables in the AP database. In addition, this diagram shows the names of each column in each table. For a database that doesn't contain many tables, like the AP database, a single database diagram may be adequate for the database. However, for a database that contains a large number of tables, it may be helpful to use several database diagrams. Then, each diagram can focus on a subset of related tables within the database.

When you first create a database diagram for a database, the tables may be placed in an illogical order, and the lines that indicate the relationships between the tables may be tangled. This makes the diagram difficult to read. To fix this, you can use standard Windows techniques to move and size the tables in the diagram. For example, you can drag the title bar of a table to move it, you can drag the edge of a table to resize it, and you can right-click anywhere in the diagram to get a context-sensitive shortcut menu. You can use these menus to add or remove tables from the diagram, or to automatically size a table. With a bit of fiddling around, you should be able to organize your diagram so it's easy to see the relationships between the tables.

When you create a database diagram, the relationships between tables are displayed as links as shown in this figure. You can tell what type of relationship exists between two tables by looking at the endpoints of the link. The “one” side is represented by a key, and the “many” side is represented by an infinity symbol. In this diagram, all of the relationships are one-to-many. For example, one row in the Vendors table can be related to many rows in the Invoices table.

As you review this diagram, notice that you can't tell which columns in each table form the relationship. However, you can see which columns are defined as primary key columns. As you may remember from chapter 1, these are the columns that are typically used on the “one” side of the relationships. From that information, you should be able to figure out which columns identify the foreign keys by reading the names of the columns. For example, it's fairly obvious that the DefaultAccountNo column in the Vendors table is related to the AccountNo column in the GLAccounts table. If you can't determine the relationships just by reading the column names, you can review the primary and foreign keys for each table by using the Object Explorer as described in the next figure.

The relationships between the tables in the AP database



Description

- *Database diagrams* can be used to illustrate the relationships between the tables in a database.
- To create a database diagram, right-click on the Database Diagrams node and select New Database Diagram to display the Add Table dialog box. If you get a dialog box that says that the database doesn't have one or more of the required support objects for database diagramming, you can select Yes to create the support objects. Then, select each table you want to add from the list that's displayed, and click the Add button. When you save the diagram, you'll be asked to enter a name for it.
- To view an existing database diagram, expand the Database Diagrams node for the database, and double-click on the diagram you want to display.
- The relationships between the tables in the diagram appear as links, where the endpoints of the links indicate the type of relationship. A key indicates the “one” side of a relationship, and the infinity symbol (∞) indicates the “many” side of a relationship.
- The primary key for a table appears as a key icon that's displayed to the left of the column or columns that define the primary key.
- You can use standard Windows techniques to move and size the tables in a database diagram to make the diagram easier to understand.

Figure 2-8 How to view the relationships between tables

If necessary, you can use a database diagram to add columns, to remove columns, or to change the names of existing columns. However, these changes actually modify the definition of the database. As a result, you'll only want to use them if the database is under development and you're sure that existing code doesn't depend on any existing columns that you delete or modify.

How to view the column definitions of a table

To view the column definitions of a table, you can use the Object Explorer to expand the Columns node for a table as shown in figure 2-9. In this figure, for example, the Object Explorer shows the columns for the Vendors table. This shows the name and data type for each column, along with an indication of whether or not it can contain null values.

In addition, the columns that define keys are marked with a key icon. Here, the first key icon indicates that the VendorID column is the primary key (PK), and the next two key icons indicate that the DefaultTermsID and DefaultAccountNo columns are foreign keys (FK).

How to modify the column definitions

If you want to modify the columns in a table, or if you want to view more detailed information about a column, you can display the table in a Table Designer tab. To do that, right-click on the table and select the Design command. In this figure, for example, you can see that the Table Designer tab for the Vendors table is displayed on the right side of the Management Studio.

The Table Designer tab is divided into two parts. The top of the tab shows three columns that display the name and data type for the column as well as whether the column allows null values. If you want, you can use these columns to modify these values. For example, if you don't want to allow null values for a column, you can remove the appropriate check mark from the Allow Nulls column.

If you want to display additional information about a column, you can select the column by clicking on its row selector. Then, additional properties are displayed in the Column Properties tab that's displayed at the bottom of the window. In this figure, for example, the properties for the DefaultTermsID column are displayed. As you can see, these properties indicate that this column has a default value of 3. Note that the properties that are available change depending on the data type of the column. For a column with the varchar data type, for example, the properties also indicate the length of the column. You'll learn more about that in chapter 8.

The columns in the Vendors table

The screenshot shows the Microsoft SQL Server Management Studio interface. On the left, the Object Explorer tree view is expanded to show the database structure, including the AP database and its tables like ContactUpdates, GLAccounts, InvoiceArchive, InvoiceLineItems, Invoices, Terms, and Vendors. The Vendors table is selected. On the right, the 'MMA17\SQLEXPRESS.AP - dbo.Vendors' tab in the Table Designer is active. It displays a grid of column definitions:

Column Name	Data Type	Allow Nulls
VendorID	int	<input type="checkbox"/>
VendorName	varchar(50)	<input type="checkbox"/>
VendorAddress1	varchar(50)	<input checked="" type="checkbox"/>
VendorAddress2	varchar(50)	<input checked="" type="checkbox"/>
VendorCity	varchar(50)	<input type="checkbox"/>
VendorState	char(2)	<input type="checkbox"/>
VendorZipCode	varchar(20)	<input type="checkbox"/>
VendorPhone	varchar(50)	<input checked="" type="checkbox"/>
VendorContactLName	varchar(50)	<input checked="" type="checkbox"/>
VendorContactFName	varchar(50)	<input checked="" type="checkbox"/>
DefaultTermsID	int	<input type="checkbox"/>
DefaultAccountNo	int	<input type="checkbox"/>

Below the grid, the 'Column Properties' tab is open, showing detailed properties for the 'DefaultTermsID' column:

- (Name) DefaultTermsID
- Allow Nulls No
- Data Type int
- Default Value or Binding ((3))

The 'Table Designer' tab is also visible at the bottom.

Description

- To view the columns for a table, expand the Tables node, expand the node for the table, and expand the Columns node. This displays the columns in the Object Explorer.
- To modify the columns for a table, expand the Tables node, right-click on the table, and select the Design command to display the table in a Table Designer tab. Then, you can click on the row selector to the left of the column name to display the properties for the column in the Column Properties tab at the bottom of the window. If necessary, you can use the Table Designer tab or the Column Properties tab to modify the properties for a column.

Figure 2-9 How to view or modify the column definitions of a table

How to view the data of a table

If you want to quickly view some data for a table, you can right-click on the table and select the Select Top 1000 Rows command. This automatically generates and executes a query that displays the top 1000 rows of the table in a Results tab that's displayed below the generated query. This works similarly to entering and executing a query as shown in figure 2-11, but it's faster since the query is automatically generated and executed.

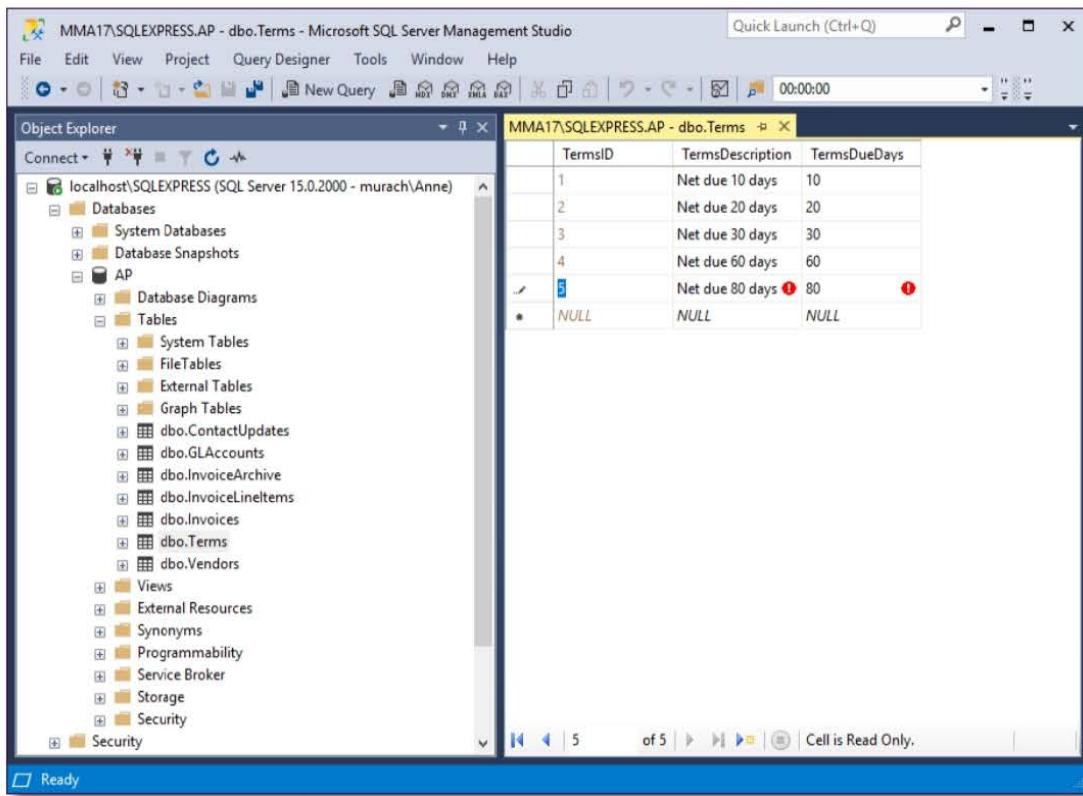
How to modify the data of a table

For tables that have more than 200 rows, you'll need to use SQL statements to modify the data for the table. However, for smaller tables such as the Terms table, the Management Studio provides an easy way to modify the data for the table. To do that, you can right-click on the table and select the Edit Top 200 Rows command. This displays the top 200 rows of the table in an editable grid. In figure 2-10, for example, the Terms table is shown after the Edit Top 200 Rows command has been executed on it. Since this table has fewer than 200 rows, this allows you to edit the entire table.

Once you execute the Edit Top 200 Rows command on a table, you can insert, update, and delete the data for the table. For example, you can insert a new row by entering it in the row at the bottom of the grid that contains NULL values. You can update existing data by clicking on the data you want to update and editing the data. And you can delete an existing row by right-clicking on the row selector to the left of the row and selecting the Delete command.

When you update the data for an existing row, the changes aren't committed to the database until you move the cursor to a different row. In this figure, for example, I have changed the number of days for the fifth row from 90 to 80. However, the changes haven't been committed to the database yet. That's why a red warning icon is displayed after the data for the second and third columns in this row. As a result, you can press the Esc key to roll back these changes. Or, you can move the cursor to another row to commit the changes to the database.

The data in the Terms table with a row being modified



The screenshot shows the Microsoft SQL Server Management Studio interface. The left pane is the Object Explorer, displaying a tree view of the database structure under 'localhost\SQLEXPRESS (SQL Server 15.0.2000 - murach\Anne)'. The 'Tables' node is expanded, showing various tables like ContactUpdates, GLAccounts, InvoiceArchive, InvoiceLineItems, Invoices, Terms, Vendors, etc. The 'Terms' table is selected. The right pane is a grid view titled 'MMA17\SQLEXPRESS.AP - dbo.Terms'. The grid has three columns: 'TermID', 'TermsDescription', and 'TermsDueDays'. There are five rows of data. The fifth row, which is the last one in the grid, has its 'TermsDescription' cell set to 'Net due 80 days' and its 'TermsDueDays' cell set to '80'. Both of these cells have red exclamation marks in their bottom-right corners, indicating they are being edited.

TermID	TermsDescription	TermsDueDays
1	Net due 10 days	10
2	Net due 20 days	20
3	Net due 30 days	30
4	Net due 60 days	60
5	Net due 80 days	80
NULL	NULL	NULL

Description

- To view the data for a table, expand the Tables node, right-click on the table, and select the Select Top 1000 Rows command. This automatically generates and executes a query that displays the top 1000 rows of the table.
- To modify the data for a table, expand the Tables node, right-click on the table, and select the Edit Top 200 Rows command. This displays the top 200 rows of the table in an editable grid. Then, you can use the grid to insert, update, and delete data from the table.

Figure 2-10 How to view or modify the data of a table

How to work with queries

Now that you know how to use the Management Studio to attach a database and view the definition for that database, you're ready to learn how to use this tool to enter and execute queries. You can use this tool to test the queries that are presented throughout this book. As you will see, the Management Studio is a powerful tool that makes it easy to work with queries.

How to enter and execute a query

To enter and edit queries, the Management Studio provides a Query Editor window like the one in figure 2-11. The Query Editor is specifically designed for writing Transact-SQL statements, but it works like most text editors. To begin, you can open a new Query Editor window by clicking on the New Query button in the toolbar. Or, you can open an existing query in a Query Editor window by clicking on the Open button in the toolbar as described in figure 2-13. Once the Query Editor is open, you can use standard techniques to enter or edit the statement in this window.

As you enter statements, you'll notice that the Query Editor automatically applies colors to various elements. For example, keywords are displayed in blue by default, and literal values are displayed in red. This makes your statements easier to read and understand and can help you identify coding errors.

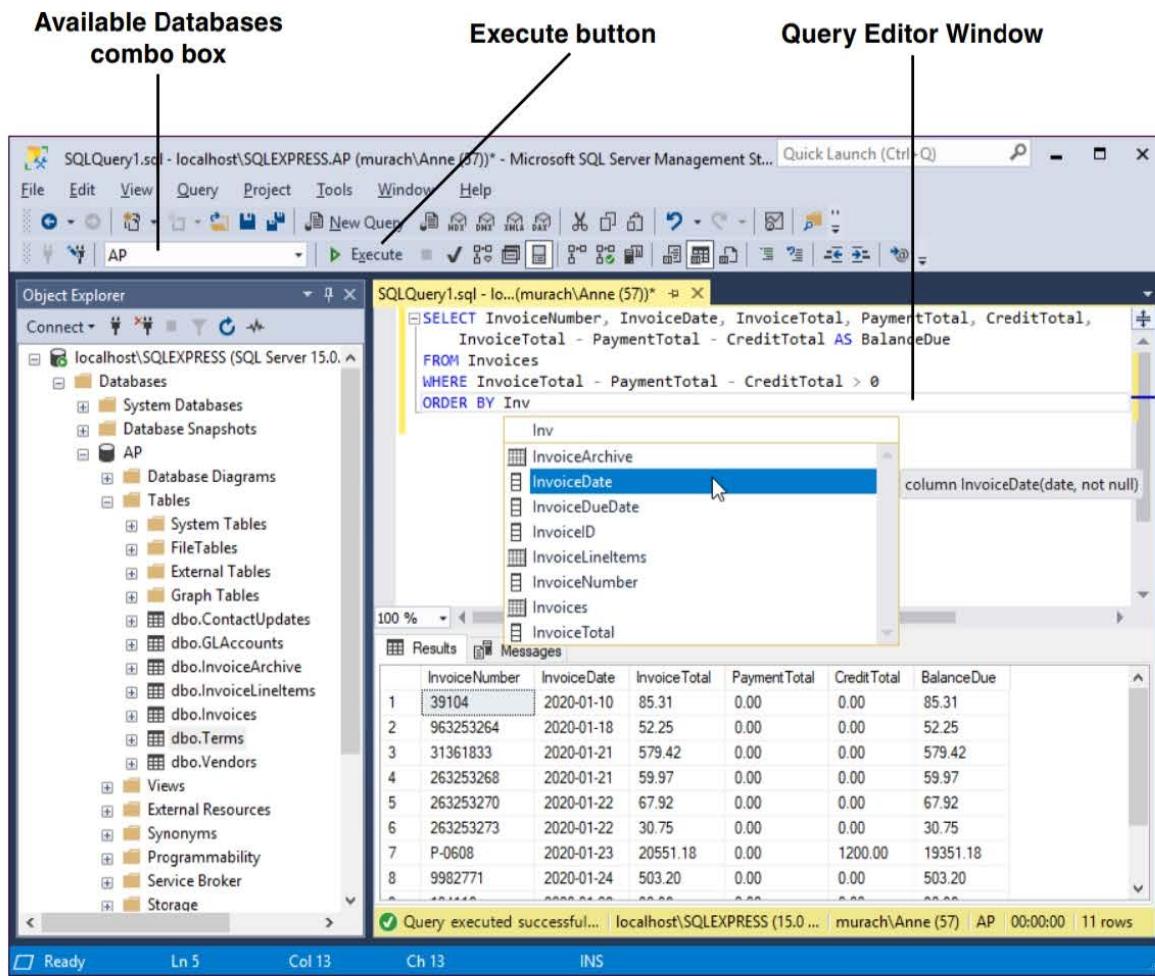
In addition, you'll notice that the Query Editor uses the *IntelliSense* feature to automatically display *completion lists* that you can use to enter parts of the SQL statement. In this figure, for example, one of these lists is being used to enter the InvoiceDate column. As you gain experience with the Management Studio, you'll find that IntelliSense can help you enter most types of SQL statements, even complex ones.

When using IntelliSense, you'll want to be sure that you identify the database the query uses before you start entering the query. That way, IntelliSense can include the names of the tables the database contains in the completion lists. To identify the database, you can select it from the Available Databases combo box in the toolbar. In addition, it's often helpful to enter the table name before you enter the columns. That way, IntelliSense can include the column names for the table in the completion lists.

By default, the IntelliSense feature is on. Since this feature can save you a lot of typing and reduce errors in your code, that's usually what you want. However, it's possible to turn some or all parts of this feature off. In addition, the IntelliSense feature isn't able to work correctly if you aren't connected to the correct SQL Server database engine or if your SQL statement contains some types of errors. As a result, if IntelliSense isn't working properly on your system, you should make sure that it's turned on, that you're connected to the database, and that your statement doesn't contain errors.

To execute a query, you can press F5 or click the Execute button in the toolbar. If the statement returns data, that data is displayed in the Results tab at the bottom of the Query Editor. In this figure, for example, the result set that's returned by the execution of a SELECT statement is displayed in the Results

A SELECT statement with a completion list



Description

- To open a new Query Editor window, click on the New Query button in the toolbar. To open a saved query in a Query Editor window, click on the Open button in the toolbar as described in figure 2-13.
- To select the database that you want to work with, use the Available Databases combo box in the toolbar.
- To enter a SQL statement, type it into the Query Editor window.
- As you enter a SQL statement, the *IntelliSense feature* automatically displays *completion lists* that help you complete the statement. To select an item from a list, use the Up or Down arrow key to select the item and press the Tab key. To hide a list, press the Esc key. To manually display a list, press Alt+Right-arrow or Ctrl+J.
- To execute a SQL statement, press the F5 key or click the Execute button in the toolbar. If the statement retrieves data, the data is displayed in the Results tab that's displayed at the bottom of the Query Editor. Otherwise, a message is displayed in the Messages tab that's displayed at the bottom of the Query Editor.

Figure 2-11 How to enter and execute a query

tab. If you execute an action query, the Messages tab is displayed instead of the Results tab. This tab will contain an indication of the number of rows that were affected by the query. The Messages tab is also used to provide error information, as you'll see in figure 2-12.

How to handle syntax errors

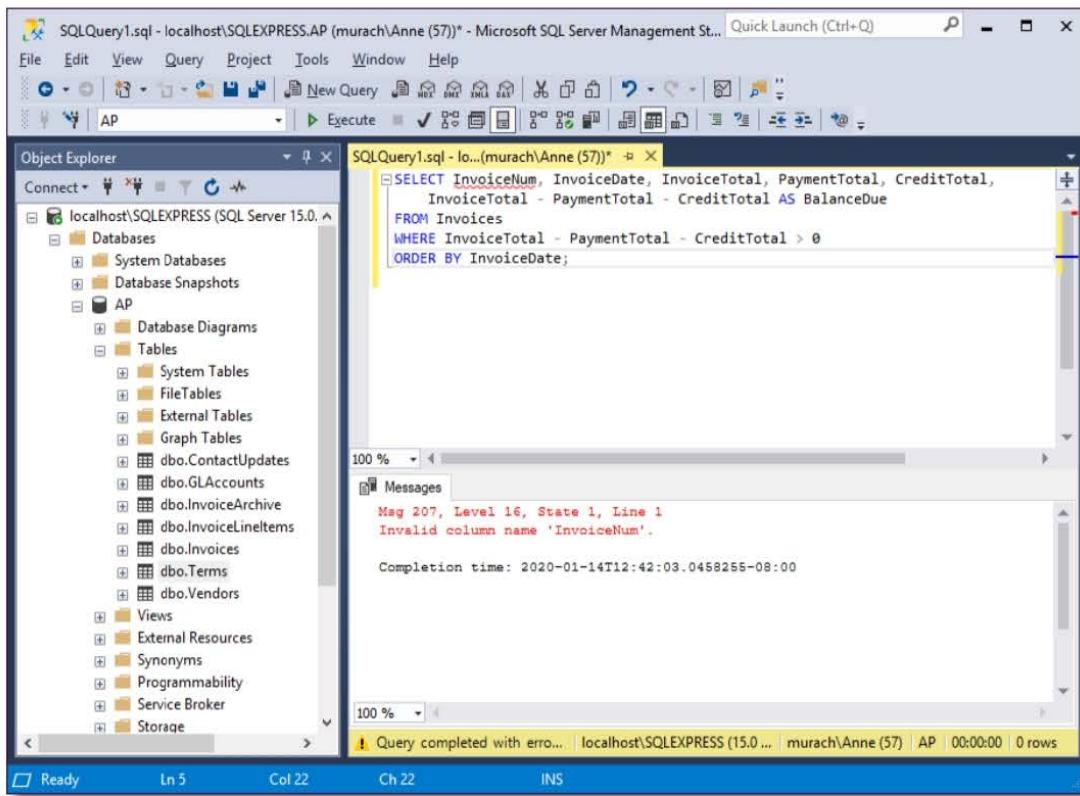
When you are entering a SQL statement, the IntelliSense feature will display wavy red underlining beneath any parts of the SQL statement that contain errors. In figure 2-12, for example, wavy red underlining is displayed beneath the first column in the SELECT statement, the InvoiceNum column. The reason for this error is that there isn't a column with this name in the Invoices table of the AP database, which is the selected database. As a result, you can correct this error by entering a valid column name, such as InvoiceNumber.

If an error occurs during the execution of a SQL statement, an error message is displayed in the Messages tab of the Query Editor. In this figure, for example, the error message indicates that InvoiceNum column is invalid. This, of course, is the same error that was detected by the IntelliSense feature.

One common error when working with SQL statements is to forget to select the correct database from the Available Databases combo box. In this figure, the AP database is selected, which is the correct database for the statement that's entered. However, if the ProductOrders database was selected, this statement would contain many errors since the Invoices table and its columns don't exist in that database. To correct this mistake, you can simply select the appropriate database for the statement.

This figure also lists some other common causes of errors. As you can see, these errors are caused by incorrect syntax. When an error is caused by invalid syntax, you can usually identify and correct the problem without much trouble. In some cases, though, you won't be able to figure out the cause of an error by the information that's provided by IntelliSense or the Messages tab. Then, you can get additional information about the error by looking it up in the SQL Server Reference or by searching the Internet for the error.

How the Management Studio displays an error message



Common causes of errors

- Forgetting to select the correct database from the Available Databases combo box
- Misspelling the name of a table or column
- Misspelling a keyword
- Omitting the closing quotation mark for a character string

Description

- Before you execute a statement, IntelliSense may display wavy red underlining beneath the parts of a SQL statement that contain errors.
- If an error occurs during the execution of a SQL statement, the Management Studio displays an error message in the Messages tab of the Query Editor.
- Most errors are caused by incorrect syntax and can be detected and corrected without any additional assistance. If not, you can get more information about an error by looking it up in the SQL Server Reference.

Figure 2-12 How to handle syntax errors

How to open and save queries

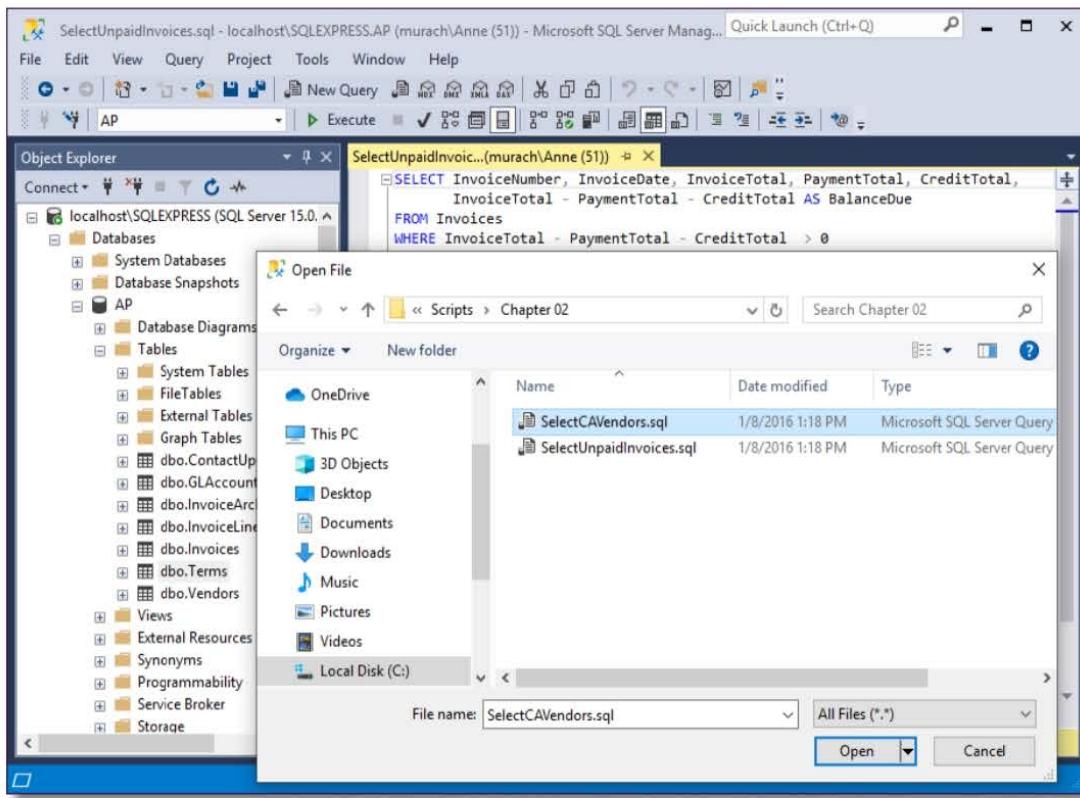
After you get a query working the way you want it to work, you may want to save it. Then, you can open it and run it again later or use it as the basis for a new query. To do that, you use the techniques in figure 2-13.

If you've used other Windows programs, you shouldn't have any trouble opening and saving query files. To save a new query, for example, or to save a modified query in the original file, you use the standard Save command. To save a modified query in a new file, you use the standard Save As command. And to open a query, you use the standard Open command. Note that when you save a query, it's saved with a file extension of sql.

As you work with queries, you may find it helpful to open two or more queries at the same time. To do that, you can open additional Query Editor windows by starting a new query or by opening an existing query. After you open two or more windows, you can switch between the queries by clicking on the appropriate tab. Then, if necessary, you can cut, copy, and paste code from one query to another.

If you open many queries and not all tabs are visible, you can use the Active Files list to switch between queries. To display this list, click on the drop-down arrow that's displayed to the right of the Query Editor tabs. Then, select the query you want from the list of active files.

The Open File dialog box



Description

- To save a query, click the Save button in the toolbar or press Ctrl+S. Then, if necessary, use the Save File As dialog box to specify a file name for the query.
- To open a query, click the Open button in the toolbar or press Ctrl+O. Then, use the Open File dialog box shown above to locate and open the query.
- To save all open queries, click the Save All button in the toolbar. Then, if necessary, use the Save File As dialog box to specify a file name for any queries that haven't already been named.
- To switch between open queries, click on the tab for the query you want to display. If you can't see the tab, click on the drop-down arrow that's displayed to the right of the Query Editor tabs, and select the query from the list of active files.
- To cut, copy, and paste code from one query to another, use the standard Windows techniques.

Figure 2-13 How to open and save queries

An introduction to the Query Designer

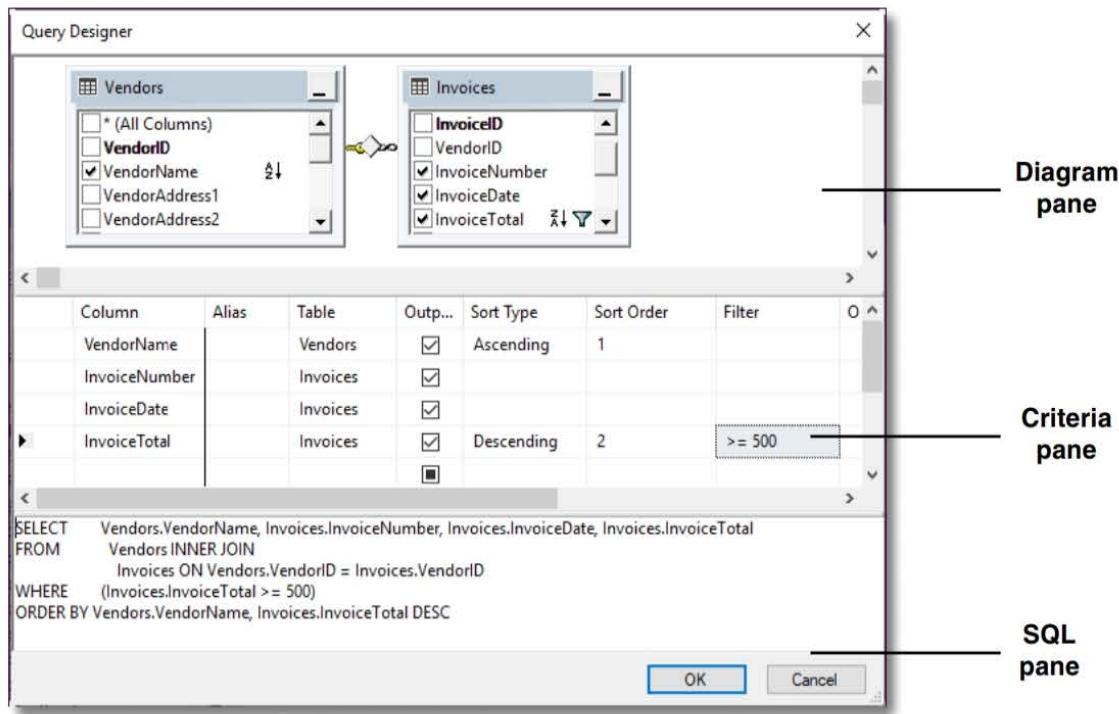
Figure 2-14 presents the Query Designer, a tool that can help you create queries using a graphical interface. In the Diagram pane, you select the tables and columns that you want to use in the query. Then, the columns you select are listed in the Criteria pane, and you can use this pane to set the criteria and sort sequence for the query. As you work in the Diagram and Criteria panes, the Query Designer generates a SQL statement and displays it in the SQL pane. When you have the statement the way you want it, you can click the OK button to insert the query into the Query Editor. From the Query Editor, you can edit the text for the query and run it just as you would any other query.

When you first start working with a database, the Query Designer can help you become familiar with the tables and columns it contains. In addition, it can help you build simple queries quickly and easily. If you analyze the SQL statements that it generates, it can also help you learn SQL.

Keep in mind, though, that the best way to learn SQL is to code it yourself. That's why this book emphasizes the use of the Query Editor. Plus, it can be difficult, and sometimes impossible, to create certain types of complex queries using the Query Designer. Because of that, you're usually better off using the Query Editor to enter complex queries yourself.

Although this figure shows how to use the Query Designer to create a SELECT statement, you should know that you can also use it to create INSERT, UPDATE, and DELETE statements. To start one of these queries, you can right-click anywhere in the Query Designer window, select the Change Type submenu, and select the type of query that you want to create.

The Query Designer window



The three panes in the Query Designer window

Pane	Description
Diagram pane	Displays the tables used by the query and lets you select the columns you want to include in the query.
Criteria pane	Displays the columns selected in the Diagram pane and lets you specify the sort order and the criteria you want to use to select the rows for the result set. You can also use this pane to select or deselect the columns that are included in the output and to create calculated values.
SQL pane	Displays the SQL statement built by the Query Designer based on the information in the Diagram and Criteria panes.

Description

- You can use the Query Designer to build simple queries quickly and easily. However, you may not be able to create more complex queries this way.
- To display the Query Designer, right-click on a blank Query Editor window and select the Design Query in Editor command. Then, you can use the Query Designer window to create the query. When you click the OK button, the query will be inserted into the Query Editor where you can edit and run it just as you would any other query.
- To modify a query with the Query Designer, select the query, right-click on the selection, and select the Design Query in Editor command.

Figure 2-14 An introduction to the Query Designer

How to view the documentation for SQL Server

Sometimes, you need to look up information about SQL statements, or you need to get more information about an error message that's returned by SQL Server. To do that, you can use your browser to search the Internet. Often, this is the quickest and easiest way to find the information you're looking for.

However, there are also times when you need to view the official documentation for SQL Server. To do that, you can start a browser and navigate to the website for the technical documentation for SQL Server as shown in figure 2-15.

How to display the SQL Server documentation

To display the documentation for SQL Server, you can begin by searching the Internet for "SQL Server technical documentation". Then, you can click on the search result that leads to that documentation. This displays a page with a table of contents in the left column and links to information about topics related to SQL Server in the main area of the page.

How to look up information in the documentation

Once you've displayed the main page for the SQL Server documentation, you can display information about a specific topic using a number of techniques. First, you can click on a link in the main area of the page. Second, you can use the table of contents to locate and display the information you need. To display the topic on the SELECT statement shown in this figure, for example, I clicked several links in the table of contents until I found the topic I wanted.

You can also filter the table of contents by title to locate a topic. To do that, just enter the text you want to search for in the Filter by Title text box and then select a topic from the ones that are listed. For example, I could have displayed the topic on the SELECT statement by entering "select" in this text box and then selecting the appropriate topic. Note that if you enter this text, though, a list of almost 50 titles will be displayed. Because of that, you'll want to use this feature only if you can enter more specific text.

The most efficient way to find the information you need is typically to use the full-text search feature. For example, I could also have located the topic on the SELECT statement by entering "select statement" in the Search text box, pressing the Enter key to display a list of topics on that statement, and clicking the appropriate topic. Although this can display a large number of topics, the one you want is typically near the topic if the search text is specific.

The documentation for the SELECT statement

The screenshot shows a web browser displaying the Microsoft SQL Docs website at docs.microsoft.com/en-us/sql/t-sql/queries/select-transact-sql?view=sql-server-ver15. The page title is "SELECT (Transact-SQL)". The left sidebar has a "Version" dropdown set to "SQL Server 2019" and a "Filter by title" dropdown. The main content area contains the "APPLIES TO:" section with checkboxes for SQL Server, Azure SQL Database, Azure Synapse Analytics (SQL DW), and Parallel Data Warehouse. Below this is a summary of the SELECT statement's purpose and syntax examples. The right sidebar includes sections for "Is this page helpful?", "In this article", and links to Syntax, Remarks, Logical Processing, Order of the SELECT statement, Permissions, Examples, and See Also.

Description

- To view the official documentation for SQL Server 2019, you can start a web browser and search for “SQL Server technical documentation”.
- The main page of the documentation displays a table of contents in the left column and links to various topics related to SQL Server in the main area of the page.
- You can click on any link in the main area to display information about that topic.
- To use the table of contents, click on a topic to display it in the middle pane. You can also filter the contents by title by entering text in the Filter by Title text box above the table and contents and then selecting a title from the list that’s displayed.
- To use full-text search, click the search icon in the upper right corner that looks like a magnifying glass, enter the text into the Search text box, and press the Enter key. Then, select the topic that you want from the resulting list of topics.

Figure 2-15 How to view the documentation for SQL Server

Perspective

In this chapter, you learned how to use the tools that you need to begin learning about SQL. To begin, you learned how to start and stop the database server. Then, you learned how to use the Management Studio to connect to the database server, to attach a database, to view the definition of a database, and to execute SQL statements against that database. Finally, you learned how to view the official documentation for SQL Server. With that as background, you're ready to go on to the next chapter where you'll start learning the details of coding your own SQL statements.

Before you go on to the next chapter, though, I recommend that you install SQL Server Express and the Management Studio on your system as described in appendix A. In addition, I recommend that you download and install the databases and sample code that come with this book as described in appendix A. That way, you can begin experimenting with these tools. In particular, you can enter and execute queries like the ones described in this chapter. Or, you can open any of the queries shown in this chapter, view their code, and execute them.

For now, though, focus on the mechanics of using the Management Studio to enter and execute queries, and don't worry if you don't understand the details of how the SQL statements are coded. You'll learn the details for coding SQL statements in the chapters that follow. In the next chapter, for example, you'll learn the details for coding a SELECT statement that retrieves data from a single table.

Terms

- database server
- client tools
- database engine
- SQL Server 2019 Express Edition
- SQL Server Express
- SQL Server Management Studio
- SQL Server Configuration Manager
- schema
- attach a database
- detach a database
- back up a database
- restore a database
- database diagram
- Query Editor
- IntelliSense feature
- completion list
- Query Designer

Before you do the exercises for this chapter

If you're working on your own PC, you'll need to set up your system as described in appendix A before you can do these exercises.

Exercises

1. Use the Management Studio to view all of the databases that are available from the server. If the AP database isn't available, follow the procedure in appendix A to create it. Then, view the tables that are available from the AP database. Finally, view the columns that are available from the Invoices table. Note the primary and foreign keys of this table and the definition for each column.
2. Right-click on the Vendors table and select the Design command to display the Vendors table in a Table Designer tab. Review the properties for each column in this table. In particular, note that the VendorID column is defined as an identity column.
3. Use the Management Studio to create a database diagram for the AP database. The diagram should include the Vendors, Invoices, InvoiceLineItems, Terms, and GLAccounts tables. Save the diagram with any name you'd like.
4. Organize the tables and connecting lines in the diagram you created in step 3 so they are easy to read. (Hint: You can use the Autosize Selected Tables button and the Arrange Tables button in the toolbar to help you do this.) Finally, review the information that's contained in each table, note the primary key of each table, and try to identify the relationships between the tables.
5. Open a new Query Editor window and then enter this SELECT statement:

```
SELECT VendorName, VendorState  
FROM Vendors  
WHERE VendorState = 'CA'
```

Press F5 to execute the query and display the results. If an error is displayed, correct the problem before you continue. (Hint: If you get an error message that indicates that 'Vendors' isn't a valid object, the AP database isn't the current database. To fix this error, select the AP database from the Available Databases combo box.) Then, save the query with a name of VendorsInCA and close it.

6. Open the query named VendorsInCA that you saved in exercise 5. Then, click the Execute Query toolbar button to execute it. (Hint: You may need to select the AP database from the Available Databases combo box.)
7. Look up information about the Query Editor in the SQL Server documentation. The easiest way to do this is to use the full-text search to look up "query editor". Continue experimenting with the documentation until you feel comfortable with it.

Section 2

The essential SQL skills

This section teaches you the essential SQL coding skills for working with the data in a SQL Server database. The first four chapters in this section show you how to retrieve data from a database using the SELECT statement. In chapter 3, you'll learn how to code the basic clauses of the SELECT statement to retrieve data from a single table. In chapter 4, you'll learn how to get data from two or more tables. In chapter 5, you'll learn how to summarize the data that you retrieve. And in chapter 6, you'll learn how to code subqueries, which are SELECT statements coded within other statements.

Next, chapter 7 shows you how to use the INSERT, UPDATE, and DELETE statements to add, update, and delete rows in a table. Chapter 8 shows you how to work with the various types of data that SQL Server supports. And finally, chapter 9 shows you how to use some of the SQL Server functions for working with data in your SQL statements. When you complete these chapters, you'll have the skills you need to code most any SELECT, INSERT, UPDATE, or DELETE statement.

3

How to retrieve data from a single table

In this chapter, you'll learn how to code SELECT statements that retrieve data from a single table. You should realize, though, that the skills covered here are the essential ones that apply to any SELECT statement you code...no matter how many tables it operates on, no matter how complex the retrieval. So you'll want to be sure you have a good understanding of the material in this chapter before you go on to the chapters that follow.

An introduction to the SELECT statement	86
The basic syntax of the SELECT statement.....	86
SELECT statement examples	88
How to code the SELECT clause	90
How to code column specifications	90
How to name the columns in a result set.....	92
How to code string expressions	94
How to code arithmetic expressions	96
How to use functions	98
How to use the DISTINCT keyword to eliminate duplicate rows	100
How to use the TOP clause to return a subset of selected rows.....	102
How to code the WHERE clause	104
How to use comparison operators	104
How to use the AND, OR, and NOT logical operators	106
How to use the IN operator.....	108
How to use the BETWEEN operator	110
How to use the LIKE operator.....	112
How to use the IS NULL clause	114
How to code the ORDER BY clause	116
How to sort a result set by a column name.....	116
How to sort a result set by an alias, an expression, or a column number....	118
How to retrieve a range of selected rows.....	120
Perspective	122

An introduction to the SELECT statement

To help you learn to code SELECT statements, this chapter starts by presenting its basic syntax. Next, it presents several examples that will give you an idea of what you can do with this statement. Then, the rest of this chapter will teach you the details of coding this statement.

The basic syntax of the SELECT statement

Figure 3-1 presents the basic syntax of the SELECT statement. The syntax summary at the top of this figure uses conventions that are similar to those used in other programming manuals. Capitalized words are *keywords* that you have to type exactly as shown. By contrast, you have to provide replacements for the lowercase words. For example, you can enter a list of columns in place of *select_list*, and you can enter a table name in place of *table_source*.

Beyond that, you can choose between the items in a syntax summary that are separated by pipes (|) and enclosed in braces ({}) or brackets ([]). And you can omit items enclosed in brackets. If you have a choice between two or more optional items, the default item is underlined. And if an element can be coded multiple times in a statement, it's followed by an ellipsis (...). You'll see examples of pipes, braces, default values, and ellipses in syntax summaries later in this chapter. For now, if you compare the syntax in this figure with the coding examples in the next figure, you should easily see how the two are related.

The syntax summary in this figure has been simplified so you can focus on the four main clauses of the SELECT statement: SELECT, FROM, WHERE, and ORDER BY. Most of the SELECT statements you code will contain all four of these clauses. However, only the SELECT clause is always required, and the FROM clause is required when you retrieve data from one or more tables.

The SELECT clause is always the first clause in a SELECT statement. It identifies the columns that will be included in the result set. These columns are retrieved from the base tables named in the FROM clause. Since this chapter focuses on retrieving data from a single table, the FROM clauses in all of the statements shown in this chapter name a single base table. In the next chapter, though, you'll learn how to retrieve data from two or more tables.

The WHERE and ORDER BY clauses are optional. The ORDER BY clause determines how the rows in the result set are sorted, and the WHERE clause determines which rows in the base table are included in the result set. The WHERE clause specifies a search condition that's used to *filter* the rows in the base table. This search condition can consist of one or more *Boolean expressions*, or *predicates*. A Boolean expression is an expression that evaluates to True or False. When the search condition evaluates to True, the row is included in the result set.

In this book, I won't use the terms "Boolean expression" or "predicate" because I don't think they clearly describe the content of the WHERE clause. Instead, I'll just use the term "search condition" to refer to an expression that evaluates to True or False.

The simplified syntax of the SELECT statement

```
SELECT select_list  
[FROM table_source]  
[WHERE search_condition]  
[ORDER BY order_by_list]
```

The four clauses of the SELECT statement

Clause	Description
SELECT	Describes the columns that will be included in the result set.
FROM	Names the table from which the query will retrieve the data.
WHERE	Specifies the conditions that must be met for a row to be included in the result set. This clause is optional.
ORDER BY	Specifies how the rows in the result set will be sorted. This clause is optional.

Description

- You use the basic SELECT statement shown above to retrieve the columns specified in the SELECT clause from the base table specified in the FROM clause and store them in a result set.
- The WHERE clause is used to *filter* the rows in the base table so that only those rows that match the search condition are included in the result set. If you omit the WHERE clause, all of the rows in the base table are included.
- The search condition of a WHERE clause consists of one or more *Boolean expressions*, or *predicates*, that result in a value of True, False, or Unknown. If the combination of all the expressions is True, the row being tested is included in the result set. Otherwise, it's not.
- If you include the ORDER BY clause, the rows in the result set are sorted in the specified sequence. Otherwise, the rows are returned in the same order as they appear in the base table. In most cases, that means that they're returned in primary key sequence.

Note

- The syntax shown above does not include all of the clauses of the SELECT statement. You'll learn about the other clauses later in this book.

SELECT statement examples

Figure 3-2 presents five SELECT statement examples. All of these statements retrieve data from the Invoices table. If you aren't already familiar with this table, you should use the Management Studio as described in the last chapter to review its definition.

The first statement in this figure retrieves all of the rows and columns from the Invoices table. Here, an asterisk (*) is used as a shorthand to indicate that all of the columns should be retrieved, and the WHERE clause is omitted so there are no conditions on the rows that are retrieved. Notice that this statement doesn't include an ORDER BY clause, so the rows are in primary key sequence. You can see the results following this statement as they're displayed by the Management Studio. Notice that both horizontal and vertical scroll bars are displayed, indicating that the result set contains more rows and columns than can be displayed on the screen at one time.

The second statement retrieves selected columns from the Invoices table. As you can see, the columns to be retrieved are listed in the SELECT clause. Like the first statement, this statement doesn't include a WHERE clause, so all the rows are retrieved. Then, the ORDER BY clause causes the rows to be sorted by the InvoiceTotal column in ascending sequence.

The third statement also lists the columns to be retrieved. In this case, though, the last column is calculated from two columns in the base table, CreditTotal and PaymentTotal, and the resulting column is given the name TotalCredits. In addition, the WHERE clause specifies that only the invoice whose InvoiceID column has a value of 17 should be retrieved.

The fourth SELECT statement includes a WHERE clause whose condition specifies a range of values. In this case, only invoices with invoice dates between 01/01/2020 and 03/31/2020 are retrieved. In addition, the rows in the result set are sorted by invoice date.

The last statement in this figure shows another variation of the WHERE clause. In this case, only those rows with invoice totals greater than 50,000 are retrieved. Since none of the rows in the Invoices table satisfy this condition, the result set is empty.

A SELECT statement that retrieves all the data from the Invoices table

```
SELECT *
FROM Invoices;
```

	InvoiceID	VendorID	InvoiceNumber	InvoiceDate	InvoiceTotal	PaymentTotal	CreditTotal
1	1	122	989319-457	2019-10-08	3813.33	3813.33	0.00
2	2	123	263253241	2019-10-10	40.20	40.20	0.00
3	3	123	963253234	2019-10-13	138.75	138.75	0.00
4	4	123	2-000-2993	2019-10-16	144.70	144.70	0.00

(114 rows)

A SELECT statement that retrieves three columns from each row, sorted in ascending sequence by invoice total

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
ORDER BY InvoiceTotal;
```

	InvoiceNumber	InvoiceDate	InvoiceTotal
1	25022117	2019-11-01	6.00
2	24863706	2019-11-10	6.00
3	24780512	2019-12-22	6.00
4	21-4923721	2019-11-13	9.95

(114 rows)

A SELECT statement that retrieves two columns and a calculated value for a specific invoice

```
SELECT InvoiceID, InvoiceTotal, CreditTotal + PaymentTotal AS TotalCredits
FROM Invoices
WHERE InvoiceID = 17;
```

	InvoiceID	InvoiceTotal	TotalCredits
1	17	10.00	10.00

A SELECT statement that retrieves all invoices between given dates

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
WHERE InvoiceDate BETWEEN '2020-01-01' AND '2020-03-31'
ORDER BY InvoiceDate;
```

	InvoiceNumber	InvoiceDate	InvoiceTotal
1	989319-467	2020-01-01	2318.03
2	263253265	2020-01-02	26.25
3	203339-13	2020-01-05	17.50
4	963253258	2020-01-06	111.00

(35 rows)

A SELECT statement that returns an empty result set

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
WHERE InvoiceTotal > 50000;
```

	InvoiceNumber	InvoiceDate	InvoiceTotal

Figure 3-2 SELECT statement examples

How to code the SELECT clause

Figure 3-3 presents an expanded syntax for the SELECT clause. The keywords shown in the first line allow you to restrict the rows that are returned by a query. You'll learn how to code them in a few minutes. First, though, you'll learn various techniques for identifying which columns are to be included in a result set.

How to code column specifications

Figure 3-3 summarizes the techniques you can use to code column specifications. You saw how to use some of these techniques in the previous figure. For example, you can code an asterisk in the SELECT clause to retrieve all of the columns in the base table, and you can code a list of column names separated by commas. Note that when you code an asterisk, the columns are returned in the order that they occur in the base table.

You can also code a column specification as an *expression*. For example, you can use an arithmetic expression to perform a calculation on two or more columns in the base table, and you can use a string expression to combine two or more string values. An expression can also include one or more functions. You'll learn more about each of these techniques in the topics that follow.

But first, you should know that when you code the SELECT clause, you should include only the columns you need. For example, you shouldn't code an asterisk to retrieve all the columns unless you need all the columns. That's because the amount of data that's retrieved can affect system performance. This is particularly important if you're developing SQL statements that will be used by application programs.

The expanded syntax of the SELECT clause

```
SELECT [ALL|DISTINCT] [TOP n [PERCENT] [WITH TIES]]
    column_specification [[AS] result_column]
    [, column_specification [[AS] result_column]] ...
```

Five ways to code column specifications

Source	Option	Syntax
Base table value	All columns	*
	Column name	column_name
Calculated value	Result of a calculation	Arithmetic expression (see figure 3-6)
	Result of a concatenation	String expression (see figure 3-5)
	Result of a function	Function (see figure 3-7)

Column specifications that use base table values

The * is used to retrieve all columns

```
SELECT *
```

Column names are used to retrieve specific columns

```
SELECT VendorName, VendorCity, VendorState
```

Column specifications that use calculated values

An arithmetic expression is used to calculate BalanceDue

```
SELECT InvoiceNumber,
       InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue
```

A string expression is used to calculate FullName

```
SELECT VendorContactFName + ' ' + VendorContactLName AS FullName
```

A function is used to calculate CurrentDate

```
SELECT InvoiceNumber, InvoiceDate,
       GETDATE() AS CurrentDate
```

Description

- Use SELECT * only when you need to retrieve all of the columns from a table. Otherwise, list the names of the columns you need.
- An *expression* is a combination of column names and operators that evaluate to a single value. In the SELECT clause, you can code arithmetic expressions, string expressions, and expressions that include one or more functions.
- After each column specification, you can code an AS clause to specify the name for the column in the result set. See figure 3-4 for details.

Note

- The ALL and DISTINCT keywords and the TOP clause let you control the number of rows that are returned by a query. See figures 3-8 and 3-9 for details.

How to name the columns in a result set

By default, a column in a result set is given the same name as the column in the base table. However, you can specify a different name if you need to. You can also name a column that contains a calculated value. When you do that, the new column name is called a *column alias*. Figure 3-4 presents two techniques for creating column aliases.

The first technique is to code the column specification followed by the AS keyword and the column alias. This is the ANSI-standard coding technique, and it's illustrated by the first example in this figure. Here, a space is added between the two words in the name of the InvoiceNumber column, the InvoiceDate column is changed to just Date, and the InvoiceTotal column is changed to Total. Notice that because a space is included in the name of the first column, it's enclosed in brackets ([]). As you'll learn in chapter 10, any name that doesn't follow SQL Server's rules for naming objects must be enclosed in either brackets or double quotes. Column aliases can also be enclosed in single quotes.

The second example in this figure illustrates another technique for creating a column alias. Here, the column is assigned to an alias using an equal sign. This technique is only available with SQL Server, not with other types of databases, and is included for compatibility with earlier versions of SQL Server. So although you may see this technique used in older code, I don't recommend it for new statements you write.

The third example in this figure illustrates what happens when you don't assign an alias to a calculated column. Here, no name is assigned to the column, which usually isn't what you want. That's why you usually assign a name to any column that's calculated from other columns in the base table.

Two SELECT statements that name the columns in the result set

A SELECT statement that uses the AS keyword (the preferred technique)

```
SELECT InvoiceNumber AS [Invoice Number], InvoiceDate AS Date,
       InvoiceTotal AS Total
  FROM Invoices;
```

A SELECT statement that uses the equal operator (an older technique)

```
SELECT [Invoice Number] = InvoiceNumber, Date = InvoiceDate,
       Total = InvoiceTotal
  FROM Invoices;
```

The result set for both SELECT statements

	Invoice Number	Date	Total
1	989319-457	2019-10-08	3813.33
2	263253241	2019-10-10	40.20
3	963253234	2019-10-13	138.75
4	2-000-2993	2019-10-16	144.70
5	963253251	2019-10-16	15.50

A SELECT statement that doesn't provide a name for a calculated column

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
       InvoiceTotal - PaymentTotal - CreditTotal
  FROM Invoices;
```

	InvoiceNumber	InvoiceDate	InvoiceTotal	(No column name)
1	989319-457	2019-10-08	3813.33	0.00
2	263253241	2019-10-10	40.20	0.00
3	963253234	2019-10-13	138.75	0.00
4	2-000-2993	2019-10-16	144.70	0.00
5	963253251	2019-10-16	15.50	0.00

Description

- By default, a column in the result set is given the same name as the column in the base table. If that's not what you want, you can specify a *column alias* or *substitute name* for the column.
- One way to name a column is to use the AS phrase as shown in the first example above. Although the AS keyword is optional, I recommend you code it for readability.
- Another way to name a column is to code the name followed by an equal sign and the column specification as shown in the second example above. This syntax is unique to Transact-SQL.
- It's generally considered a good practice to specify an alias for a column that contains a calculated value. If you don't, no name is assigned to it as shown in the third example above.
- If an alias includes spaces or special characters, you must enclose it in double quotes or brackets ([]). That's true of all names you use in Transact-SQL. SQL Server also lets you enclose column aliases in single quotes for compatibility with earlier releases.

Figure 3-4 How to name the columns in a result set

How to code string expressions

A *string expression* consists of a combination of one or more character columns and *literal values*. To combine, or *concatenate*, the columns and values, you use the *concatenation operator* (+). This is illustrated by the examples in figure 3-5.

The first example shows how to concatenate the VendorCity and VendorState columns in the Vendors table. Notice that because no alias is assigned to this column, it doesn't have a name in the result set. Also notice that the data in the VendorState column appears immediately after the data in the VendorCity column in the results. That's because of the way VendorCity is defined in the database. Because it's defined as a variable-length column (the varchar data type), only the actual data in the column is included in the result. By contrast, if the column had been defined with a fixed length, any spaces following the name would have been included in the result. You'll learn about data types and how they affect the data in your result set in chapter 8.

The second example shows how to format a string expression by adding spaces and punctuation. Here, the VendorCity column is concatenated with a *string literal*, or *string constant*, that contains a comma and a space. Then, the VendorState column is concatenated with that result, followed by a string literal that contains a single space and the VendorZipCode column.

Occasionally, you may need to include a single quotation mark or an apostrophe within a literal string. If you simply type a single quote, however, the system will misinterpret it as the end of the literal string. As a result, you must code two quotation marks in a row. This is illustrated by the third example in this figure.

How to concatenate string data

```
SELECT VendorCity, VendorState, VendorCity + VendorState
FROM Vendors;
```

	VendorCity	VendorState	(No column name)
1	Madison	WI	MadisonWI
2	Washington	DC	WashingtonDC
3	Washington	DC	WashingtonDC

How to format string data using literal values

```
SELECT VendorName,
       VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors;
```

	VendorName	Address
1	US Postal Service	Madison, WI 53707
2	National Information Data Ctr	Washington, DC 20090
3	Register of Copyrights	Washington, DC 20559
4	Jobtrak	Los Angeles, CA 90025

How to include apostrophes in literal values

```
SELECT VendorName + "'s Address: ',
       VendorCity + ', ' + VendorState + ' ' + VendorZipCode
FROM Vendors;
```

	(No column name)	(No column name)
1	US Postal Service's Address:	Madison, WI 53707
2	National Information Data Ctr's Address:	Washington, DC 20090
3	Register of Copyrights's Address:	Washington, DC 20559
4	Jobtrak's Address:	Los Angeles, CA 90025
5	Newbridge Book Clubs's Address:	Washington, NJ 07882
6	California Chamber Of Commerce's Address:	Sacramento, CA 95827

Description

- A *string expression* can consist of one or more character columns, one or more *literal values*, or a combination of character columns and literal values.
- The columns specified in a string expression must contain string data (that means they're defined with a data type such as char or varchar).
- The literal values in a string expression also contain string data, so they can be called *string literals* or *string constants*. To create a literal value, enclose one or more characters within single quotation marks (').
- You can use the *concatenation operator* (+) to combine columns and literals in a string expression.
- You can include a single quote within a literal value by coding two single quotation marks as shown in the third example above.

Figure 3-5 How to code string expressions

How to code arithmetic expressions

Figure 3-6 shows how to code *arithmetic expressions*. To start, it summarizes the five *arithmetic operators* you can use in this type of expression. Then, it presents three examples that illustrate how you use these operators.

The SELECT statement in the first example includes an arithmetic expression that calculates the balance due for an invoice. This expression subtracts the PaymentTotal and CreditTotal columns from the InvoiceTotal column. The resulting column is given the name BalanceDue.

When SQL Server evaluates an arithmetic expression, it performs the operations from left to right based on the *order of precedence*. This order says that multiplication, division, and modulo operations are done first, followed by addition and subtraction. If that's not what you want, you can use parentheses to specify how you want an expression evaluated. Then, the expressions in the innermost sets of parentheses are evaluated first, followed by the expressions in outer sets of parentheses. Within each set of parentheses, the expression is evaluated from left to right in the order of precedence. Of course, you can also use parentheses to clarify an expression even if they're not needed for the expression to be evaluated properly.

To illustrate how parentheses and the order of precedence affect the evaluation of an expression, consider the second example in this figure. Here, the expressions in the second and third columns both use the same operators. When SQL Server evaluates the expression in the second column, it performs the multiplication operation before the addition operation because multiplication comes before addition in the order of precedence. When SQL Server evaluates the expression in the third column, however, it performs the addition operation first because it's enclosed in parentheses. As you can see in the result set shown here, these two expressions result in different values.

Although you're probably familiar with the addition, subtraction, multiplication, and division operators, you may not be familiar with the modulo operator. This operator returns the remainder of a division of two integers. This is illustrated in the third example in this figure. Here, the second column contains an expression that returns the quotient of a division operation. Note that the result of the division of two integers is always an integer. You'll learn more about that in chapter 8. The third column contains an expression that returns the remainder of the division operation. If you study this example for a minute, you should quickly see how this works.

The arithmetic operators in order of precedence

*	Multiplication
/	Division
%	Modulo (Remainder)
+	Addition
-	Subtraction

A SELECT statement that calculates the balance due

```
SELECT InvoiceTotal, PaymentTotal, CreditTotal,
       InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue
FROM Invoices;
```

	InvoiceTotal	PaymentTotal	CreditTotal	BalanceDue
1	3813.33	3813.33	0.00	0.00
2	40.20	40.20	0.00	0.00
3	138.75	138.75	0.00	0.00

A SELECT statement that uses parentheses to control the sequence of operations

```
SELECT InvoiceID,
       InvoiceID + 7 * 3 AS OrderOfPrecedence,
       (InvoiceID + 7) * 3 AS AddFirst
FROM Invoices
ORDER BY InvoiceID;
```

	InvoiceID	OrderOfPrecedence	AddFirst
1	1	22	24
2	2	23	27
3	3	24	30

A SELECT statement that uses the modulo operator

```
SELECT InvoiceID,
       InvoiceID / 10 AS Quotient,
       InvoiceID % 10 AS Remainder
FROM Invoices
ORDER BY InvoiceID;
```

	InvoiceID	Quotient	Remainder
9	9	0	9
10	10	1	0
11	11	1	1

Description

- Unless parentheses are used, the operations in an expression take place from left to right in the *order of precedence*. For arithmetic expressions, multiplication, division, and modulo operations are done first, followed by addition and subtraction.
- Whenever necessary, you can use parentheses to clarify or override the sequence of operations. Then, the operations in the innermost sets of parentheses are done first, followed by the operations in the next sets, and so on.

How to use functions

Figure 3-7 introduces you to *functions* and illustrates how you use them in column specifications. A function performs an operation and returns a value. For now, don't worry about the details of how the functions shown here work. You'll learn more about all of these functions in chapter 9. Instead, just focus on how they're used in column specifications.

To code a function, you begin by entering its name followed by a set of parentheses. If the function requires one or more *parameters*, you enter them within the parentheses and separate them with commas. When you enter a parameter, you need to be sure it has the correct data type. You'll learn more about that in chapter 9.

The first example in this figure shows how to use the LEFT function to extract the first character of the VendorContactFName and VendorContactLName columns. The first parameter of this function specifies the string value, and the second parameter specifies the number of characters to return. The results of the two functions are then concatenated to form initials as shown in the result set for this statement.

The second example shows how to use the CONVERT function to change the data type of a value. This function requires two parameters. The first parameter specifies the new data type, and the second parameter specifies the value to convert. In addition, this function accepts an optional third parameter that specifies the format of the returned value. The first CONVERT function shown here, for example, converts the PaymentDate column to a character value with the format mm/dd/yy. And the second CONVERT function converts the PaymentTotal column to a variable-length character value that's formatted with commas. These functions are included in a string expression that concatenates their return values with the InvoiceNumber column and three literal values.

The third example uses two functions that work with dates. The first one, GETDATE, returns the current date. Notice that although this function doesn't accept any parameters, the parentheses are still included. The second function, DATEDIFF, gets the difference between two date values. This function requires three parameters. The first one specifies the units in which the result will be expressed. In this example, the function will return the number of days between the two dates. The second and third parameters specify the start date and the end date. Here, the second parameter is the invoice date and the third parameter is the current date, which is obtained using the GETDATE function.

A SELECT statement that uses the LEFT function

```
SELECT VendorContactFName, VendorContactLName,
       LEFT(VendorContactFName, 1) +
       LEFT(VendorContactLName, 1) AS Initials
FROM Vendors;
```

	VendorContactFName	VendorContactLName	Initials
1	Francesco	Alberto	FA
2	Ania	Irvin	AI
3	Lukas	Liana	LL

A SELECT statement that uses the CONVERT function

```
SELECT 'Invoice: #' + InvoiceNumber
      + ', dated ' + CONVERT(char(8), PaymentDate, 1)
      + ' for $' + CONVERT(varchar(9), PaymentTotal, 1)
FROM Invoices;
```

	(No column name)
1	Invoice: #989319-457, dated 11/07/19 for \$3,813.33
2	Invoice: #263253241, dated 11/14/19 for \$40.20

A SELECT statement that computes the age of an invoice

```
SELECT InvoiceDate,
       GETDATE() AS 'Today''s Date',
       DATEDIFF(day, InvoiceDate, GETDATE()) AS Age
FROM Invoices;
```

	InvoiceDate	Today's Date	Age
1	2020-02-02	2020-03-01 12:35:44.310	28
2	2020-02-01	2020-03-01 12:35:44.310	29
3	2020-01-31	2020-03-01 12:35:44.310	30

Description

- An expression can include any of the *functions* that are supported by SQL Server. A function performs an operation and returns a value.
- A function consists of the function name, followed by a set of parentheses that contains any *parameters*, or *arguments*, required by the function. If a function requires two or more arguments, you separate them with commas.
- For more information on using functions, see chapter 9.

How to use the DISTINCT keyword to eliminate duplicate rows

By default, all of the rows in the base table that satisfy the search condition you specify in the WHERE clause are included in the result set. In some cases, though, that means that the result set will contain duplicate rows, or rows whose column values are identical. If that's not what you want, you can include the DISTINCT keyword in the SELECT clause to eliminate the duplicate rows.

Figure 3-8 illustrates how this works. Here, both SELECT statements retrieve the VendorCity and VendorState columns from the Vendors table. The first statement, however, doesn't include the DISTINCT keyword. Because of that, the same city and state can appear in the result set multiple times. In the results shown in this figure, for example, you can see that Anaheim CA occurs twice and Boston MA occurs three times. By contrast, the second statement includes the DISTINCT keyword, so each city/state combination is included only once.

Notice that, in addition to including the DISTINCT keyword, the second statement doesn't include the ORDER BY clause. That's because when you include the DISTINCT keyword, the result set is automatically sorted by its first column. In this case, that's the same column that was used to sort the result set returned by the first statement.

A SELECT statement that returns all rows

```
SELECT VendorCity, VendorState  
FROM Vendors  
ORDER BY VendorCity;
```

	VendorCity	VendorState
1	Anaheim	CA
2	Anaheim	CA
3	Ann Arbor	MI
4	Auburn Hills	MI
5	Boston	MA
6	Boston	MA
7	Boston	MA
8	Brea	CA

(122 rows)

A SELECT statement that eliminates duplicate rows

```
SELECT DISTINCT VendorCity, VendorState  
FROM Vendors;
```

	VendorCity	VendorState
1	Anaheim	CA
2	Ann Arbor	MI
3	Auburn Hills	MI
4	Boston	MA
5	Brea	CA
6	Carol Stream	IL
7	Charlotte	NC
8	Chicago	IL

(53 rows)

Description

- The DISTINCT keyword prevents duplicate (identical) rows from being included in the result set. It also causes the result set to be sorted by its first column.
- The ALL keyword causes all rows matching the search condition to be included in the result set, regardless of whether rows are duplicated. Since this is the default, it's a common practice to omit the ALL keyword.
- To use the DISTINCT or ALL keyword, code it immediately after the SELECT keyword as shown above.

Figure 3-8 How to use the DISTINCT keyword to eliminate duplicate rows

How to use the TOP clause to return a subset of selected rows

In addition to eliminating duplicate rows, you can limit the number of rows that are retrieved by a SELECT statement. To do that, you use the TOP clause.

Figure 3-9 shows you how.

You can use the TOP clause in one of two ways. First, you can use it to retrieve a specific number of rows from the beginning, or top, of the result set. To do that, you code the TOP keyword followed by an integer value that specifies the number of rows to be returned. This is illustrated in the first example in this figure. Here, only five rows are returned. Notice that this statement also includes an ORDER BY clause that sorts the rows by the InvoiceTotal column in descending sequence. That way, the invoices with the highest invoice totals will be returned.

You can also use the TOP clause to retrieve a specific percent of the rows in the result set. To do that, you include the PERCENT keyword as shown in the second example. In this case, the result set includes six rows, which is five percent of the total of 114 rows.

By default, the TOP clause causes the exact number or percent of rows you specify to be retrieved. However, if additional rows match the values in the last row, you can include those additional rows by including WITH TIES in the TOP clause. This is illustrated in the third example in this figure. Here, the SELECT statement says to retrieve the top five rows from a result set that includes the VendorID and InvoiceDate columns sorted by the InvoiceDate column. As you can see, however, the result set includes six rows instead of five. That's because WITH TIES is included in the TOP clause, and the columns in the sixth row have the same values as the columns in the fifth row.

A SELECT statement with a TOP clause

```
SELECT TOP 5 VendorID, InvoiceTotal
FROM Invoices
ORDER BY InvoiceTotal DESC;
```

	VendorID	InvoiceTotal
1	110	37966.19
2	110	26881.40
3	110	23517.58
4	72	21842.00
5	110	20551.18

A SELECT statement with a TOP clause and the PERCENT keyword

```
SELECT TOP 5 PERCENT VendorID, InvoiceTotal
FROM Invoices
ORDER BY InvoiceTotal DESC;
```

	VendorID	InvoiceTotal
1	110	37966.19
2	110	26881.40
3	110	23517.58
4	72	21842.00
5	110	20551.18
6	110	10976.06

A SELECT statement with a TOP clause and the WITH TIES keyword

```
SELECT TOP 5 WITH TIES VendorID, InvoiceDate
FROM Invoices
ORDER BY InvoiceDate ASC;
```

	VendorID	InvoiceDate
1	122	2019-10-08
2	123	2019-10-10
3	123	2019-10-13
4	123	2019-10-16
5	123	2019-10-16
6	123	2019-10-16

Description

- You can use the TOP clause within a SELECT clause to limit the number of rows included in the result set. When you use this clause, the first n rows that meet the search condition are included, where n is an integer.
- If you include PERCENT, the first n percent of the selected rows are included in the result set.
- If you include WITH TIES, additional rows will be included if their values match, or *tie*, the values of the last row.
- You should include an ORDER BY clause whenever you use the TOP keyword. Otherwise, the rows in the result set will be in no particular sequence.

Figure 3-9 How to use the TOP clause to return a subset of selected rows

How to code the WHERE clause

Earlier in this chapter, I mentioned that to improve performance, you should code your SELECT statements so they retrieve only the columns you need. That goes for retrieving rows too: The fewer rows you retrieve, the more efficient the statement will be. Because of that, you'll almost always include a WHERE clause on your SELECT statements with a search condition that filters the rows in the base table so that only the rows you need are retrieved. In the topics that follow, you'll learn a variety of ways to code this clause.

How to use comparison operators

Figure 3-10 shows you how to use the *comparison operators* in the search condition of a WHERE clause. As you can see in the syntax summary at the top of this figure, you use a comparison operator to compare two expressions. If the result of the comparison is True, the row being tested is included in the query results.

The examples in this figure show how to use some of the comparison operators. The first WHERE clause, for example, uses the equal operator (=) to retrieve only those rows whose VendorState column have a value of IA. Since the state code is a string literal, it must be included in single quotes. By contrast, the numeric literal used in the second WHERE clause is not enclosed in quotes. This clause uses the greater than (>) operator to retrieve only those rows that have a balance due greater than zero.

The third WHERE clause illustrates another way to retrieve all the invoices with a balance due. Like the second clause, it uses the greater than operator. Instead of comparing the balance due to a value of zero, however, it compares the invoice total to the total of the payments and credits that have been applied to the invoice.

The fourth WHERE clause illustrates how you can use comparison operators other than the equal operator with string data. In this example, the less than operator (<) is used to compare the value of the VendorName column to a literal string that contains the letter M. That will cause the query to return all vendors with names that begin with the letters A through L.

You can also use the comparison operators with date literals, as illustrated by the fifth and sixth WHERE clauses. The fifth clause will retrieve rows with invoice dates on or before January 31, 2020, and the sixth clause will retrieve rows with invoice dates on or after January 1, 2020. Like string literals, date literals must be enclosed in single quotes. In addition, you can use different formats to specify dates as shown by the two date literals shown in this figure. You'll learn more about the acceptable date formats in chapter 8.

The last WHERE clause shows how you can test for a not equal condition. To do that, you code a less than sign followed by a greater than sign. In this case, only rows with a credit total that's not equal to zero will be retrieved.

Whenever possible, you should compare expressions that have similar data types. If you attempt to compare expressions that have different data types, SQL Server may implicitly convert the data type for you. Often, this implicit

The syntax of the WHERE clause with comparison operators

```
WHERE expression_1 operator expression_2
```

The comparison operators

=	Equal
>	Greater than
<	Less than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal

Examples of WHERE clauses that retrieve...

Vendors located in Iowa

```
WHERE VendorState = 'IA'
```

Invoices with a balance due (two variations)

```
WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0  
WHERE InvoiceTotal > PaymentTotal + CreditTotal
```

Vendors with names from A to L

```
WHERE VendorName < 'M'
```

Invoices on or before a specified date

```
WHERE InvoiceDate <= '2020-01-31'
```

Invoices on or after a specified date

```
WHERE InvoiceDate >= '1/1/20'
```

Invoices with credits that don't equal zero

```
WHERE CreditTotal <> 0
```

Description

- You can use a comparison operator to compare any two expressions that result in like data types. Although unlike data types may be converted to data types that can be compared, the comparison may produce unexpected results.
- If a comparison results in a True value, the row being tested is included in the result set. If it's False or Unknown, the row isn't included.
- To use a string literal or a *date literal* in a comparison, enclose it in quotes. To use a numeric literal, enter the number without quotes.
- Character comparisons performed on SQL Server databases are not case-sensitive. So, for example, 'CA' and 'Ca' are considered equivalent.

conversion is acceptable. However, implicit conversions will occasionally yield unexpected results. In that case, you can use the CONVERT function you saw earlier in this chapter or the CAST function you'll learn about in chapter 8 to explicitly convert data types so the comparison yields the results you want.

How to use the AND, OR, and NOT logical operators

Figure 3-11 shows how to use *logical operators* in a WHERE clause. You can use the AND and OR operators to combine two or more search conditions into a *compound condition*. And you can use the NOT operator to negate a search condition. The examples in this figure illustrate how these operators work.

The first two examples illustrate the difference between the AND and OR operators. When you use the AND operator, both conditions must be true. So, in the first example, only those vendors in New Jersey whose year-to-date purchases are greater than 200 are retrieved from the Vendors table. When you use the OR operator, though, only one of the conditions must be true. So, in the second example, all the vendors from New Jersey and all the vendors whose year-to-date purchases are greater than 200 are retrieved.

The third example shows a compound condition that uses two NOT operators. As you can see, this expression is somewhat difficult to understand. Because of that, and because using the NOT operator can reduce system performance, you should avoid using this operator whenever possible. The fourth example in this figure, for instance, shows how the search condition in the third example can be rephrased to eliminate the NOT operator. Notice that the condition in the fourth example is much easier to understand.

The last two examples in this figure show how the order of precedence for the logical operators and the use of parentheses affect the result of a search condition. By default, the NOT operator is evaluated first, followed by AND and then OR. However, you can use parentheses to override the order of precedence or to clarify a logical expression, just as you can with arithmetic expressions. In the next to last example, for instance, no parentheses are used, so the two conditions connected by the AND operator are evaluated first. In the last example, though, parentheses are used so the two conditions connected by the OR operator are evaluated first. If you take a minute to review the results shown in this figure, you should be able to see how these two conditions differ.

The syntax of the WHERE clause with logical operators

```
WHERE [NOT] search_condition_1 {AND|OR} [NOT] search_condition_2 ...
```

Examples of queries using logical operators

A search condition that uses the AND operator

```
WHERE VendorState = 'NJ' AND YTDPurchases > 200
```

A search condition that uses the OR operator

```
WHERE VendorState = 'NJ' OR YTDPurchases > 200
```

A search condition that uses the NOT operator

```
WHERE NOT (InvoiceTotal >= 5000 OR NOT InvoiceDate <= '2020-02-01')
```

The same condition rephrased to eliminate the NOT operator

```
WHERE InvoiceTotal < 5000 AND InvoiceDate <= '2020-02-01'
```

A compound condition without parentheses

```
WHERE InvoiceDate > '01/01/2020'  
    OR InvoiceTotal > 500  
    AND InvoiceTotal - PaymentTotal - CreditTotal > 0
```

	InvoiceNumber	InvoiceDate	InvoiceTotal	BalanceDue
1	263253265	2020-01-02	26.25	0.00
2	203339-13	2020-01-05	17.50	0.00
3	111-92R-10093	2020-01-06	39.77	0.00
4	963253258	2020-01-06	111.00	0.00

(34 rows)

The same compound condition with parentheses

```
WHERE (InvoiceDate > '01/01/2020'  
    OR InvoiceTotal > 500)  
    AND InvoiceTotal - PaymentTotal - CreditTotal > 0
```

	InvoiceNumber	InvoiceDate	InvoiceTotal	BalanceDue
1	39104	2020-01-10	85.31	85.31
2	963253264	2020-01-18	52.25	52.25
3	31361833	2020-01-21	579.42	579.42
4	263253268	2020-01-21	59.97	59.97

(11 rows)

Description

- You can use the AND and OR *logical operators* to create *compound conditions* that consist of two or more conditions. You use the AND operator to specify that the search must satisfy both of the conditions, and you use the OR operator to specify that the search must satisfy at least one of the conditions.
- You can use the NOT operator to negate a condition. Because this operator can make the search condition difficult to read, you should rephrase the condition if possible so it doesn't use NOT.
- When SQL Server evaluates a compound condition, it evaluates the operators in this sequence: (1) NOT, (2) AND, and (3) OR. You can use parentheses to override this order of precedence or to clarify the sequence in which the operations will be evaluated.

Figure 3-11 How to use the AND, OR, and NOT logical operators

How to use the IN operator

Figure 3-12 shows how to code a WHERE clause that uses the IN operator. When you use this operator, the value of the test expression is compared with the list of expressions in the IN phrase. If the test expression is equal to one of the expressions in the list, the row is included in the query results. This is illustrated by the first example in this figure, which will return all rows whose TermsID column is equal to 1, 3, or 4.

You can also use the NOT operator with the IN phrase to test for a value that's not in a list of expressions. This is illustrated by the second example in this figure. In this case, only those vendors who are not in California, Nevada, or Oregon are retrieved.

If you look at the syntax of the IN phrase shown at the top of this figure, you'll see that you can code a *subquery* in place of a list of expressions. Subqueries are a powerful tool that you'll learn about in detail in chapter 6. For now, though, you should know that a subquery is simply a SELECT statement within another statement. In the third example in this figure, for instance, a subquery is used to return a list of VendorID values for vendors who have invoices dated December 1, 2019. Then, the WHERE clause retrieves a vendor row only if the vendor is in that list. Note that for this to work, the subquery must return a single column, in this case, VendorID.

The syntax of the WHERE clause with an IN phrase

```
WHERE test_expression [NOT] IN ({subquery|expression_1 [, expression_2]...})
```

Examples of the IN phrase

An IN phrase with a list of numeric literals

```
WHERE TermsID IN (1, 3, 4)
```

An IN phrase preceded by NOT

```
WHERE VendorState NOT IN ('CA', 'NV', 'OR')
```

An IN phrase with a subquery

```
WHERE VendorID IN  
(SELECT VendorID  
FROM Invoices  
WHERE InvoiceDate = '2019-12-01')
```

Description

- You can use the IN phrase to test whether an expression is equal to a value in a list of expressions. Each of the expressions in the list must evaluate to the same type of data as the test expression.
- The list of expressions can be coded in any order without affecting the order of the rows in the result set.
- You can use the NOT operator to test for an expression that's not in the list of expressions.
- You can also compare the test expression to the items in a list returned by a *subquery* as illustrated by the third example above. You'll learn more about coding subqueries in chapter 6.

How to use the BETWEEN operator

Figure 3-13 shows how to use the BETWEEN operator in a WHERE clause. When you use this operator, the value of a test expression is compared to the range of values specified in the BETWEEN phrase. If the value falls within this range, the row is included in the query results.

The first example in this figure shows a simple WHERE clause that uses the BETWEEN operator. It retrieves invoices with invoice dates between January 1, 2020 and January 31, 2020. Note that the range is inclusive, so invoices with invoice dates of January 1 and January 31 are included in the results.

The second example shows how to use the NOT operator to select rows that are not within a given range. In this case, vendors with zip codes that aren't between 93600 and 93799 are included in the results.

The third example shows how you can use a calculated value in the test expression. Here, the PaymentTotal and CreditTotal columns are subtracted from the InvoiceTotal column to give the balance due. Then, this value is compared to the range specified in the BETWEEN phrase.

The last example shows how you can use calculated values in the BETWEEN phrase. Here, the first value is the result of the GETDATE function, and the second value is the result of the GETDATE function plus 30 days. So the query results will include all those invoices that are due between the current date and 30 days from the current date.

The syntax of the WHERE clause with a BETWEEN phrase

```
WHERE test_expression [NOT] BETWEEN begin_expression AND end_expression
```

Examples of the BETWEEN phrase

A BETWEEN phrase with literal values

```
WHERE InvoiceDate BETWEEN '2020-01-01' AND '2020-01-31'
```

A BETWEEN phrase preceded by NOT

```
WHERE VendorZipCode NOT BETWEEN 93600 AND 93799
```

A BETWEEN phrase with a test expression coded as a calculated value

```
WHERE InvoiceTotal - PaymentTotal - CreditTotal BETWEEN 200 AND 500
```

A BETWEEN phrase with the upper and lower limits coded as calculated values

```
WHERE InvoiceDueDate BETWEEN GetDate() AND GetDate() + 30
```

Description

- You can use the BETWEEN phrase to test whether an expression falls within a range of values. The lower limit must be coded as the first expression, and the upper limit must be coded as the second expression. Otherwise, the result set will be empty.
- The two expressions used in the BETWEEN phrase for the range of values are inclusive. That is, the result set will include values that are equal to the upper or lower limit.
- You can use the NOT operator to test for an expression that's not within the given range.

Warning about date comparisons

- All columns that have the datetime2 data type include both a date and time, and so does the value returned by the GETDATE function. But when you code a date literal like '2020-01-01', the time defaults to 00:00:00 on a 24-hour clock, or 12:00 AM (midnight). As a result, a date comparison may not yield the results you expect. For instance, January 31, 2020 at 2:00 PM isn't between '2020-01-01' and '2020-01-31'.
- To learn more about date comparisons, please see chapter 9.

How to use the LIKE operator

One final operator you can use in a search condition is the LIKE operator shown in figure 3-14. You use this operator along with the *wildcards* shown at the top of this figure to specify a *string pattern*, or *mask*, you want to match. The examples shown in this figure illustrate how this works.

In the first example, the LIKE phrase specifies that all vendors in cities that start with the letters SAN should be included in the query results. Here, the percent sign (%) indicates that any characters can follow these three letters. So San Diego and Santa Ana are both included in the results.

The second example selects all vendors whose vendor name starts with the letters COMPU, followed by any one character, the letters ER, and any characters after that. Two vendor names that match that pattern are Compuserve and Computerworld.

The third example searches the values in the VendorContactLName column for a name that can be spelled two different ways: Damien or Damion. To do that, the mask specifies the two possible characters in the fifth position, E and O, within brackets.

The fourth example uses brackets to specify a range of values. In this case, the VendorState column is searched for values that start with the letter N and end with any letter from A to J. That excludes states like Nevada (NV) and New York (NY).

The fifth example shows how to use the caret (^) to exclude one or more characters from the pattern. Here, the pattern says that the value in the VendorState column must start with the letter N, but must not end with the letters K through Y. This produces the same result as the previous statement.

The last example in this figure shows how to use the NOT operator with a LIKE phrase. The condition in this example tests the VendorZipCode column for values that don't start with the numbers 1 through 9. The result is all zip codes that start with the number 0.

The LIKE operator provides a powerful technique for finding information in a database that can't be found using any other technique. Keep in mind, however, that this technique requires a lot of overhead, so it can reduce system performance. For this reason, you should avoid using the LIKE operator in production SQL code whenever possible.

If you need to search the text that's stored in your database, a better option is to use the *Full-Text Search* feature that's provided by SQL Server. This feature provides more powerful and flexible ways to search for text, and it performs more efficiently than the LIKE operator. However, Full-Text Search is an advanced feature that requires some setup and administration and is too complex to explain here. For more information, you can look up "full-text search" in the SQL Server documentation.

The syntax of the WHERE clause with a LIKE phrase

```
WHERE match_expression [NOT] LIKE pattern
```

Wildcard symbols

Symbol	Description
%	Matches any string of zero or more characters.
_	Matches any single character.
[]	Matches a single character listed within the brackets.
[-]	Matches a single character within the given range.
[^]	Matches a single character not listed after the caret.

WHERE clauses that use the LIKE operator

Example	Results that match the mask
WHERE VendorCity LIKE 'SAN%'	"San Diego" and "Santa Ana"
WHERE VendorName LIKE 'COMPU_ER%'	"Compuserve" and "Computerworld"
WHERE VendorContactLName LIKE 'DAMI[EO]N'	"Damien" and "Damion"
WHERE VendorState LIKE 'N[A-J]'	"NC" and "NJ" but not "NV" or "NY"
WHERE VendorState LIKE 'N[^K-Y]'	"NC" and "NJ" but not "NV" or "NY"
WHERE VendorZipCode NOT LIKE '[1-9]%'	"02107" and "08816"

Description

- You use the LIKE operator to retrieve rows that match a *string pattern*, called a *mask*. Within the mask, you can use special characters, called *wildcards*, that determine which values in the column satisfy the condition.
- You can use the NOT keyword before the LIKE keyword. Then, only those rows with values that don't match the string pattern will be included in the result set.
- Most LIKE phrases will significantly degrade performance compared to other types of searches, so use them only when necessary.

How to use the IS NULL clause

In chapter 1, you learned that a column can contain a *null value*. A null isn't the same as zero, a blank string that contains one or more spaces (' '), or an empty string (""). Instead, a null value indicates that the data is not applicable, not available, or unknown. When you allow null values in one or more columns, you need to know how to test for them in search conditions. To do that, you can use the IS NULL clause as shown in figure 3-15.

This figure uses a table named NullSample to illustrate how to search for null values. This table contains two columns. The first column, InvoiceID, is an identity column. The second column, InvoiceTotal, contains the total for the invoice, which can be a null value. As you can see in the first example, the invoice with InvoiceID 3 contains a null value.

The second example in this figure shows what happens when you retrieve all the invoices with invoice totals equal to zero. Notice that the row that has a null invoice total isn't included in the result set. Likewise, it isn't included in the result set that contains all the invoices with invoices totals that aren't equal to zero, as illustrated by the third example. Instead, you have to use the IS NULL clause to retrieve rows with null values, as shown in the fourth example.

You can also use the NOT operator with the IS NULL clause as illustrated in the last example in this figure. When you use this operator, all of the rows that don't contain null values are included in the query results.

The syntax of the WHERE clause with the IS NULL clause

```
WHERE expression IS [NOT] NULL
```

The contents of the NullSample table

```
SELECT *
FROM NullSample;
```

	InvoiceID	InvoiceTotal
1	1	125.00
2	2	0.00
3	3	NULL
4	4	2199.99
5	5	0.00

A SELECT statement that retrieves rows with zero values

```
SELECT *
FROM NullSample
WHERE InvoiceTotal = 0;
```

	InvoiceID	InvoiceTotal
1	2	0.00
2	5	0.00

A SELECT statement that retrieves rows with non-zero values

```
SELECT *
FROM NullSample
WHERE InvoiceTotal <> 0;
```

	InvoiceID	InvoiceTotal
1	1	125.00
2	4	2199.99

A SELECT statement that retrieves rows with null values

```
SELECT *
FROM NullSample
WHERE InvoiceTotal IS NULL;
```

	InvoiceID	InvoiceTotal
1	3	NULL

A SELECT statement that retrieves rows without null values

```
SELECT *
FROM NullSample
WHERE InvoiceTotal IS NOT NULL;
```

	InvoiceID	InvoiceTotal
1	1	125.00
2	2	0.00
3	4	2199.99
4	5	0.00

Description

- A *null value* represents a value that's unknown, unavailable, or not applicable. It isn't the same as a zero, a blank space (' '), or an empty string ('').
- To test for a null value, you can use the IS NULL clause. You can also use the NOT keyword with this clause to test for values that aren't null.
- The definition of each column in a table indicates whether or not it can store null values. Before you work with a table, you should identify those columns that allow null values so you can accommodate them in your queries.

How to code the ORDER BY clause

The ORDER BY clause specifies the sort order for the rows in a result set. In most cases, you can use column names from the base table to specify the sort order as you saw in some of the examples earlier in this chapter. However, you can also use other techniques to sort the rows in a result set. In addition, you can use the OFFSET and FETCH clauses of the ORDER BY clause to retrieve a range of rows from the sorted result set.

How to sort a result set by a column name

Figure 3-16 presents the expanded syntax of the ORDER BY clause. As you can see, you can sort by one or more expressions in either ascending or descending sequence. This is illustrated by the three examples in this figure.

The first two examples show how to sort the rows in a result set by a single column. In the first example, the rows in the Vendors table are sorted in ascending sequence by the VendorName column. Since ascending is the default sequence, the ASC keyword is omitted. In the second example, the rows are sorted by the VendorName column in descending sequence.

To sort by more than one column, you simply list the names in the ORDER BY clause separated by commas as shown in the third example. Here, the rows in the Vendors table are first sorted by the VendorState column in ascending sequence. Then, within each state, the rows are sorted by the VendorCity column in ascending sequence. Finally, within each city, the rows are sorted by the VendorName column in ascending sequence. This can be referred to as a *nested sort* because one sort is nested within another.

Although all of the columns in this example are sorted in ascending sequence, you should know that doesn't have to be the case. For example, I could have sorted by the VendorName column in descending sequence like this:

```
ORDER BY VendorState, VendorCity, VendorName DESC
```

Note that the DESC keyword in this example applies only to the VendorName column. The VendorState and VendorCity columns are still sorted in ascending sequence.

The expanded syntax of the ORDER BY clause

```
ORDER BY expression [ASC|DESC] [, expression [ASC|DESC]] ...
```

An ORDER BY clause that sorts by one column in ascending sequence

```
SELECT VendorName,
       VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors
ORDER BY VendorName;
```

	VendorName	Address
1	Abbey Office Furnishings	Fresno, CA 93722
2	American Booksellers Assoc	Tarrytown, NY 10591
3	American Express	Los Angeles, CA 90096

An ORDER BY clause that sorts by one column in descending sequence

```
SELECT VendorName,
       VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors
ORDER BY VendorName DESC;
```

	VendorName	Address
1	Zylka Design	Fresno, CA 93711
2	Zip Print & Copy Center	Fresno, CA 93777
3	Zee Medical Service Co	Washington, IA 52353

An ORDER BY clause that sorts by three columns

```
SELECT VendorName,
       VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors
ORDER BY VendorState, VendorCity, VendorName;
```

	VendorName	Address
1	AT&T	Phoenix, AZ 85062
2	Computer Library	Phoenix, AZ 85023
3	Wells Fargo Bank	Phoenix, AZ 85038
4	Aztek Label	Anaheim, CA 92807
5	Blue Shield of California	Anaheim, CA 92850
6	Diversified Printing & Pub	Brea, CA 92621
7	Abbey Office Furnishings	Fresno, CA 93722
8	ASC Signs	Fresno, CA 93703
9	BFI Industries	Fresno, CA 93792

Description

- The ORDER BY clause specifies how you want the rows in the result set sorted. You can sort by one or more columns, and you can sort each column in either ascending (ASC) or descending (DESC) sequence. ASC is the default.
- By default, in an ascending sort, nulls appear first in the sort sequence, followed by special characters, then numbers, then letters. Although you can change this sequence, that's beyond the scope of this book.
- You can sort by any column in the base table regardless of whether it's included in the SELECT clause. The exception is if the query includes the DISTINCT keyword. Then, you can only sort by columns included in the SELECT clause.

Figure 3-16 How to sort a result set by a column name

How to sort a result set by an alias, an expression, or a column number

Figure 3-17 presents three more techniques you can use to specify sort columns. First, you can use a column alias that's defined in the SELECT clause. The first SELECT statement in this figure, for example, sorts by a column named Address, which is an alias for the concatenation of the VendorCity, VendorState, and VendorZipCode columns. Within the Address column, the result set is also sorted by the VendorName column.

You can also use an arithmetic or string expression in the ORDER BY clause, as illustrated by the second example in this figure. Here, the expression consists of the VendorContactLName column concatenated with the VendorContactFName column. Here, neither of these columns is included in the SELECT clause.

Although SQL Server allows this seldom-used coding technique, many other database systems do not.

The last example in this figure shows how you can use column numbers to specify a sort order. To use this technique, you code the number that corresponds to the column of the result set, where 1 is the first column, 2 is the second column, and so on. In this example, the ORDER BY clause sorts the result set by the second column, which contains the concatenated address, then by the first column, which contains the vendor name. The result set returned by this statement is the same as the result set returned by the first statement. Notice, however, that the statement that uses column numbers is more difficult to read because you have to look at the SELECT clause to see what columns the numbers refer to. In addition, if you add or remove columns from the SELECT clause, you may also have to change the ORDER BY clause to reflect the new column positions. As a result, you should avoid using this technique.

An ORDER BY clause that uses an alias

```
SELECT VendorName,  
       VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address  
FROM Vendors  
ORDER BY Address, VendorName;
```

	VendorName	Address
1	Aztek Label	Anaheim, CA 92807
2	Blue Shield of California	Anaheim, CA 92850
3	Malloy Lithographing Inc	Ann Arbor, MI 48106

An ORDER BY clause that uses an expression

```
SELECT VendorName,  
       VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address  
FROM Vendors  
ORDER BY VendorContactLName + VendorContactFName;
```

	VendorName	Address
1	Dristas Groom & McCormick	Fresno, CA 93720
2	Internal Revenue Service	Fresno, CA 93888
3	US Postal Service	Madison, WI 53707

An ORDER BY clause that uses column positions

```
SELECT VendorName,  
       VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address  
FROM Vendors  
ORDER BY 2, 1;
```

	VendorName	Address
1	Aztek Label	Anaheim, CA 92807
2	Blue Shield of California	Anaheim, CA 92850
3	Malloy Lithographing Inc	Ann Arbor, MI 48106

Description

- The ORDER BY clause can include a column alias that's specified in the SELECT clause.
- The ORDER BY clause can include any valid expression. The expression can refer to any column in the base table, even if it isn't included in the result set.
- The ORDER BY clause can use numbers to specify the columns to use for sorting. In that case, 1 represents the first column in the result set, 2 represents the second column, and so on.

Figure 3-17 How to sort a result set by an alias, an expression, or a column number

How to retrieve a range of selected rows

Earlier in this chapter, you saw how to use the TOP clause to return a subset of the rows selected by a query. When you use this clause, the rows are always returned from the beginning of the result set. By contrast, the OFFSET and FETCH clauses let you return a subset of rows starting anywhere in a sorted result set. Figure 3-18 illustrates how these clauses work.

The first example in this figure shows how to use the OFFSET and FETCH clauses to retrieve rows from the beginning of a result set. In this case, the first five rows are retrieved. By contrast, the second example retrieves ten rows from the result set starting with the eleventh row. Notice that the FETCH clause in the first example uses the FIRST keyword, and the FETCH clause in the second example uses the NEXT keyword. Although these keywords are interchangeable, they're typically used as shown here.

You can also return all of the rows to the end of a result set after skipping the specified number of rows. To do that, you just omit the FETCH clause. For instance, if you omitted this clause from the second example in this figure, rows 11 through the end of the result set would be retrieved.

The OFFSET and FETCH clauses are most useful when a client application needs to retrieve and process one page of data at a time. For example, suppose an application can work with up to 20 rows of a result set at a time. Then, the first query would retrieve rows 1 through 20, the second query would retrieve rows 21 through 40, and so on.

Because a new result set is created each time a query is executed, the client application must make sure that the result set doesn't change between queries. For example, if after retrieving the first 20 rows of a result set as described above one of those rows is deleted, the row that would have been the 21st row now becomes the 20th row. Because of that, this row isn't included when the next 20 rows are retrieved. To prevent this problem, an application can execute all of the queries within a transaction whose isolation level is set to either SNAPSHOT or SERIALIZABLE. For information on how transactions and isolation levels work within SQL Server, see chapter 16.

The syntax of the ORDER BY clause for retrieving a range of rows

```
ORDER BY order_by_list
    OFFSET offset_row_count {ROW|ROWS}
    [FETCH {FIRST|NEXT} fetch_row_count {ROW|ROWS} ONLY]
```

An ORDER BY clause that retrieves the first five rows

```
SELECT VendorID, InvoiceTotal
FROM Invoices
ORDER BY InvoiceTotal DESC
OFFSET 0 ROWS
FETCH FIRST 5 ROWS ONLY;
```

	VendorID	InvoiceTotal
1	110	37966.19
2	110	26881.40
3	110	23517.58
4	72	21842.00
5	110	20551.18

An ORDER BY clause that retrieves rows 11 through 20

```
SELECT VendorName, VendorCity, VendorState, VendorZipCode
FROM Vendors
WHERE VendorState = 'CA'
ORDER BY VendorCity
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY;
```

	VendorName	VendorCity	VendorState	VendorZipCode
1	Robbins Mobile Lock And Key	Fresno	CA	93726
2	BFI Industries	Fresno	CA	93792
3	California Data Marketing	Fresno	CA	93721
4	Yale Industrial Trucks-Fresno	Fresno	CA	93706
5	Costco	Fresno	CA	93711
6	Graylift	Fresno	CA	93745
7	Shields Design	Fresno	CA	93728
8	Fresno County Tax Collector	Fresno	CA	93715
9	Gary McKeighan Insurance	Fresno	CA	93711
10	Ph Photographic Services	Fresno	CA	93726

Description

- The OFFSET clause specifies the number of rows that should be skipped before rows are returned from the result set.
- The FETCH clause specifies the number of rows that should be retrieved after skipping the specified number of rows. If you omit the FETCH clause, all of the rows to the end of the result set are retrieved.
- The number of rows to be skipped and retrieved can be specified as an integer or an expression that results in an integer that is greater than or equal to zero.
- The OFFSET and FETCH clauses are most useful when a client application needs to retrieve one page of data at a time.

Figure 3-18 How to retrieve a range of selected rows

Perspective

The goal of this chapter has been to teach you the basic skills for coding SELECT statements. You'll use these skills in almost every SELECT statement you code. As you'll see in the chapters that follow, however, there's a lot more to coding SELECT statements than what's presented here. In the next three chapters, then, you'll learn additional skills for coding SELECT statements. When you complete those chapters, you'll know everything you need to know about retrieving data from a SQL Server database.

Terms

keyword	order of precedence
filter	function
Boolean expression	parameter
predicate	argument
expression	date literal
column alias	comparison operator
substitute name	logical operator
string expression	compound condition
concatenate	subquery
concatenation operator	string pattern
literal value	mask
string literal	wildcard
string constant	Full-Text Search
arithmetic expression	null value
arithmetic operator	nested sort

Exercises

1. Write a SELECT statement that returns three columns from the Vendors table: VendorContactFName, VendorContactLName, and VendorName. Sort the result set by last name, then by first name.
2. Write a SELECT statement that returns four columns from the Invoices table, named Number, Total, Credits, and Balance:

Number	Column alias for the InvoiceNumber column
Total	Column alias for the InvoiceTotal column
Credits	Sum of the PaymentTotal and CreditTotal columns
Balance	InvoiceTotal minus the sum of PaymentTotal and CreditTotal

Use the AS keyword to assign column aliases.

3. Write a SELECT statement that returns one column from the Vendors table named Full Name. Create this column from the VendorContactFName and VendorContactLName columns. Format it as follows: last name, comma, first name (for example, “Doe, John”). Sort the result set by last name, then by first name.
4. Write a SELECT statement that returns three columns:

InvoiceTotal	From the Invoices table
10%	10% of the value of InvoiceTotal
Plus 10%	The value of InvoiceTotal plus 10%

(For example, if InvoiceTotal is 100.0000, 10% is 10.0000, and Plus 10% is 110.0000.) Only return those rows with a balance due greater than 1000. Sort the result set by InvoiceTotal, with the largest invoice first.

5. Modify the solution to exercise 2 to filter for invoices with an InvoiceTotal that's greater than or equal to \$500 but less than or equal to \$10,000.
6. Modify the solution to exercise 3 to filter for contacts whose last name begins with the letter A, B, C, or E.
7. Write a SELECT statement that determines whether the PaymentDate column of the Invoices table has any invalid values. To be valid, PaymentDate must be a null value if there's a balance due and a non-null value if there's no balance due. Code a compound condition in the WHERE clause that tests for these conditions.

4

How to retrieve data from two or more tables

In the last chapter, you learned how to create result sets that contain data from a single table. Now, this chapter will show you how to create result sets that contain data from two or more tables. To do that, you can use either a join or a union.

How to work with inner joins.....	126
How to code an inner join.....	126
When and how to use correlation names.....	128
How to work with tables from different databases.....	130
How to use compound join conditions	132
How to use a self-join	134
Inner joins that join more than two tables	136
How to use the implicit inner join syntax.....	138
How to work with outer joins.....	140
How to code an outer join.....	140
Outer join examples.....	142
Outer joins that join more than two tables	144
Other skills for working with joins.....	146
How to combine inner and outer joins	146
How to use cross joins	148
How to work with unions	150
The syntax of a union	150
Unions that combine data from different tables	150
Unions that combine data from the same table	152
How to use the EXCEPT and INTERSECT operators	154
Perspective	156

How to work with inner joins

A *join* lets you combine columns from two or more tables into a single result set. In the topics that follow, you'll learn how to use the most common type of join, an *inner join*. You'll learn how to use other types of joins later in this chapter.

How to code an inner join

Figure 4-1 presents the *explicit syntax* for coding an inner join. As you'll see later in this chapter, SQL Server also provides an implicit syntax that you can use to code inner joins. However, the syntax shown in this figure is the one you'll use most often.

To join data from two tables, you code the names of the two tables in the FROM clause along with the JOIN keyword and an ON phrase that specifies the *join condition*. The join condition indicates how the two tables should be compared. In most cases, they're compared based on the relationship between the primary key of the first table and a foreign key of the second table. The SELECT statement in this figure, for example, joins data from the Vendors and Invoices tables based on the VendorID column in each table. Notice that because the equal operator is used in this condition, the value of the VendorID column in a row in the Vendors table must match the VendorID in a row in the Invoices table for that row to be included in the result set. In other words, only vendors with one or more invoices will be included. Although you'll code most inner joins using the equal operator, you should know that you can compare two tables based on other conditions, too.

In this example, the Vendors table is joined with the Invoices table using a column that has the same name in both tables: VendorID. Because of that, the columns must be qualified to indicate which table they come from. As you can see, you code a *qualified column name* by entering the table name and a period in front of the column name. Although this example uses qualified column names only in the join condition, you must qualify a column name anywhere it appears in the statement if the same name occurs in both tables. If you don't, SQL Server will return an error indicating that the column name is ambiguous. Of course, you can also qualify column names that aren't ambiguous. However, I recommend you do that only if it clarifies your code.

The explicit syntax for an inner join

```
SELECT select_list
  FROM table_1
    [INNER] JOIN table_2
      ON join_condition_1
    [[INNER] JOIN table_3
      ON join_condition_2]...
```

A SELECT statement that joins the Vendors and Invoices tables

```
SELECT InvoiceNumber, VendorName
  FROM Vendors JOIN Invoices
    ON Vendors.VendorID = Invoices.VendorID;
```

The result set

(114 rows)

	InvoiceNumber	VendorName
1	QP58872	IBM
2	Q545443	IBM
3	547481328	Blue Cross
4	547479217	Blue Cross
5	547480102	Blue Cross
6	P02-88D77S7	Fresno County Tax Collector
7	40318	Data Reproductions Corp

Description

- A *join* is used to combine columns from two or more tables into a result set based on the *join conditions* you specify. For an *inner join*, only those rows that satisfy the join condition are included in the result set.
- A join condition names a column in each of the two tables involved in the join and indicates how the two columns should be compared. In most cases, you use the equal operator to retrieve rows with matching columns. However, you can also use any of the other comparison operators in a join condition.
- In most cases, you'll join two tables based on the relationship between the primary key in one table and a foreign key in the other table. However, you can also join tables based on relationships not defined in the database. These are called *ad hoc relationships*.
- If the two columns in a join condition have the same name, you have to qualify them with the table name so that SQL Server can distinguish between them. To code a *qualified column name*, type the table name, followed by a period, followed by the column name.

Notes

- The INNER keyword is optional and is seldom used.
- This syntax for coding an inner join can be referred to as the *explicit syntax*. It is also called the *SQL-92 syntax* because it was introduced by the SQL-92 standards.
- You can also code an inner join using the *implicit syntax*. See figure 4-7 for more information.

When and how to use correlation names

When you name the tables to be joined in the FROM clause, you can assign temporary names to the tables called *correlation names* or *table aliases*. To do that, you use the AS phrase just as you do when you assign a column alias. After you assign a correlation name, you must use that name in place of the original table name throughout the query. This is illustrated in figure 4-2.

The first SELECT statement in this figure joins data from the Vendors and Invoices table. Here, both tables have been assigned correlation names that consist of a single letter. Although short correlation names like this can reduce typing, they can also make a query more difficult to read and maintain. As a result, you should only use correlation names when they simplify or clarify the query.

The correlation name used in the second SELECT statement in this figure, for example, simplifies the name of the InvoiceLineItems table to just LineItems. That way, the shorter name can be used to refer to the InvoiceID column of the table in the join condition. Although this doesn't improve the query in this example much, it can have a dramatic effect on a query that refers to the InvoiceLineItems table several times.

The syntax for an inner join that uses correlation names

```
SELECT select_list
FROM table_1 [AS] n1
    [INNER] JOIN table_2 [AS] n2
        ON n1.column_name operator n2.column_name
    [[INNER] JOIN table_3 [AS] n3
        ON n2.column_name operator n3.column_name]...
```

An inner join with correlation names that make the query more difficult to read

```
SELECT InvoiceNumber, VendorName, InvoiceDueDate,
       InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue
  FROM Vendors AS v JOIN Invoices AS i
    ON v.VendorID = i.VendorID
 WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0
 ORDER BY InvoiceDueDate DESC;
```

	InvoiceNumber	VendorName	InvoiceDueDate	BalanceDue
1	0-2436	Malloy Lithographing Inc	2020-02-29	10976.06
2	547480102	Blue Cross	2020-02-29	224.00
3	9982771	Ford Motor Credit Company	2020-02-23	503.20

(11 rows)

An inner join with a correlation name that simplifies the query

```
SELECT InvoiceNumber, InvoiceLineItemAmount, InvoiceLineItemDescription
  FROM Invoices JOIN InvoiceLineItems AS LineItems
    ON Invoices.InvoiceID = LineItems.InvoiceID
 WHERE AccountNo = 540
 ORDER BY InvoiceDate;
```

	InvoiceNumber	InvoiceLineItemAmount	InvoiceLineItemDescription
1	I77271-001	478.00	Publishers Marketing
2	972110	207.78	Prospect list
3	133560	175.00	Card deck advertising

(6 rows)

Description

- *Correlation names* are temporary table names assigned in the FROM clause. You can use correlation names when long table names make qualified column names long or confusing. A correlation name can also be called a *table alias*.
- If you assign a correlation name to a table, you must use that name to refer to the table within your query. You can't use the original table name.
- Although the AS keyword is optional, I recommend you use it because it makes the FROM clause easier to read.
- You can use a correlation name for any table in a join without using correlation names for all of the other tables.
- Use correlation names whenever they simplify or clarify the query. Avoid using correlation names when they make a query more confusing or difficult to read.

How to work with tables from different databases

Although it's not common, you may occasionally need to join data from tables that reside in different databases. To do that, you have to qualify one or more of the table names. Figure 4-3 shows you how.

To start, this figure presents the syntax of a *fully-qualified object name*. As you can see, a fully-qualified name consists of four parts: a server name, a database name, a schema name, and the name of the object itself. In this chapter, you'll learn how to qualify table names. However, you should realize that you can use this syntax with other objects as well.

The first SELECT statement in this figure illustrates the use of fully-qualified object names. This statement joins data from two tables (Vendors and Customers) in two different databases (AP and ProductOrders). Both databases are on the same server (DBServer) and are stored in the same schema (dbo). Here, correlation names are assigned to both of these tables to make them easier to refer to in the join condition.

Although you can qualify all table names this way, you typically specify only the parts that are different from the current settings. When you start the Management Studio, for example, you connect to a specific server. As long as you work with databases on that server, then, you don't need to include the server name. Similarly, before you execute a statement, you typically select the database it uses. So as long as you work with tables in that database, you don't need to include the database name. You can also omit the schema name as long as you work with tables in the user's default schema (see chapter 17) or in the dbo schema. That's why all of the statements you've seen up to this point have included only the table name.

When you omit one or more parts from a fully-qualified object name, you create a *partially-qualified object name*. The second SELECT statement in this figure, for example, shows how the first statement can be rewritten using partially-qualified object names. Here, the server name, database name, and schema name are all omitted from the Vendors table since it resides in the default schema (dbo) within the current database (AP) on the current server. The Customers table, however, must be qualified with the database name because it's not in the AP database. Notice that because the schema name falls between the database name and the table name, two periods were coded to indicate that this part of the name was omitted.

Before you can specify a server name as shown in this figure, you must add a *linked server* to the current instance of the server. A linked server is a virtual server that specifies all the information necessary to be able to connect to a local or remote server. To add a linked server, you can use the stored procedure named `sp_AddLinkedServer`. In this figure, for example, the stored procedure adds a linked server named DBServer to the master database for the current instance of SQL Server. This server has all the information necessary to connect to an instance of SQL Server Express that's running on the local machine. However, a similar syntax could be used to connect to an instance of SQL Server running on a remote server.

The syntax of a fully-qualified object name

```
linked_server.database.schema.object
```

A join with fully-qualified table names

```
SELECT VendorName, CustLastName, CustFirstName,
       VendorState AS State, VendorCity AS City
  FROM DBServer.AP.dbo.Vendors AS Vendors
    JOIN DBServer.ProductOrders.dbo.Customers AS Customers
      ON Vendors.VendorZipCode = Customers.CustZip
 ORDER BY State, City;
```

The same join with partially-qualified table names

```
SELECT VendorName, CustLastName, CustFirstName,
       VendorState AS State, VendorCity AS City
  FROM Vendors
    JOIN ProductOrders..Customers AS Customers
      ON Vendors.VendorZipCode = Customers.CustZip
 ORDER BY State, City;
```

The result set

	VendorName	CustLastName	CustFirstName	State	City
1	Wells Fargo Bank	Marissa	Kyle	AZ	Phoenix
2	Aztek Label	Irvin	Ania	CA	Anaheim
3	Gary McKeighan Insurance	Neftaly	Thalia	CA	Fresno
4	Gary McKeighan Insurance	Holbrooke	Rashad	CA	Fresno
5	Shields Design	Damien	Deborah	CA	Fresno

(37 rows)

A stored procedure that adds a linked server

```
USE master;
EXEC sp_addlinkedserver
  @server='DBServer',
  @srvproduct='',
  @provider='SQLNCLI',
  @datasrc='localhost\SqlExpress';
```

Description

- A *fully-qualified object name* is made up of four parts: the server name, the database name, the schema name (typically dbo), and the name of the object (typically a table). This syntax can be used when joining tables from different databases or databases on different servers.
- If the server or database name is the same as the current server or database name, or if the schema name is dbo or the name of the user's default schema, you can omit that part of the name to create a *partially-qualified object name*. If the omitted name falls between two other parts of the name, code two periods to indicate that the name is omitted.
- Before you can specify a server name, you must add a *linked server* to the current instance of the server. To do that, you can use the stored procedure named `sp_AddLinkedServer`. Then, you can specify the name of the linked server. To remove a linked server, you can use the stored procedure named `sp_DropServer`.

To remove a linked server, you can use the stored procedure named `sp_DropServer`. For more information about working with linked servers, look up “Linked Servers (Database Engine)” in the documentation for SQL Server.

How to use compound join conditions

Although a join condition typically consists of a single comparison, you can include two or more comparisons in a join condition using the AND and OR operators. Figure 4-4 illustrates how this works.

In the first SELECT statement in this figure, you can see that the `Invoices` and `InvoiceLineItems` tables are joined based on two comparisons. First, the primary key of the `Invoices` table, `InvoiceID`, is compared with the foreign key of the `InvoiceLineItems` table, also named `InvoiceID`. As in previous examples, this comparison uses an equal condition. Then, the `InvoiceTotal` column in the `Invoices` table is tested for a value greater than the value of the `InvoiceLineItemAmount` column in the `InvoiceLineItems` table. That means that only those invoices that have two or more line items will be included in the result set. You can see part the result set in this figure.

Another way to code these conditions is to code the primary join condition in the `FROM` clause and the other condition in the `WHERE` clause. This is illustrated by the second SELECT statement in this figure.

When you code separate compound join conditions like this, the join condition in the `ON` expression is performed before the tables are joined, and the search condition in the `WHERE` clause is performed after the tables are joined. Because of that, you might expect a SELECT statement to execute more efficiently if you code the search condition in the `ON` expression. However, SQL Server examines the join and search conditions as it optimizes the query. So you don’t need to worry about which technique is most efficient. Instead, you should code the conditions so they’re easy to understand.

An inner join with two conditions

```
SELECT InvoiceNumber, InvoiceDate,  
       InvoiceTotal, InvoiceLineItemAmount  
  FROM Invoices JOIN InvoiceLineItems AS LineItems  
    ON (Invoices.InvoiceID = LineItems.InvoiceID) AND  
        (Invoices.InvoiceTotal > LineItems.InvoiceLineItemAmount)  
 ORDER BY InvoiceNumber;
```

The same join with the second condition coded in a WHERE clause

```
SELECT InvoiceNumber, InvoiceDate,  
       InvoiceTotal, InvoiceLineItemAmount  
  FROM Invoices JOIN InvoiceLineItems AS LineItems  
    ON Invoices.InvoiceID = LineItems.InvoiceID  
 WHERE Invoices.InvoiceTotal > LineItems.InvoiceLineItemAmount  
 ORDER BY InvoiceNumber;
```

The result set

	InvoiceNumber	InvoiceDate	InvoiceTotal	InvoiceLineItemAmount
1	97/522	2019-12-28	1962.13	1197.00
2	97/522	2019-12-28	1962.13	765.13
3	I77271-001	2019-10-26	662.00	50.00
4	I77271-001	2019-10-26	662.00	75.60
5	I77271-001	2019-10-26	662.00	58.40
6	I77271-001	2019-10-26	662.00	478.00

Description

- A join condition can include two or more conditions connected by AND or OR operators.
- In most cases, your code will be easier to read if you code the join condition in the ON expression and search conditions in the WHERE clause.

Figure 4-4 How to use compound join conditions

How to use a self-join

A *self-join* is a join where a table is joined with itself. Although self-joins are rare, there are some unique queries that are best solved using self-joins.

Figure 4-5 presents an example of a self-join that uses the Vendors table. Notice that since the same table is used twice, correlation names are used to distinguish between the two occurrences of the table. In addition, each column name used in the query is qualified by the correlation name since the columns occur in both tables.

The join condition in this example uses three comparisons. The first two match the VendorCity and VendorState columns in the two tables. As a result, the query will return rows for vendors that reside in the same city and state as another vendor. Because a vendor resides in the same city and state as itself, however, a third comparison is included to exclude rows that match a vendor with itself. To do that, this condition uses the not equal operator to compare the VendorID columns in the two tables.

Notice that the DISTINCT keyword is also included in this SELECT statement. That way, a vendor appears only once in the result set. Otherwise, it would appear once for each row with a matching city and state.

This example also shows how you can use columns other than key columns in a join condition. Keep in mind, however, that this is an unusual situation and you're not likely to code joins like this often.

A self-join that returns vendors from cities in common with other vendors

```
SELECT DISTINCT Vendors1.VendorName, Vendors1.VendorCity,  
    Vendors1.VendorState  
FROM Vendors AS Vendors1 JOIN Vendors AS Vendors2  
    ON (Vendors1.VendorCity = Vendors2.VendorCity) AND  
        (Vendors1.VendorState = Vendors2.VendorState) AND  
        (Vendors1.VendorID <> Vendors2.VendorID)  
ORDER BY Vendors1.VendorState, Vendors1.VendorCity;
```

The result set

	VendorName	VendorCity	VendorState
1	AT&T	Phoenix	AZ
2	Computer Library	Phoenix	AZ
3	Wells Fargo Bank	Phoenix	AZ
4	Aztek Label	Anaheim	CA
5	Blue Shield of California	Anaheim	CA
6	Abbey Office Furnishings	Fresno	CA
7	ASC Signs	Fresno	CA
8	BFI Industries	Fresno	CA

(84 rows)

Description

- A *self-join* is a join that joins a table with itself.
- When you code a self-join, you must use correlation names for the tables, and you must qualify each column name with the correlation name.
- Self-joins frequently include the DISTINCT keyword to eliminate duplicate rows.

Figure 4-5 How to use a self-join

Inner joins that join more than two tables

So far in this chapter, you've seen how to join data from two tables. However, SQL Server lets you join data from up to 256 tables. Of course, it's not likely that you'll ever need to join data from more than a few tables. In addition, each join requires additional system resources, so you should limit the number of joined tables whenever possible.

The SELECT statement in figure 4-6 joins data from four tables: Vendors, Invoices, InvoiceLineItems, and GLAccounts. Each of the joins is based on the relationship between the primary key of one table and a foreign key of the other table. For example, the AccountNo column is the primary key of the GLAccounts table and a foreign key of the InvoiceLineItems table.

Below the SELECT statement, you can see three tables. The first one presents the result of the join between the Vendors and Invoices tables. This table can be referred to as an *interim table* because it contains interim results. Similarly, the second table shows the result of the join between the first interim table and the InvoiceLineItems table. And the third table shows the result of the join between the second interim table and the GLAccounts table after the ORDER BY sequence is applied.

As you review the three tables in this figure, keep in mind that SQL Server may not actually process the joins as illustrated here. However, the idea of interim tables should help you understand how multi-table joins work.

A SELECT statement that joins four tables

```

SELECT VendorName, InvoiceNumber, InvoiceDate,
       InvoiceLineItemAmount AS LineItemAmount, AccountDescription
  FROM Vendors
    JOIN Invoices ON Vendors.VendorID = Invoices.VendorID
    JOIN InvoiceLineItems
      ON Invoices.InvoiceID = InvoiceLineItems.InvoiceID
    JOIN GLAccounts ON InvoiceLineItems.AccountNo = GLAccounts.AccountNo
 WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0
 ORDER BY VendorName, LineItemAmount DESC;

```

The first interim table

	VendorName	InvoiceNumber	InvoiceDate
1	Blue Cross	547480102	2020-02-01
2	Cardinal Business Media, Inc.	134116	2020-01-28
3	Data Reproductions Corp	39104	2020-01-10
4	Federal Express Corporation	963253264	2020-01-18
5	Federal Express Corporation	263253268	2020-01-21
6	Federal Express Corporation	263253270	2020-01-22
7	Federal Express Corporation	263253273	2020-01-22

(11 rows)

The second interim table

	VendorName	InvoiceNumber	InvoiceDate	LineItemAmount
1	Blue Cross	547480102	2020-02-01	224.00
2	Cardinal Business Media, Inc.	134116	2020-01-28	90.36
3	Data Reproductions Corp	39104	2020-01-10	85.31
4	Federal Express Corporation	263253273	2020-01-22	30.75
5	Federal Express Corporation	963253264	2020-01-18	52.25
6	Federal Express Corporation	263253268	2020-01-21	59.97
7	Federal Express Corporation	263253270	2020-01-22	67.92

(11 rows)

The final result set

	VendorName	InvoiceNumber	InvoiceDate	LineItemAmount	AccountDescription
1	Blue Cross	547480102	2020-02-01	224.00	Group Insurance
2	Cardinal Business Media, Inc.	134116	2020-01-28	90.36	Direct Mail Advertising
3	Data Reproductions Corp	39104	2020-01-10	85.31	Book Printing Costs
4	Federal Express Corporation	263253270	2020-01-22	67.92	Freight
5	Federal Express Corporation	263253268	2020-01-21	59.97	Freight
6	Federal Express Corporation	963253264	2020-01-18	52.25	Freight
7	Federal Express Corporation	263253273	2020-01-22	30.75	Freight

(11 rows)

Description

- You can think of a multi-table join as a series of two-table joins proceeding from left to right. The first two tables are joined to produce an *interim result set* or *interim table*. Then, the interim table is joined with the next table, and so on.

How to use the implicit inner join syntax

Earlier in this chapter, I mentioned that SQL Server provides an *implicit syntax* for joining tables. This syntax was used prior to the SQL-92 standards. Although I recommend you use the explicit syntax, you should be familiar with the implicit syntax in case you ever need to maintain SQL statements that use it.

Figure 4-7 presents the implicit syntax for an inner join along with two statements that use it. As you can see, the tables to be joined are simply listed in the FROM clause. Then, the join conditions are included in the WHERE clause.

The first SELECT statement, for example, joins data from the Vendors and Invoices table. Like the SELECT statement you saw back in figure 4-1, these tables are joined based on an equal comparison between the VendorID columns in the two tables. In this case, though, the comparison is coded as the search condition of the WHERE clause. If you compare the result set shown in this figure with the one in figure 4-1, you'll see that they're identical.

The second SELECT statement uses the implicit syntax to join data from four tables. This is the same join you saw in figure 4-6. Notice in this example that the three join conditions are combined in the WHERE clause using the AND operator. In addition, an AND operator is used to combine the join conditions with the search condition.

Because the explicit syntax for joins lets you separate join conditions from search conditions, statements that use the explicit syntax are typically easier to read than those that use the implicit syntax. In addition, the explicit syntax helps you avoid a common coding mistake with the implicit syntax: omitting the join condition. As you'll learn later in this chapter, an implicit join without a join condition results in a cross join, which can return a large number of rows. For these reasons, I recommend you use the explicit syntax in all your new SQL code.

The implicit syntax for an inner join

```
SELECT select_list
FROM table_1, table_2 [, table_3]...
WHERE table_1.column_name operator table_2.column_name
[AND table_2.column_name operator table_3.column_name]...
```

A SELECT statement that joins the Vendors and Invoices tables

```
SELECT InvoiceNumber, VendorName
FROM Vendors, Invoices
WHERE Vendors.VendorID = Invoices.VendorID;
```

The result set

	InvoiceNumber	VendorName
1	QP58872	IBM
2	Q545443	IBM
3	547481328	Blue Cross
4	547479217	Blue Cross
5	547480102	Blue Cross
6	P02-88D77S7	Fresno County Tax Collector
7	40318	Data Reproductions Corp

A statement that joins four tables

```
SELECT VendorName, InvoiceNumber, InvoiceDate,
       InvoiceLineItemAmount AS LineItemAmount, AccountDescription
  FROM Vendors, Invoices, InvoiceLineItems, GLAccounts
 WHERE Vendors.VendorID = Invoices.VendorID
   AND Invoices.InvoiceID = InvoiceLineItems.InvoiceID
   AND InvoiceLineItems.AccountNo = GLAccounts.AccountNo
   AND InvoiceTotal - PaymentTotal - CreditTotal > 0
 ORDER BY VendorName, LineItemAmount DESC;
```

The result set

	VendorName	InvoiceNumber	InvoiceDate	LineItemAmount	AccountDescription
1	Blue Cross	547480102	2020-02-01	224.00	Group Insurance
2	Cardinal Business Media, Inc.	134116	2020-01-28	90.36	Direct Mail Advertising
3	Data Reproductions Corp	39104	2020-01-10	85.31	Book Printing Costs
4	Federal Express Corporation	263253270	2020-01-22	67.92	Freight
5	Federal Express Corporation	263253268	2020-01-21	59.97	Freight
6	Federal Express Corporation	963253264	2020-01-18	52.25	Freight
7	Federal Express Corporation	263253273	2020-01-22	30.75	Freight

Description

- Instead of coding a join condition in the FROM clause, you can code it in the WHERE clause along with any search conditions. Then, you simply list the tables you want to join in the FROM clause separated by commas.
- This syntax for coding joins is referred to as the *implicit syntax*, or the *theta syntax*. It was used prior to the SQL-92 standards, which introduced the explicit syntax.
- If you omit the join condition from the WHERE clause, a cross join is performed. You'll learn about cross joins later in this chapter.

Figure 4-7 How to use the implicit inner join syntax

How to work with outer joins

Although inner joins are the type of join you'll use most often, SQL Server also supports *outer joins*. Unlike an inner join, an outer join returns all of the rows from one or both tables involved in the join, regardless of whether the join condition is true. You'll see how this works in the topics that follow.

How to code an outer join

Figure 4-8 presents the explicit syntax for coding an outer join. Because this syntax is similar to the explicit syntax for inner joins, you shouldn't have any trouble understanding how it works. The main difference is that you include the LEFT, RIGHT, or FULL keyword to specify the type of outer join you want to perform. As you can see in the syntax, you can also include the OUTER keyword, but it's optional and is usually omitted.

The table in this figure summarizes the differences between left, right, and full outer joins. When you use a *left outer join*, the result set includes all the rows from the first, or left, table. Similarly, when you use a *right outer join*, the result set includes all the rows from the second, or right, table. And when you use a *full outer join*, the result set includes all the rows from both tables.

The example in this figure illustrates a left outer join. Here, the Vendors table is joined with the Invoices table. Notice that the result set includes vendor rows even if no matching invoices are found. In that case, null values are returned for the columns in the Invoices table.

When coding outer joins, it's a common practice to avoid using right joins. To do that, you can substitute a left outer join for a right outer join by reversing the order of the tables in the FROM clause and using the LEFT keyword instead of RIGHT. This often makes it easier to read statements that join more than two tables.

In addition to the explicit syntax for coding outer joins, earlier versions of SQL Server provided for an implicit syntax. This worked much the same as the implicit syntax for coding inner joins. For outer joins, however, you used the *= operator in the WHERE clause to identify a left outer join, and you used the =* operator to identify a right outer join. Although you can't use these operators in SQL Server 2005 and later, you should at least be aware of them in case you come across them in older queries.

The explicit syntax for an outer join

```
SELECT select_list
  FROM table_1
    {LEFT|RIGHT|FULL} [OUTER] JOIN table_2
      ON join_condition_1
    [{LEFT|RIGHT|FULL} [OUTER] JOIN table_3
      ON join_condition_2]...
```

What outer joins do

Joins of this type	Keep unmatched rows from
Left outer join	The first (left) table
Right outer join	The second (right) table
Full outer join	Both tables

A SELECT statement that uses a left outer join

```
SELECT VendorName, InvoiceNumber, InvoiceTotal
  FROM Vendors LEFT JOIN Invoices
    ON Vendors.VendorID = Invoices.VendorID
  ORDER BY VendorName;
```

	VendorName	InvoiceNumber	InvoiceTotal
1	Abbey Office Furnishings	203339-13	17.50
2	American Booksellers Assoc	NULL	NULL
3	American Express	NULL	NULL
4	ASC Signs	NULL	NULL
5	Ascom Hasler Mailing Systems	NULL	NULL
6	AT&T	NULL	NULL

(202 rows)

Description

- An *outer join* retrieves all rows that satisfy the join condition, plus unmatched rows in one or both tables.
- In most cases, you use the equal operator to retrieve rows with matching columns. However, you can also use any of the other comparison operators.
- When a row with unmatched columns is retrieved, any columns from the other table that are included in the result set are given null values.

Notes

- The OUTER keyword is optional and typically omitted.
- Prior to SQL Server 2005, you could also use the implicit syntax to code left outer and right outer joins. To do that, you listed the tables to be joined in the FROM clause, and you used the *= (left) and =* (right) operators in the WHERE clause to specify the join condition.

Figure 4-8 How to code an outer join

Outer join examples

To give you a better understanding of how outer joins work, figure 4-9 presents three more examples. These examples use the Departments and Employees tables shown at the top of this figure. In each case, the join condition joins the tables based on the values in their DeptNo columns.

The first SELECT statement performs a left outer join on these two tables. In the result set produced by this statement, you can see that department number 3 is included in the result set even though none of the employees in the Employees table work in that department. Because of that, a null value is assigned to the LastName column from that table.

The second SELECT statement uses a right outer join. In this case, all of the rows from the Employees table are included in the result set. Notice, however, that two of the employees, Watson and Locario, are assigned to a department that doesn't exist in the Departments table. Of course, if the DeptNo column in this table had been defined as a foreign key to the Departments table, this would not have been allowed. In this case, though, a foreign key wasn't defined, so null values are returned for the DeptName column in these two rows.

The third SELECT statement in this figure illustrates a full outer join. If you compare the results of this query with the results of the queries that use a left and right outer join, you'll see that this is a combination of the two joins. In other words, each row in the Departments table is included in the result set, along with each row in the Employees table. Because the DeptNo column from both tables is included in this example, you can clearly identify the row in the Departments table that doesn't have a matching row in the Employees table and the two rows in the Employees table that don't have matching rows in the Departments table.

The Departments table

	DeptName	DeptNo
1	Accounting	1
2	Payroll	2
3	Operations	3
4	Personnel	4
5	Maintenance	5

The Employees table

	EmployeeID	LastName	FirstName	DeptNo
1	1	Smith	Cindy	2
2	2	Jones	Elmer	4
3	3	Simonian	Ralph	2
4	4	Hernandez	Olivia	1
5	5	Aaronsen	Robert	2
6	6	Watson	Denise	6
7	7	Hardy	Thomas	5
8	8	O'Leary	Rhea	4
9	9	Locario	Paulo	6

A left outer join

```
SELECT DeptName, Departments.DeptNo,
       LastName
  FROM Departments LEFT JOIN Employees
    ON Departments.DeptNo =
      Employees.DeptNo;
```

	DeptName	DeptNo	LastName
1	Accounting	1	Hernandez
2	Payroll	2	Smith
3	Payroll	2	Simonian
4	Payroll	2	Aaronsen
5	Operations	3	NULL
6	Personnel	4	Jones
7	Personnel	4	O'Leary
8	Maintenance	5	Hardy

A right outer join

```
SELECT DeptName, Employees.DeptNo,
       LastName
  FROM Departments RIGHT JOIN Employees
    ON Departments.DeptNo =
      Employees.DeptNo;
```

	DeptName	DeptNo	LastName
1	Payroll	2	Smith
2	Personnel	4	Jones
3	Payroll	2	Simonian
4	Accounting	1	Hernandez
5	Payroll	2	Aaronsen
6	NULL	6	Watson
7	Maintenance	5	Hardy
8	Personnel	4	O'Leary
9	NULL	6	Locario

A full outer join

```
SELECT DeptName, Departments.DeptNo,
       Employees.DeptNo, LastName
  FROM Departments FULL JOIN Employees
    ON Departments.DeptNo =
      Employees.DeptNo;
```

	DeptName	DeptNo	DeptNo	LastName
1	Accounting	1	1	Hernandez
2	Payroll	2	2	Smith
3	Payroll	2	2	Simonian
4	Payroll	2	2	Aaronsen
5	Operations	3	NULL	NULL
6	Personnel	4	4	Jones
7	Personnel	4	4	O'Leary
8	Maintenance	5	5	Hardy
9	NULL	NULL	6	Watson
10	NULL	NULL	6	Locario

Description

- From these examples, you can see that none of the employees in the Employees table work in the Operations department, and two of the employees (Watson and Locario) work in a department that doesn't exist in the Departments table.

Figure 4-9 Outer join examples

Outer joins that join more than two tables

Like inner joins, you can use outer joins to join data from more than two tables. The two examples in figure 4-10 illustrate how this works. These examples use the Departments and Employees tables you saw in the previous figure, along with a Projects table. All three of these tables are shown at the top of this figure.

The first example in this figure uses left outer joins to join the data in the three tables. Here, you can see once again that none of the employees in the Employees table are assigned to the Operations department. Because of that, null values are returned for the columns in both the Employees and Projects tables. In addition, you can see that two employees, Hardy and Jones, aren't assigned to a project.

The second example in this figure uses full outer joins to join the three tables. This result set includes unmatched rows from the Departments and Employees table just like the result set you saw in figure 4-9 that was created using a full outer join. In addition, the result set in this example includes an unmatched row from the Projects table: the one for project number P1014. In other words, none of the employees are assigned to this project.

The Departments table

	DeptName	DeptNo
1	Accounting	1
2	Payroll	2
3	Operations	3
4	Personnel	4
5	Maintenance	5

The Employees table

	EmployeeID	LastName	FirstName	DeptNo
1	1	Smith	Cindy	2
2	2	Jones	Elmer	4
3	3	Simonian	Ralph	2
4	4	Hemandez	Olivia	1
5	5	Aaronsen	Robert	2
6	6	Watson	Denise	6
7	7	Hardy	Thomas	5
8	8	O'Leary	Rhea	4
9	9	Locario	Paulo	6

The Projects table

	ProjectNo	EmployeeID
1	P1011	8
2	P1011	4
3	P1012	3
4	P1012	1
5	P1012	5
6	P1013	6
7	P1013	9
8	P1014	10

A SELECT statement that joins the three tables using left outer joins

```
SELECT DeptName, LastName, ProjectNo
FROM Departments
    LEFT JOIN Employees
        ON Departments.DeptNo = Employees.DeptNo
    LEFT JOIN Projects
        ON Employees.EmployeeID = Projects.EmployeeID
ORDER BY DeptName, LastName, ProjectNo;
```

	DeptName	LastName	ProjectNo
1	Accounting	Hemandez	P1011
2	Maintenance	Hardy	NULL
3	Operations	NULL	NULL
4	Payroll	Aaronsen	P1012
5	Payroll	Simonian	P1012
6	Payroll	Smith	P1012
7	Personnel	Jones	NULL
8	Personnel	O'Leary	P1011

A SELECT statement that joins the three tables using full outer joins

```
SELECT DeptName, LastName, ProjectNo
FROM Departments
    FULL JOIN Employees
        ON Departments.DeptNo = Employees.DeptNo
    FULL JOIN Projects
        ON Employees.EmployeeID = Projects.EmployeeID
ORDER BY DeptName;
```

	DeptName	LastName	ProjectNo
1	NULL	Watson	P1013
2	NULL	Locario	P1013
3	NULL	NULL	P1014
4	Accounting	Hemandez	P1011
5	Maintenance	Hardy	NULL
6	Operations	NULL	NULL
7	Payroll	Smith	P1012
8	Payroll	Simonian	P1012
9	Payroll	Aaronsen	P1012
10	Personnel	Jones	NULL
11	Personnel	O'Leary	P1011

Figure 4-10 Outer joins that join more than two tables

Other skills for working with joins

The two topics that follow present two additional skills for working with joins. In the first topic, you'll learn how to use inner and outer joins in the same statement. Then, in the second topic, you'll learn how to use another type of join, called a cross join.

How to combine inner and outer joins

Figure 4-11 shows how you can combine inner and outer joins. In this example, the Departments table is joined with the Employees table using an inner join. The result is an interim table that includes departments with one or more employees. Notice that the EmployeeID column is shown in this table even though it's not included in the final result set. That's because it's used by the join that follows.

After the Departments and Employees tables are joined, the interim table is joined with the Projects table using a left outer join. The result is a table that includes all of the departments that have employees assigned to them, all of the employees assigned to those departments, and the projects those employees are assigned. Here, you can clearly see that two employees, Hardy and Jones, haven't been assigned projects.

The Departments table

	DeptName	DeptNo
1	Accounting	1
2	Payroll	2
3	Operations	3
4	Personnel	4
5	Maintenance	5

The Employees table

	EmployeeID	LastName	FirstName	DeptNo
1	1	Smith	Cindy	2
2	2	Jones	Elmer	4
3	3	Simonian	Ralph	2
4	4	Hemandez	Olivia	1
5	5	Aaronsen	Robert	2
6	6	Watson	Denise	6
7	7	Hardy	Thomas	5
8	8	O'Leary	Rhea	4
9	9	Locario	Paulo	6

The Projects table

	ProjectNo	EmployeeID
1	P1011	8
2	P1011	4
3	P1012	3
4	P1012	1
5	P1012	5
6	P1013	6
7	P1013	9
8	P1014	10

A SELECT statement that combines an outer and an inner join

```
SELECT DeptName, LastName, ProjectNo
FROM Departments
JOIN Employees
    ON Departments.DeptNo = Employees.DeptNo
LEFT JOIN Projects
    ON Employees.EmployeeID = Projects.EmployeeID
ORDER BY DeptName;
```

The interim table

	DeptName	LastName	EmployeeID
1	Payroll	Smith	1
2	Personnel	Jones	2
3	Payroll	Simonian	3
4	Accounting	Hemandez	4
5	Payroll	Aaronsen	5
6	Maintenance	Hardy	7
7	Personnel	O'Leary	8

The result set

	DeptName	LastName	ProjectNo
1	Accounting	Hemandez	P1011
2	Maintenance	Hardy	NULL
3	Payroll	Smith	P1012
4	Payroll	Simonian	P1012
5	Payroll	Aaronsen	P1012
6	Personnel	Jones	NULL
7	Personnel	O'Leary	P1011

Description

- You can combine inner and outer joins within a single SELECT statement using the explicit join syntax. You can't combine inner and outer joins using the implicit syntax.

Figure 4-11 How to combine inner and outer joins

How to use cross joins

A *cross join* produces a result set that includes each row from the first table joined with each row from the second table. The result set is known as the *Cartesian product* of the tables. Figure 4-12 shows how to code a cross join using either the explicit or implicit syntax.

To use the explicit syntax, you include the CROSS JOIN keywords between the two tables in the FROM clause. Notice that because of the way a cross join works, you don't include a join condition. The same is true when you use the implicit syntax. In that case, you simply list the tables in the FROM clause and omit the join condition from the WHERE clause.

The two SELECT statements in this figure illustrate how cross joins work. Both of these statements combine data from the Departments and Employees tables. As you can see, the result is a table that includes 45 rows. That's each of the five rows in the Departments table combined with each of the nine rows in the Employees table. Although this result set is relatively small, you can imagine how large it would be if the tables included hundreds or thousands of rows.

As you study these examples, you should realize that cross joins have few practical uses. As a result, you'll rarely, if ever, need to use one.

How to code a cross join using the explicit syntax

The explicit syntax for a cross join

```
SELECT select_list
FROM table_1 CROSS JOIN table_2
```

A cross join that uses the explicit syntax

```
SELECT Departments.DeptNo, DeptName, EmployeeID, LastName
FROM Departments CROSS JOIN Employees
ORDER BY Departments.DeptNo;
```

How to code a cross join using the implicit syntax

The implicit syntax for a cross join

```
SELECT select_list
FROM table_1, table_2
```

A cross join that uses the implicit syntax

```
SELECT Departments.DeptNo, DeptName, EmployeeID, LastName
FROM Departments, Employees
ORDER BY Departments.DeptNo;
```

The result set created by the statements above

	DeptNo	DeptName	EmployeeID	LastName
1	1	Accounting	1	Smith
2	1	Accounting	2	Jones
3	1	Accounting	3	Simonian
4	1	Accounting	4	Hernandez
5	1	Accounting	5	Aaronsen
6	1	Accounting	6	Watson
7	1	Accounting	7	Hardy

(45 rows)

Description

- A *cross join* joins each row from the first table with each row from the second table. The result set returned by a cross join is known as a *Cartesian product*.
- To code a cross join using the explicit syntax, use the CROSS JOIN keywords in the FROM clause.
- To code a cross join using the implicit syntax, list the tables in the FROM clause and omit the join condition from the WHERE clause.

Figure 4-12 How to use cross joins

How to work with unions

Like a join, a *union* combines data from two or more tables. Instead of combining columns from base tables, however, a union combines rows from two or more result sets. You'll see how that works in the topics that follow.

The syntax of a union

Figure 4-13 shows how to code a union. As the syntax shows, you create a union by connecting two or more SELECT statements with the UNION keyword. For this to work, the result of each SELECT statement must have the same number of columns, and the data types of the corresponding columns in each table must be compatible.

In this syntax, I have indented all of the SELECT statements that are connected by the UNION operator to make it easier to see how this statement works. However, in a production environment, it's common to see the SELECT statements and the UNION operator coded at the same level of indentation.

If you want to sort the result of a union operation, you can code an ORDER BY clause after the last SELECT statement. Note that the column names you use in this clause must be the same as those used in the first SELECT statement. That's because the column names you use in the first SELECT statement are the ones that are used in the result set.

By default, a union operation removes duplicate rows from the result set. If that's not what you want, you can include the ALL keyword. In most cases, though, you'll omit this keyword.

Unions that combine data from different tables

The example in this figure shows how to use a union to combine data from two different tables. In this case, the ActiveInvoices table contains invoices with outstanding balances, and the PaidInvoices table contains invoices that have been paid in full. Both of these tables have the same structure as the Invoices table you've seen in previous figures.

This union operation combines the rows in both tables that have an invoice date on or after 1/1/2020. Notice that the first SELECT statement includes a column named Source that contains the literal value "Active." The second SELECT statement includes a column by the same name, but it contains the literal value "Paid." This column is used to indicate which table each row in the result set came from.

Although this column is assigned the same name in both SELECT statements, you should realize that doesn't have to be the case. In fact, none of the columns have to have the same names. Corresponding columns do have to have compatible data types. But the corresponding relationships are determined by the order in which the columns are coded in the SELECT clauses, not by their names. When you use column aliases, though, you'll typically assign the same name to corresponding columns so the statement is easier to understand.

The syntax for a union operation

```

SELECT_statement_1
UNION [ALL]
  SELECT_statement_2
[UNION [ALL]
  SELECT_statement_3]...
[ORDER BY order_by_list]
```

A union that combines invoice data from two different tables

```

SELECT 'Active' AS Source, InvoiceNumber, InvoiceDate, InvoiceTotal
FROM ActiveInvoices
WHERE InvoiceDate >= '01/01/2020'
UNION
  SELECT 'Paid' AS Source, InvoiceNumber, InvoiceDate, InvoiceTotal
  FROM PaidInvoices
  WHERE InvoiceDate >= '01/01/2020'
ORDER BY InvoiceTotal DESC;
```

The result set

	Source	InvoiceNumber	InvoiceDate	InvoiceTotal
1	Paid	P-0259	2020-01-19 00:00:00	26881.40
2	Paid	O-2060	2020-01-24 00:00:00	23517.58
3	Active	P-0608	2020-01-23 00:00:00	20551.18
4	Active	O-2436	2020-01-31 00:00:00	10976.06
5	Paid	989319-447	2020-01-24 00:00:00	3689.99
6	Paid	989319-467	2020-01-01 00:00:00	2318.03
7	Paid	989319-417	2020-01-23 00:00:00	2051.59
8	Paid	97/222	2020-01-25 00:00:00	1000.46
9	Paid	963253230	2020-01-07 00:00:00	739.20

(35 rows)

Description

- A *union* combines the result sets of two or more SELECT statements into one result set.
- Each result set must return the same number of columns, and the corresponding columns in each result set must have compatible data types.
- By default, a union eliminates duplicate rows. If you want to include duplicate rows, code the ALL keyword.
- The column names in the final result set are taken from the first SELECT clause. Column aliases assigned by the other SELECT clauses have no effect on the final result set.
- To sort the rows in the final result set, code an ORDER BY clause after the last SELECT statement. This clause must refer to the column names assigned in the first SELECT clause.

Figure 4-13 How to combine data from different tables

Unions that combine data from the same table

Figure 4-14 shows how to use unions to combine data from a single table. In the first example, rows from the Invoices table that have a balance due are combined with rows from the same table that are paid in full. As in the example in the previous figure, a column named Source is added at the beginning of each interim table. That way, the final result set indicates whether each invoice is active or paid.

The second example in this figure shows how you can use a union with data that's joined from two tables. Here, each SELECT statement joins data from the Invoices and Vendors tables. The first SELECT statement retrieves invoices with totals greater than \$10,000. Then, it calculates a payment of 33% of the invoice total. The two other SELECT statements are similar. The second one retrieves invoices with totals between \$500 and \$10,000 and calculates a 50% payment. And the third one retrieves invoices with totals less than \$500 and sets the payment amount at 100% of the total. Although this is somewhat unrealistic, it helps illustrate the flexibility of union operations.

Notice in this example that the same column aliases are assigned in each SELECT statement. Although the aliases in the second and third SELECT statements have no effect on the query, I think they make the query easier to read. In particular, it makes it easy to see that the three SELECT statements have the same number and types of columns.

A union that combines information from the Invoices table

```

SELECT 'Active' AS Source, InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0
UNION
SELECT 'Paid' AS Source, InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
WHERE InvoiceTotal - PaymentTotal - CreditTotal <= 0
ORDER BY InvoiceTotal DESC;

```

The result set

	Source	InvoiceNumber	InvoiceDate	InvoiceTotal
1	Paid	O-2058	2019-11-28	37966.19
2	Paid	P-0259	2020-01-19	26881.40
3	Paid	O-2060	2020-01-24	23517.58
4	Paid	40318	2019-12-01	21842.00
5	Active	P-0608	2020-01-23	20551.18

(114 rows)

A union that combines payment data from the same joined tables

```

SELECT InvoiceNumber, VendorName, '33% Payment' AS PaymentType,
       InvoiceTotal AS Total, (InvoiceTotal * 0.333) AS Payment
  FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
   WHERE InvoiceTotal > 10000
UNION
SELECT InvoiceNumber, VendorName, '50% Payment' AS PaymentType,
       InvoiceTotal AS Total, (InvoiceTotal * 0.5) AS Payment
  FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
   WHERE InvoiceTotal BETWEEN 500 AND 10000
UNION
SELECT InvoiceNumber, VendorName, 'Full amount' AS PaymentType,
       InvoiceTotal AS Total, InvoiceTotal AS Payment
  FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
   WHERE InvoiceTotal < 500
ORDER BY PaymentType, VendorName, InvoiceNumber;

```

The result set

	InvoiceNumber	VendorName	PaymentType	Total	Payment
6	P-0608	Malloy Lithographing Inc	33% Payment	20551.18	6843.5429400
7	509786	Bertelsmann Industry Svcs. Inc	50% Payment	6940.25	3470.1250000
8	587056	Cahners Publishing Company	50% Payment	2184.50	1092.2500000
9	367447	Computerworld	50% Payment	2433.00	1216.5000000

(114 rows)

Figure 4-14 Unions that combine data from the same table

How to use the EXCEPT and INTERSECT operators

Like the UNION operator, the EXCEPT and INTERSECT operators work with two or more result sets as shown in figure 4-15. Because of that, all three of these operators can be referred to as *set operators*. In addition, the EXCEPT and INTERSECT operators follow many of the same rules as the UNION operator.

The first query shown in this figure uses the EXCEPT operator to return the first and last names of all customers in the Customers table except any customers whose first and last names also exist in the Employees table. Since Thomas Hardy is the only name that's the same in both tables, this is the only row that's excluded from the result set for the query that comes before the EXCEPT operator.

The second query shown in this figure uses the INTERSECT operator to return the first and last names of all customers in the Customers table whose first and last names also exist in the Employees table. Since Thomas Hardy is the only name that exists in both tables, this is the only row that's returned for the result set for this query.

When you use the EXCEPT and INTERSECT operators, you must follow many of the same rules that you must follow when working with the UNION operator. To start, both of the statements that are connected by these operators must return the same number of columns. In addition, the data types for these columns must be compatible. Finally, when two queries are joined by an EXCEPT or INTERSECT operator, the column names in the final result set are taken from the first query. That's why the ORDER BY clause uses the CustomerLast column specified by the first query instead of the LastName column specified by the second query. If you understand how the UNION operator works, you shouldn't have any trouble understanding these rules.

Although it's often possible to get the same result sets by using an inner join or a subquery instead of the EXCEPT and INTERSECT operators, these operators are a helpful feature of SQL Server that can make it easier to compare two result sets.

The syntax for the EXCEPT and INTERSECT operations

```
SELECT_statement_1
{EXCEPT | INTERSECT}
  SELECT_statement_2
[ORDER BY order_by_list]
```

The Customers table

	CustomerFirst	CustomerLast
1	Maria	Anders
2	Ana	Trujillo
3	Antonio	Moreno
4	Thomas	Hardy
5	Christina	Berglund
6	Hanna	Moos

(24 rows)

The Employees table

	FirstName	LastName
4	Olivia	Hernandez
5	Robert	Aaronsen
6	Denise	Watson
7	Thomas	Hardy
8	Rhea	O'Leary
9	Paulo	Locario

(9 rows)

A query that excludes rows from the first query if they also occur in the second query

```
SELECT CustomerFirst, CustomerLast
FROM Customers
EXCEPT
  SELECT FirstName, LastName
  FROM Employees
  ORDER BY CustomerLast;
```

The result set

	CustomerFirst	CustomerLast
4	Donna	Chelan
5	Fred	Citeaux
6	Karl	Jablonski
7	Yoshi	Latimer

(23 rows)

A query that only includes rows that occur in both queries

```
SELECT CustomerFirst, CustomerLast
FROM Customers
INTERSECT
  SELECT FirstName, LastName
  FROM Employees;
```

The result set

	CustomerFirst	CustomerLast
1	Thomas	Hardy

(1 row)

Description

- The number of columns must be the same in both SELECT statements.
- The data types for each column must be compatible.
- The column names in the final result set are taken from the first SELECT statement.

Figure 4-15 How to use the EXCEPT and INTERSECT operators

Perspective

In this chapter, you learned a variety of techniques for combining data from two or more tables into a single result set. In particular, you learned how to use the SQL-92 syntax for combining data using inner joins. Of all the techniques presented in this chapter, this is the one you'll use most often. So you'll want to be sure you understand it thoroughly before you go on.

Terms

join	interim table
join condition	implicit syntax
inner join	theta syntax
ad hoc relationship	outer join
qualified column name	left outer join
explicit syntax	right outer join
correlation name	full outer join
table alias	cross join
fully-qualified object name	Cartesian product
partially-qualified object name	union
self-join	set operator
interim result set	

Exercises

Unless otherwise stated, use the explicit join syntax.

1. Write a SELECT statement that returns all columns from the Vendors table inner-joined with the Invoices table.
2. Write a SELECT statement that returns four columns:

VendorName	From the Vendors table
InvoiceNumber	From the Invoices table
InvoiceDate	From the Invoices table
Balance	InvoiceTotal minus the sum of PaymentTotal and CreditTotal

The result set should have one row for each invoice with a non-zero balance.

Sort the result set by VendorName in ascending order.

3. Write a SELECT statement that returns three columns:

VendorName	From the Vendors table
DefaultAccountNo	From the Vendors table
AccountDescription	From the GLAccounts table

The result set should have one row for each vendor, with the account number and account description for that vendor's default account number. Sort the result set by AccountDescription, then by VendorName.

4. Generate the same result set described in exercise 2, but use the implicit join syntax.
5. Write a SELECT statement that returns five columns from three tables, all using column aliases:

Vendor	VendorName column
Date	InvoiceDate column
Number	InvoiceNumber column
#	InvoiceSequence column
LineItem	InvoiceLineItemAmount column

Assign the following correlation names to the tables:

v	Vendors table
i	Invoices table
li	InvoiceLineItems table

Sort the final result set by Vendor, Date, Number, and #.

6. Write a SELECT statement that returns three columns:

VendorID	From the Vendors table
VendorName	From the Vendors table
Name	A concatenation of VendorContactFName and VendorContactLName, with a space in between

The result set should have one row for each vendor whose contact has the same first name as another vendor's contact. Sort the final result set by Name.

Hint: Use a self-join.

7. Write a SELECT statement that returns two columns from the GLAccounts table: AccountNo and AccountDescription. The result set should have one row for each account number that has never been used. Sort the final result set by AccountNo.

Hint: Use an outer join to the InvoiceLineItems table.

8. Use the UNION operator to generate a result set consisting of two columns from the Vendors table: VendorName and VendorState. If the vendor is in California, the VendorState value should be "CA"; otherwise, the VendorState value should be "Outside CA." Sort the final result set by VendorName.

5

How to code summary queries

In this chapter, you'll learn how to code queries that summarize data. For example, you can use summary queries to report sales totals by vendor or state, or to get a count of the number of invoices that were processed each day of the month. You'll also learn how to use a special type of function called an aggregate function. Aggregate functions allow you to easily do jobs like figure averages or totals, or find the highest value for a given column. So you'll use them frequently in your summary queries.

How to work with aggregate functions	160
How to code aggregate functions	160
Queries that use aggregate functions.....	162
How to group and summarize data.....	164
How to code the GROUP BY and HAVING clauses	164
Queries that use the GROUP BY and HAVING clauses.....	166
How the HAVING clause compares to the WHERE clause	168
How to code complex search conditions	170
How to summarize data using SQL Server extensions	172
How to use the ROLLUP operator	172
How to use the CUBE operator	174
How to use the GROUPING SETS operator.....	176
How to use the OVER clause.....	178
Perspective	180

How to work with aggregate functions

In chapter 3, you were introduced to *scalar functions*, which operate on a single value and return a single value. In this chapter, you'll learn how to use *aggregate functions*, which operate on a series of values and return a single summary value. Because aggregate functions typically operate on the values in columns, they are sometimes referred to as *column functions*. A query that contains one or more aggregate functions is typically referred to as a *summary query*.

How to code aggregate functions

Figure 5-1 presents the syntax of the most common aggregate functions. Since the purpose of these functions is self-explanatory, I'll focus mainly on how you use them.

All of the functions but one operate on an expression. In the query in this figure, for example, the expression that's coded for the SUM function calculates the balance due of an invoice using the InvoiceTotal, PaymentTotal, and CreditTotal columns. The result is a single value that represents the total amount due for all the selected invoices. If you look at the WHERE clause in this example, you'll see that it includes only those invoices with a balance due.

In addition to an expression, you can also code the ALL or DISTINCT keyword in these functions. ALL is the default, which means that all values are included in the calculation. The exceptions are null values, which are always excluded from these functions.

If you don't want duplicate values included, you can code the DISTINCT keyword. In most cases, you'll use DISTINCT only with the COUNT function. You'll see an example of that in the next figure. You won't use it with MIN or MAX because it has no effect on those functions. And it doesn't usually make sense to use it with the AVG and SUM functions.

Unlike the other aggregate functions, you can't use the ALL or DISTINCT keywords or an expression with COUNT(*). Instead, you code this function exactly as shown in the syntax. The value returned by this function is the number of rows in the base table that satisfy the search condition of the query, including rows with null values. The COUNT(*) function in the query in this figure, for example, indicates that the Invoices table contains 11 invoices with a balance due.

The syntax of the aggregate functions

Function syntax	Result
<code>AVG([ALL DISTINCT] expression)</code>	The average of the non-null values in the expression.
<code>SUM([ALL DISTINCT] expression)</code>	The total of the non-null values in the expression.
<code>MIN([ALL DISTINCT] expression)</code>	The lowest non-null value in the expression.
<code>MAX([ALL DISTINCT] expression)</code>	The highest non-null value in the expression.
<code>COUNT([ALL DISTINCT] expression)</code>	The number of non-null values in the expression.
<code>COUNT(*)</code>	The number of rows selected by the query.

A summary query that counts unpaid invoices and calculates the total due

```
SELECT COUNT(*) AS NumberOfInvoices,
       SUM(InvoiceTotal - PaymentTotal - CreditTotal) AS TotalDue
  FROM Invoices
 WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
```

The result set

NumberOfInvoices	TotalDue
11	32020.42

Description

- *Aggregate functions*, also called *column functions*, perform a calculation on the values in a set of selected rows. You specify the values to be used in the calculation by coding an expression for the function's argument. In many cases, the expression is just the name of a column.
- A SELECT statement that includes an aggregate function can be called a *summary query*.
- The expression you specify for the AVG and SUM functions must result in a numeric value. The expression for the MIN, MAX, and COUNT functions can result in a numeric, date, or string value.
- By default, all values are included in the calculation regardless of whether they're duplicated. If you want to omit duplicate values, code the DISTINCT keyword. This keyword is typically used only with the COUNT function.
- All of the aggregate functions except for COUNT(*) ignore null values.
- Aggregate functions are often used with the GROUP BY clause of the SELECT statement, which is used to group the rows in a result set. See figure 5-3 for more information.
- If you code an aggregate function in the SELECT clause, that clause can't include non-aggregate columns from the base table unless the column is specified in a GROUP BY clause or the OVER clause is included for each aggregate function. See figure 5-10 for more information on the OVER clause.

Queries that use aggregate functions

Figure 5-2 presents four more queries that use aggregate functions. Before I describe these queries, you should know that with three exceptions, a SELECT clause that contains an aggregate function can contain only aggregate functions. The first exception is if the column specification results in a literal value. This is illustrated by the first column in the first two queries in this figure. The second exception is if the query includes a GROUP BY clause. Then, the SELECT clause can include any columns specified in the GROUP BY clause. The third exception is if the aggregate functions include the OVER clause. Then, the SELECT clause can include any columns from the base tables. You'll see how you use the GROUP BY and OVER clauses later in this chapter.

The first two queries in this figure use the COUNT(*) function to count the number of rows in the Invoices table that satisfy the search condition. In both cases, only those invoices with invoice dates after 7/1/2019 are included in the count. In addition, the first query uses the AVG function to calculate the average amount of those invoices and the SUM function to calculate the total amount of those invoices. By contrast, the second query uses the MIN and MAX functions to calculate the minimum and maximum invoice amounts.

Although the MIN, MAX, and COUNT functions are typically used on columns that contain numeric data, they can also be used on columns that contain character or date data. In the third query, for example, they're used on the VendorName column in the Vendors table. Here, the MIN function returns the name of the vendor that's lowest in the sort sequence, the MAX function returns the name of the vendor that's highest in the sort sequence, and the COUNT function returns the total number of vendors. Note that since the VendorName column can't contain null values, the COUNT(*) function would have returned the same result.

The fourth query illustrates how using the DISTINCT keyword can affect the result of a COUNT function. Here, the first COUNT function uses the DISTINCT keyword to count the number of vendors that have invoices dated after 7/1/2019 in the Invoices table. To do that, it looks for distinct values in the VendorID column. By contrast, because the second COUNT function doesn't include the DISTINCT keyword, it counts every invoice after 7/1/2019. Of course, you could accomplish the same thing using the COUNT(*) function. I used COUNT(VendorID) here only to illustrate the difference between coding and not coding the DISTINCT keyword.

A summary query that uses the COUNT(*), AVG, and SUM functions

```
SELECT 'After 7/1/2019' AS SelectionDate, COUNT(*) AS NumberOfInvoices,
       AVG(InvoiceTotal) AS AverageInvoiceAmount,
       SUM(InvoiceTotal) AS TotalInvoiceAmount
  FROM Invoices
 WHERE InvoiceDate > '2019-07-01';
```

	SelectionDate	NumberOfInvoices	AverageInvoiceAmount	TotalInvoiceAmount
1	After 7/1/2019	114	1879.7413	214290.51

A summary query that uses the MIN and MAX functions

```
SELECT 'After 7/1/2019' AS SelectionDate, COUNT(*) AS NumberOfInvoices,
       MAX(InvoiceTotal) AS HighestInvoiceTotal,
       MIN(InvoiceTotal) AS LowestInvoiceTotal
  FROM Invoices
 WHERE InvoiceDate > '2019-07-01';
```

	SelectionDate	NumberOfInvoices	HighestInvoiceTotal	LowestInvoiceTotal
1	After 7/1/2019	114	37966.19	6.00

A summary query that works on non-numeric columns

```
SELECT MIN(VendorName) AS FirstVendor,
       MAX(VendorName) AS LastVendor,
       COUNT(VendorName) AS NumberOfVendors
  FROM Vendors;
```

	FirstVendor	LastVendor	NumberOfVendors
1	Abbey Office Furnishings	Zylka Design	122

A summary query that uses the DISTINCT keyword

```
SELECT COUNT(DISTINCT VendorID) AS NumberOfVendors,
       COUNT(VendorID) AS NumberOfInvoices,
       AVG(InvoiceTotal) AS AverageInvoiceAmount,
       SUM(InvoiceTotal) AS TotalInvoiceAmount
  FROM Invoices
 WHERE InvoiceDate > '2019-07-01';
```

	NumberOfVendors	NumberOfInvoices	AverageInvoiceAmount	TotalInvoiceAmount
1	34	114	1879.7413	214290.51

Notes

- If you want to count all of the selected rows, you'll typically use the COUNT(*) function as illustrated by the first two examples above. An alternative is to code the name of any column in the base table that can't contain null values, as illustrated by the third example.
- If you want to count only the rows with unique values in a specified column, you can code the COUNT function with the DISTINCT keyword followed by the name of the column, as illustrated by the fourth example.

How to group and summarize data

Now that you understand how aggregate functions work, you're ready to learn how to group data and use aggregate functions to summarize the data in each group. To do that, you need to learn about two more clauses of the SELECT statement: GROUP BY and HAVING.

How to code the GROUP BY and HAVING clauses

Figure 5-3 presents the syntax of the SELECT statement with the GROUP BY and HAVING clauses. The GROUP BY clause determines how the selected rows are grouped, and the HAVING clause determines which groups are included in the final results. As you can see, these clauses are coded after the WHERE clause but before the ORDER BY clause. That makes sense because the search condition in the WHERE clause is applied before the rows are grouped, and the sort sequence in the ORDER BY clause is applied after the rows are grouped.

In the GROUP BY clause, you list one or more columns or expressions separated by commas. Then, the rows that satisfy the search condition in the WHERE clause are grouped by those columns or expressions in ascending sequence. That means that a single row is returned for each unique set of values in the GROUP BY columns. This will make more sense when you see the examples in the next figure that group by two columns. For now, take a look at the example in this figure that groups by a single column.

This example calculates the average invoice amount for each vendor who has invoices in the Invoices table that average over \$2,000. To do that, it groups the invoices by VendorID. Then, the AVG function calculates the average of the InvoiceTotal column. Because this query includes a GROUP BY clause, this function calculates the average invoice total for each group rather than for the entire result set. In that case, the aggregate function is called a *vector aggregate*. By contrast, aggregate functions like the ones you saw earlier in this chapter that return a single value for all the rows in a result set are called *scalar aggregates*.

The example in this figure also includes a HAVING clause. The search condition in this clause specifies that only those vendors with invoices that average over \$2,000 should be included. Note that this condition must be applied after the rows are grouped and the average for each group has been calculated.

In addition to the AVG function, the SELECT clause includes the VendorID column. That makes sense since the rows are grouped by this column. However, the columns used in the GROUP BY clause don't have to be included in the SELECT clause.

The syntax of the SELECT statement with the GROUP BY and HAVING clauses

```
SELECT select_list  
FROM table_source  
[WHERE search_condition]  
[GROUP BY group_by_list]  
[HAVING search_condition]  
[ORDER BY order_by_list]
```

A summary query that calculates the average invoice amount by vendor

```
SELECT VendorID, AVG(InvoiceTotal) AS AverageInvoiceAmount  
FROM Invoices  
GROUP BY VendorID  
HAVING AVG(InvoiceTotal) > 2000  
ORDER BY AverageInvoiceAmount DESC;
```

	VendorID	AverageInvoiceAmount
1	110	23978.482
2	72	10963.655
3	104	7125.34
4	99	6940.25
5	119	4901.26
6	122	2575.3288
7	86	2433.00
8	100	2184.50

Description

- The GROUP BY clause groups the rows of a result set based on one or more columns or expressions. It's typically used in SELECT statements that include aggregate functions.
- If you include aggregate functions in the SELECT clause, the aggregate is calculated for each set of values that result from the columns named in the GROUP BY clause.
- If you include two or more columns or expressions in the GROUP BY clause, they form a hierarchy where each column or expression is subordinate to the previous one.
- When a SELECT statement includes a GROUP BY clause, the SELECT clause can include aggregate functions, the columns used for grouping, and expressions that result in a constant value.
- A group-by list typically consists of the names of one or more columns separated by commas. However, it can contain any expression except for those that contain aggregate functions.
- The HAVING clause specifies a search condition for a group or an aggregate. This condition is applied after the rows that satisfy the search condition in the WHERE clause are grouped.

Figure 5-3 How to code the GROUP BY and HAVING clauses

Queries that use the GROUP BY and HAVING clauses

Figure 5-4 presents three more queries that group data. If you understood the query in the last figure, you shouldn't have any trouble understanding how the first query in this figure works. It groups the rows in the Invoices table by VendorID and returns a count of the number of invoices for each vendor.

The second query in this figure illustrates how you can group by more than one column. Here, a join is used to combine the VendorState and VendorCity columns from the Vendors table with a count and average of the invoices in the Invoices table. Because the rows are grouped by both state and city, a row is returned for each state and city combination. Then, the ORDER BY clause sorts the rows by city within state. Without this clause, the rows would be returned in no particular sequence.

The third query is identical to the second query except that it includes a HAVING clause. This clause uses the COUNT function to limit the state and city groups that are included in the result set to those that have two or more invoices. In other words, it excludes groups that have only one invoice.

A summary query that counts the number of invoices by vendor

```
SELECT VendorID, COUNT(*) AS InvoiceQty
FROM Invoices
GROUP BY VendorID;
```

	VendorID	InvoiceQty
1	34	2
2	37	3
3	48	1
4	72	2
5	80	2

(34 rows)

A summary query that calculates the number of invoices and the average invoice amount for the vendors in each state and city

```
SELECT VendorState, VendorCity, COUNT(*) AS InvoiceQty,
       AVG(InvoiceTotal) AS InvoiceAvg
  FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
 GROUP BY VendorState, VendorCity
 ORDER BY VendorState, VendorCity;
```

	VendorState	VendorCity	InvoiceQty	InvoiceAvg
1	AZ	Phoenix	1	662.00
2	CA	Fresno	19	1208.7457
3	CA	Los Angeles	1	503.20
4	CA	Oxnard	3	188.00
5	CA	Pasadena	5	196.12

(20 rows)

A summary query that limits the groups to those with two or more invoices

```
SELECT VendorState, VendorCity, COUNT(*) AS InvoiceQty,
       AVG(InvoiceTotal) AS InvoiceAvg
  FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
 GROUP BY VendorState, VendorCity
 HAVING COUNT(*) >= 2
 ORDER BY VendorState, VendorCity;
```

	VendorState	VendorCity	InvoiceQty	InvoiceAvg
1	CA	Fresno	19	1208.7457
2	CA	Oxnard	3	188.00
3	CA	Pasadena	5	196.12
4	CA	Sacramento	7	253.0014
5	CA	San Francisco	3	1211.04

(12 rows)

Note

- You can use a join with a summary query to group and summarize the data in two or more tables.

Figure 5-4 Queries that use the GROUP BY and HAVING clauses

How the HAVING clause compares to the WHERE clause

As you've seen, you can limit the groups included in a result set by coding a search condition in the HAVING clause. In addition, you can apply a search condition to each row before it's included in a group. To do that, you code the search condition in the WHERE clause just as you would for any SELECT statement. To make sure you understand the differences between search conditions coded in the HAVING and WHERE clauses, figure 5-5 presents two examples.

In the first example, the invoices in the *Invoices* table are grouped by vendor name, and a count and average invoice amount are calculated for each group. Then, the HAVING clause limits the groups in the result set to those that have an average invoice total greater than \$500.

By contrast, the second example includes a search condition in the WHERE clause that limits the invoices included in the groups to those that have an invoice total greater than \$500. In other words, the search condition in this example is applied to every row. In the previous example, it was applied to each group of rows.

Beyond this, there are also two differences in the expressions that you can include in the WHERE and HAVING clauses. First, the HAVING clause can include aggregate functions as you saw in the first example in this figure, but the WHERE clause can't. That's because the search condition in a WHERE clause is applied before the rows are grouped. Second, although the WHERE clause can refer to any column in the base tables, the HAVING clause can only refer to columns included in the SELECT or GROUP BY clause. That's because it filters the summarized result set that's defined by the SELECT, FROM, WHERE, and GROUP BY clauses. In other words, it doesn't filter the base tables.

A summary query with a search condition in the HAVING clause

```
SELECT VendorName, COUNT(*) AS InvoiceQty,
       AVG(InvoiceTotal) AS InvoiceAvg
  FROM Vendors JOIN Invoices
    ON Vendors.VendorID = Invoices.VendorID
 GROUP BY VendorName
 HAVING AVG(InvoiceTotal) > 500
 ORDER BY InvoiceQty DESC;
```

	VendorName	InvoiceQty	InvoiceAvg
1	United Parcel Service	9	2575.3288
2	Zylka Design	8	867.5312
3	Malloy Lithographing Inc	5	23978.482
4	Data Reproductions Corp	2	10963.655
5	IBM	2	600.06

(19 rows)

A summary query with a search condition in the WHERE clause

```
SELECT VendorName, COUNT(*) AS InvoiceQty,
       AVG(InvoiceTotal) AS InvoiceAvg
  FROM Vendors JOIN Invoices
    ON Vendors.VendorID = Invoices.VendorID
 WHERE InvoiceTotal > 500
 GROUP BY VendorName
 ORDER BY InvoiceQty DESC;
```

	VendorName	InvoiceQty	InvoiceAvg
1	United Parcel Service	9	2575.3288
2	Zylka Design	7	946.6714
3	Malloy Lithographing Inc	5	23978.482
4	Ingram	2	1077.21
5	Pollstar	1	1750.00

(20 rows)

Description

- When you include a WHERE clause in a SELECT statement that uses grouping and aggregates, the search condition is applied before the rows are grouped and the aggregates are calculated. That way, only the rows that satisfy the search condition are grouped and summarized.
- When you include a HAVING clause in a SELECT statement that uses grouping and aggregates, the search condition is applied after the rows are grouped and the aggregates are calculated. That way, only the groups that satisfy the search condition are included in the result set.
- A HAVING clause can only refer to a column included in the SELECT or GROUP BY clause. A WHERE clause can refer to any column in the base tables.
- Aggregate functions can only be coded in the HAVING clause. A WHERE clause can't contain aggregate functions.

Figure 5-5 How the HAVING clause compares to the WHERE clause

How to code complex search conditions

You can code compound search conditions in a HAVING clause just as you can in a WHERE clause. This is illustrated by the first query in figure 5-6. This query groups invoices by invoice date and calculates a count of the invoices and the sum of the invoice totals for each date. In addition, the HAVING clause specifies three conditions. First, the invoice date must be between 1/1/2020 and 1/31/2020. Second, the invoice count must be greater than 1. And third, the sum of the invoice totals must be greater than \$100.

Because the second and third conditions in the HAVING clause in this query include aggregate functions, they must be coded in the HAVING clause. The first condition, however, doesn't include an aggregate function, so it could be coded in either the HAVING or WHERE clause. The second statement in this figure, for example, shows this condition coded in the WHERE clause. Note that the query returns the same result set regardless of where you code this condition.

So how do you know where to code a search condition? In general, I think your code will be easier to read if you include all the search conditions in the HAVING clause. If, on the other hand, you prefer to code non-aggregate search conditions in the WHERE clause, that's OK, too.

Since a search condition in the WHERE clause is applied before the rows are grouped while a search condition in the HAVING clause isn't applied until after the grouping, you might expect a performance advantage by coding all search conditions in the HAVING clause. However, SQL Server takes care of this performance issue for you when it optimizes the query. To do that, it automatically moves search conditions to whichever clause will result in the best performance, as long as that doesn't change the logic of your query. As a result, you can code search conditions wherever they result in the most readable code without worrying about system performance.

A summary query with a compound condition in the HAVING clause

```
SELECT InvoiceDate, COUNT(*) AS InvoiceQty, SUM(InvoiceTotal) AS InvoiceSum  
FROM Invoices  
GROUP BY InvoiceDate  
HAVING InvoiceDate BETWEEN '2020-01-01' AND '2020-01-31'  
    AND COUNT(*) > 1  
    AND SUM(InvoiceTotal) > 100  
ORDER BY InvoiceDate DESC;
```

The same query coded with a WHERE clause

```
SELECT InvoiceDate, COUNT(*) AS InvoiceQty, SUM(InvoiceTotal) AS InvoiceSum  
FROM Invoices  
WHERE InvoiceDate BETWEEN '2020-01-01' AND '2020-01-31'  
GROUP BY InvoiceDate  
HAVING COUNT(*) > 1  
    AND SUM(InvoiceTotal) > 100  
ORDER BY InvoiceDate DESC;
```

The result set returned by both queries

	InvoiceDate	InvoiceQty	InvoiceSum
1	2020-01-24	4	27777.77
2	2020-01-23	3	22647.21
3	2020-01-21	2	639.39
4	2020-01-19	2	27481.40
5	2020-01-08	2	207.25
6	2020-01-07	2	897.20
7	2020-01-06	2	150.77

Description

- You can use the AND and OR operators to code compound search conditions in a HAVING clause just as you can in a WHERE clause.
- If a search condition includes an aggregate function, it must be coded in the HAVING clause. Otherwise, it can be coded in either the HAVING or the WHERE clause.
- In most cases, your code will be easier to read if you code all the search conditions in the HAVING clause, but you can code non-aggregate search conditions in the WHERE clause if you prefer.

How to summarize data using SQL Server extensions

So far, this chapter has discussed standard SQL keywords and functions. However, you should also know about four extensions SQL Server provides for summarizing data: the ROLLUP, CUBE, and GROUPING SETS operators and the OVER clause.

How to use the ROLLUP operator

You can use the ROLLUP operator to add one or more summary rows to a result set that uses grouping and aggregates. The two examples in figure 5-7 illustrate how this works.

The first example shows how the ROLLUP operator works when you group by a single column. Here, the invoices in the Invoices table are grouped by VendorID, and an invoice count and invoice total are calculated for each vendor. Notice that because the ROLLUP operator is included in the GROUP BY clause, an additional row is added at the end of the result set. This row summarizes all the aggregate columns in the result set. In this case, it summarizes the InvoiceCount and InvoiceTotal columns. Because the VendorID column can't be summarized, it's assigned a null value.

The second query in this figure shows how the ROLLUP operator works when you group by two columns. This query groups the vendors in the Vendors table by state and city and counts the number of vendors in each group. Notice that in addition to a summary row at the end of the result set, summary rows are included for each state.

You should also notice the ORDER BY clause in this query. It causes the rows in the result set to be sorted by city in descending sequence within state in descending sequence. The reason these columns are sorted in descending sequence is that the sort is performed after the summary rows are added to the result set, and those rows have null values in the VendorCity column. In addition, the final summary row has a null value in the VendorState column. So if you sorted these columns in ascending sequence, the rows with null values would appear before the rows they summarize, which isn't what you want.

You can also use another function, the GROUPING function, to work with null columns in a summary row. However, this function is typically used in conjunction with the CASE function, which you'll learn about in chapter 9. So I'll present the GROUPING function in that chapter.

The syntax of the ROLLUP operator shown in these two examples was introduced with SQL Server 2008. The syntax that was used with earlier versions of SQL Server is shown below both of the examples. To use this syntax, you code the WITH ROLLUP phrase at the end of the GROUP BY clause. Because the syntax that was introduced with SQL Server 2008 is more consistent with other SQL Server extensions, you should use it for new development unless you're using a version of SQL Server before 2008.

A summary query that includes a final summary row (SQL Server 2008 or later)

```
SELECT VendorID, COUNT(*) AS InvoiceCount,
       SUM(InvoiceTotal) AS InvoiceTotal
  FROM Invoices
 GROUP BY ROLLUP(VendorID);
```

	VendorID	InvoiceCount	InvoiceTotal
30	117	1	16.62
31	119	1	4901.26
32	121	8	6940.25
33	122	9	23177.96
34	123	47	4378.02
35	NULL	114	214290.51

Summary row

Another way to code the GROUP BY clause

```
GROUP BY VendorID WITH ROLLUP
```

A summary query that includes a summary row for each grouping level (SQL Server 2008 or later)

```
SELECT VendorState, VendorCity, COUNT(*) AS QtyVendors
  FROM Vendors
 WHERE VendorState IN ('IA', 'NJ')
 GROUP BY ROLLUP(VendorState, VendorCity)
 ORDER BY VendorState DESC, VendorCity DESC;
```

	VendorState	VendorCity	QtyVendors
1	NJ	Washington	1
2	NJ	Fairfield	1
3	NJ	East Brunswick	2
4	NJ	NULL	4
5	IA	Washington	1
6	IA	Fairfield	1
7	IA	NULL	2
8	NULL	NULL	6

Summary row for state 'NJ'

Summary row for state 'IA'

Summary row for all rows

Another way to code the GROUP BY clause

```
GROUP BY VendorState, VendorCity WITH ROLLUP
```

Description

- The ROLLUP operator adds a summary row for each group specified. It also adds a summary row to the end of the result set that summarizes the entire result set. If the GROUP BY clause specifies a single group, only the final summary row is added.
- The sort sequence in the ORDER BY clause is applied after the summary rows are added. Because of that, you'll want to sort grouping columns in descending sequence so the summary row for each group, which can contain null values, appears after the other rows in the group.
- When you use the ROLLUP operator, you can't use the DISTINCT keyword in any of the aggregate functions.
- You can use the GROUPING function with the ROLLUP operator to determine if a summary row has a null value assigned to a given column. See chapter 9 for details.

Figure 5-7 How to use the ROLLUP operator

How to use the CUBE operator

Figure 5-8 shows you how to use the CUBE operator. This operator is similar to the ROLLUP operator, except that it adds summary rows for every combination of groups. This is illustrated by the two examples in this figure. As you can see, these examples are the same as the ones in figure 5-7 except that they use the CUBE operator instead of the ROLLUP operator.

In the first example, the result set is grouped by a single column. In this case, a single summary row is added at the end of the result set that summarizes all the groups. In other words, this works the same as it does with the ROLLUP operator.

In the second example, however, you can see how CUBE differs from ROLLUP when you group by two or more columns. In this case, the result set includes a summary row for each state just as it did when the ROLLUP operator was used. In addition, it includes a summary row for each city. The eighth row in this figure, for example, indicates that there are two vendors in cities named Washington. If you look at the first and fifth rows in the result set, you'll see that one of those vendors is in Washington, New Jersey and one is in Washington, Iowa. The same is true of the city named Fairfield. There are also two vendors in the city of East Brunswick, but both are in New Jersey.

As with the ROLLUP operator, the syntax of the CUBE operator shown in these examples was introduced with SQL Server 2008. The syntax that was used prior to SQL Server 2008 is shown below both of the examples. If you understand how this syntax works for the ROLLUP operator, you shouldn't have much trouble understanding how it works for the CUBE operator. Although this syntax won't work for versions of SQL Server prior to 2008, you can use it for new development.

Now that you've seen how the CUBE operator works, you may be wondering when you would use it. The fact is, you probably won't use it except to add a summary row to a result set that's grouped by a single column. And in that case, you could just as easily use the ROLLUP operator. In some unique cases, however, the CUBE operator can provide useful information that you can't get any other way.

A summary query that includes a final summary row (SQL Server 2008 or later)

```
SELECT VendorID, COUNT(*) AS InvoiceCount,
       SUM(InvoiceTotal) AS InvoiceTotal
  FROM Invoices
 GROUP BY CUBE(VendorID);
```

VendorID	InvoiceCount	InvoiceTotal
31	119	4901.26
32	121	6940.25
33	122	23177.96
34	123	4378.02
35	NULL	214290.51
Summary row		

Another way to code the GROUP BY clause

```
GROUP BY VendorID WITH CUBE
```

A summary query that includes a summary row for each set of groups (SQL Server 2008 or later)

```
SELECT VendorState, VendorCity, COUNT(*) AS QtyVendors
  FROM Vendors
 WHERE VendorState IN ('IA', 'NJ')
 GROUP BY CUBE (VendorState, VendorCity)
 ORDER BY VendorState DESC, VendorCity DESC;
```

VendorState	VendorCity	QtyVendors
1 NJ	Washington	1
2 NJ	Fairfield	1
3 NJ	East Brunswick	2
4 NJ	NULL	4
5 IA	Washington	1
6 IA	Fairfield	1
7 IA	NULL	2
8 NULL	Washington	2
9 NULL	Fairfield	2
10 NULL	East Brunswick	2
11 NULL	NULL	6
Summary row for state 'NJ'		
Summary row for state 'IA'		
Summary row for city 'Washington'		
Summary row for city 'Fairfield'		
Summary row for city 'East Brunswick'		
Summary row for all rows		

Another way to code the GROUP BY clause

```
GROUP BY VendorState, VendorCity WITH CUBE
```

Description

- The CUBE operator adds a summary row for every combination of groups specified. It also adds a summary row to the end of the result set that summarizes the entire result set.
- When you use the CUBE operator, you can't use the DISTINCT keyword in any of the aggregate functions.
- You can use the GROUPING function with the CUBE operator to determine if a summary row has a null value assigned to a given column. See chapter 9 for details.

How to use the GROUPING SETS operator

Figure 5-9 shows you how to use the GROUPING SETS operator that was introduced with SQL Server 2008. This operator is similar to the ROLLUP and CUBE operators. However, the GROUPING SETS operator only includes summary rows, it only adds those summary rows for each specified group, and it uses a syntax that's similar to the 2008 syntax for the ROLLUP and CUBE operators.

The first example in this figure is similar to the second example that's presented in the previous two figures. However, this example uses the GROUPING SETS operator instead of the CUBE or ROLLUP operator. Here, the result set only includes summary rows for the two grouping elements: state and city. To start, it displays the summary rows for the states. Then, it displays the summary rows for the cities.

The second example in this figure shows some additional features that are available when you use the GROUPING SETS operator. To start, within the parentheses after the GROUPING SETS operator, you can add additional sets of parentheses to create composite groups that consist of multiple columns. In addition, you can add an empty set of parentheses to add a group for a summary row that summarizes the entire result set. In this example, the first group is the vendor's state and city, the second group is the vendor's zip code, and the third group is an empty set of parentheses that adds a summary row for the entire result set.

When you use composite groups, you should know that you can add additional summary rows by applying the ROLLUP and CUBE operators to a composite group. To do that, you code the ROLLUP or CUBE operator before the composite group. In this figure, for instance, the third example shows a GROUPING SETS clause that applies the ROLLUP operator to the composite state/city group. This adds a summary row for each state, and it adds a summary row for the entire result set, much like the second example in figure 5-7. Although this may seem confusing at first, with a little experimentation, you should be able to get the result set you want.

A summary query with two groups

```
SELECT VendorState, VendorCity, COUNT(*) AS QtyVendors
FROM Vendors
WHERE VendorState IN ('IA', 'NJ')
GROUP BY GROUPING SETS(VendorState, VendorCity)
ORDER BY VendorState DESC, VendorCity DESC;
```

	VendorState	VendorCity	QtyVendors
1	NJ	NULL	4
2	IA	NULL	2
3	NULL	Washington	2
4	NULL	Fairfield	2
5	NULL	East Brunswick	2

A summary query with a composite grouping

```
SELECT VendorState, VendorCity, VendorZipCode,
       COUNT(*) AS QtyVendors
FROM Vendors
WHERE VendorState IN ('IA', 'NJ')
GROUP BY GROUPING SETS((VendorState, VendorCity), VendorZipCode, ())
ORDER BY VendorState DESC, VendorCity DESC;
```

	VendorState	VendorCity	VendorZipCode	QtyVendors
1	NJ	Washington	NULL	1
2	NJ	Fairfield	NULL	1
3	NJ	East Brunswick	NULL	2
4	IA	Washington	NULL	1
5	IA	Fairfield	NULL	1
6	NULL	NULL	07004	1
7	NULL	NULL	07882	1
8	NULL	NULL	08810	1
9	NULL	NULL	08816	1
10	NULL	NULL	52353	1
11	NULL	NULL	52556	1
12	NULL	NULL	NULL	6

A summary query with a group that uses the ROLLUP operator

```
GROUP BY GROUPING SETS(ROLLUP(VendorState, VendorCity), VendorZipCode)
```

Description

- The GROUPING SETS operator creates a summary row for each specified group.
- Within the parentheses after the GROUPING SETS operator, you can add additional sets of parentheses to create composite groups.
- Within the parentheses after the GROUPING SETS operator, you can add an empty set of parentheses to add a summary row that summarizes the entire result set.
- For a composite group, you can add the ROLLUP or CUBE operator to add additional summary rows. This performs the ROLLUP or CUBE operation on the composite group, which also adds a summary row that summarizes the entire result set.

Figure 5-9 How to use the GROUPING SETS operator

How to use the OVER clause

So far in this chapter, you've learned how to code summary queries that return just the summarized data. But what if you want to return the individual rows that are used to calculate the summaries along with the summary data? To do that, you can use the OVER clause as shown in figure 5-10.

In the syntax at the top of this figure, you can see that you code the OVER clause after the aggregate function, followed by a set of parentheses. Within the parentheses, you can code a PARTITION BY clause, an ORDER BY clause, or both of these clauses. This is illustrated by the three examples in this figure.

The first example calculates the total, count, and average of the invoices in the Invoices table. Here, I used the PARTITION BY clause to indicate that the invoices should be grouped by invoice date. If you look at the results of this query, you'll see that the Invoices table contains a single invoice for the first three dates. Because of that, the total, count, and average columns for those dates are calculated based on a single invoice. By contrast, the next three invoices are for the same date. In this case, the total, count, and average columns are calculated based on all three invoices.

The second example is similar, but it uses the ORDER BY clause instead of the PARTITION BY clause. Because of that, the calculations aren't grouped by the invoice date like they are in the first example. Instead, the summaries accumulate from one date to the next. For example, the total and average columns for the first invoice are the same as the invoice total because they're calculated based on just that total. The total and average columns for the second invoice, however, are calculated based on both the first and second invoices. The total and average columns for the third invoice are calculated based on the first, second, and third invoices. And so on. In addition, the count column indicates the sequence of the invoice date within the result set. A total that accumulates like this is called a *cumulative total*, and an average that's calculated based on a cumulative total is called a *moving average*.

If the result set contains more than one invoice for the same date, the summary values for all of the invoices are accumulated at the same time. This is illustrated by the fourth, fifth, and sixth invoices in this example. Here, the cumulative total column for each invoice includes the invoice totals for all three invoices for the same date, and the moving average is based on that cumulative total. Because the cumulative total now includes six rows, the count column is set to 6. In other words, the summary values for all three rows are the same.

The last example in this figure uses both the PARTITION BY and ORDER BY clauses. In this case, the invoices are grouped by terms ID and ordered by invoice date. Because of that, the summaries for each invoice date are accumulated separately within each terms ID. When one terms ID ends and the next one begins, the accumulation starts again.

The syntax of the OVER clause

```
aggregate_function OVER ([partition_by_clause] [order_by_clause])
```

A query that groups the summary data by date

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
       SUM(InvoiceTotal) OVER (PARTITION BY InvoiceDate) AS DateTotal,
       COUNT(InvoiceTotal) OVER (PARTITION BY InvoiceDate) AS DateCount,
       AVG(InvoiceTotal) OVER (PARTITION BY InvoiceDate) AS DateAvg
FROM Invoices;
```

	InvoiceNumber	InvoiceDate	InvoiceTotal	DateTotal	DateCount	DateAvg
1	989319-457	2019-10-08	3813.33	3813.33	1	3813.33
2	263253241	2019-10-10	40.20	40.20	1	40.20
3	963253234	2019-10-13	138.75	138.75	1	138.75
4	2-000-2993	2019-10-16	144.70	202.95	3	67.65
5	963253251	2019-10-16	15.50	202.95	3	67.65
6	963253261	2019-10-16	42.75	202.95	3	67.65

A query that calculates a cumulative total and moving average

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
       SUM(InvoiceTotal) OVER (ORDER BY InvoiceDate) AS CumTotal,
       COUNT(InvoiceTotal) OVER (ORDER BY InvoiceDate) AS Count,
       AVG(InvoiceTotal) OVER (ORDER BY InvoiceDate) AS MovingAvg
FROM Invoices;
```

	InvoiceNumber	InvoiceDate	InvoiceTotal	CumTotal	Count	MovingAvg
1	989319-457	2019-10-08	3813.33	3813.33	1	3813.33
2	263253241	2019-10-10	40.20	3853.53	2	1926.765
3	963253234	2019-10-13	138.75	3992.28	3	1330.76
4	2-000-2993	2019-10-16	144.70	4195.23	6	699.205
5	963253251	2019-10-16	15.50	4195.23	6	699.205
6	963253261	2019-10-16	42.75	4195.23	6	699.205

The same query grouped by TermsID

```
SELECT InvoiceNumber, TermsID, InvoiceDate, InvoiceTotal,
       SUM(InvoiceTotal)
          OVER (PARTITION BY TermsID ORDER BY InvoiceDate) AS CumTotal,
       COUNT(InvoiceTotal)
          OVER (PARTITION BY TermsID ORDER BY InvoiceDate) AS Count,
       AVG(InvoiceTotal)
          OVER (PARTITION BY TermsID ORDER BY InvoiceDate) AS MovingAvg
FROM Invoices;
```

	InvoiceNumber	TermsID	InvoiceDate	InvoiceTotal	CumTotal	Count	MovingAvg
22	97-1024A	2	2020-01-20	356.48	9415.08	16	588.4425
23	31361833	2	2020-01-21	579.42	9994.50	17	587.9117
24	134116	2	2020-01-28	90.36	10084.86	18	560.27
25	989319-457	3	2019-10-08	3813.33	3813.33	1	3813.33
26	263253241	3	2019-10-10	40.20	3853.53	2	1926.765
27	963253234	3	2019-10-13	138.75	3992.28	3	1330.76

Description

- When used with the aggregate functions, the OVER clause lets you summarize the data in a result set while still returning the rows used to calculate the summary.

Figure 5-10 How to use the OVER clause

Perspective

In this chapter, you learned how to code queries that group and summarize data. In most cases, you'll be able to use the techniques presented here to get the summary information you need. If not, you may want to find out about another tool provided by SQL Server 2019 called Analysis Services. This tool provides a graphical interface that lets you build complex data models based on cubes. Then, you can use those models to analyze the database using complex patterns and correlations. You can find out more about this tool by searching for "Analysis Services" in the documentation for SQL Server.

Terms

scalar function
aggregate function
column function
summary query

scalar aggregate
vector aggregate
cumulative total
moving average

Exercises

1. Write a SELECT statement that returns two columns from the Invoices table: VendorID and PaymentSum, where PaymentSum is the sum of the PaymentTotal column. Group the result set by VendorID.
2. Write a SELECT statement that returns two columns: VendorName and PaymentSum, where PaymentSum is the sum of the PaymentTotal column. Group the result set by VendorName. Return only 10 rows, corresponding to the 10 vendors who've been paid the most.
Hint: Use the TOP clause and join Vendors to Invoices.
3. Write a SELECT statement that returns three columns: VendorName, InvoiceCount, and InvoiceSum. InvoiceCount is the count of the number of invoices, and InvoiceSum is the sum of the InvoiceTotal column. Group the result set by vendor. Sort the result set so the vendor with the highest number of invoices appears first.
4. Write a SELECT statement that returns three columns: AccountDescription, LineItemCount, and LineItemSum. LineItemCount is the number of entries in the InvoiceLineItems table that have that AccountNo. LineItemSum is the sum of the InvoiceLineItemAmount column for that AccountNo. Filter the result set to include only those rows with LineItemCount greater than 1. Group the result set by account description, and sort it by descending LineItemCount.
Hint: Join the GLAccounts table to the InvoiceLineItems table.
5. Modify the solution to exercise 4 to filter for invoices dated from October 1, 2019 to December 31, 2019.
Hint: Join to the Invoices table to code a search condition based on InvoiceDate.

6. Write a SELECT statement that answers the following question: What is the total amount invoiced for each AccountNo? Use the ROLLUP operator to include a row that gives the grand total.

Hint: Use the InvoiceLineItemAmount column of the InvoiceLineItems table.

7. Write a SELECT statement that returns four columns: VendorName, AccountDescription, LineItemCount, and LineItemSum. LineItemCount is the row count, and LineItemSum is the sum of the InvoiceLineItemAmount column. For each vendor and account, return the number and sum of line items, sorted first by vendor, then by account description.

Hint: Use a four-table join.

8. Write a SELECT statement that answers this question: Which vendors are being paid from more than one account? Return two columns: the vendor name and the total number of accounts that apply to that vendor's invoices.

Hint: Use the DISTINCT keyword to count InvoiceLineItems.AccountNo.

9. Write a SELECT statement that returns six columns:

VendorID	From the Invoices table
InvoiceDate	From the Invoices table
InvoiceTotal	From the Invoices table
VendorTotal	The sum of the invoice totals for each vendor
VendorCount	The count of invoices for each vendor
VendorAvg	The average of the invoice totals for each vendor

The result set should include the individual invoices for each vendor.

6

How to code subqueries

A subquery is a SELECT statement that's coded within another SQL statement. As a result, you can use subqueries to build queries that would be difficult or impossible to do otherwise. In this chapter, you'll learn how to use subqueries within SELECT statements. Then, in the next chapter, you'll learn how to use them when you code INSERT, UPDATE, and DELETE statements.

An introduction to subqueries	184
How to use subqueries	184
How subqueries compare to joins.....	186
How to code subqueries in search conditions.....	188
How to use subqueries with the IN operator	188
How to compare the result of a subquery with an expression.....	190
How to use the ALL keyword	192
How to use the ANY and SOME keywords.....	194
How to code correlated subqueries.....	196
How to use the EXISTS operator	198
Other ways to use subqueries	200
How to code subqueries in the FROM clause	200
How to code subqueries in the SELECT clause.....	202
Guidelines for working with complex queries.....	204
A complex query that uses subqueries	204
A procedure for building complex queries.....	206
How to work with common table expressions	208
How to code a CTE.....	208
How to code a recursive CTE.....	210
Perspective	212

An introduction to subqueries

Since you know how to code SELECT statements, you already know how to code a *subquery*. It's simply a SELECT statement that's coded within another SQL statement. The trick to using subqueries, then, is knowing where and when to use them. You'll learn the specifics of using subqueries throughout this chapter. The two topics that follow, however, will give you an overview of where and when to use them.

How to use subqueries

In figure 6-1, you can see that a subquery can be coded, or *introduced*, in the WHERE, HAVING, FROM, or SELECT clause of a SELECT statement. The SELECT statement in this figure, for example, illustrates how you can use a subquery in the search condition of a WHERE clause. When it's used in a search condition, a subquery can be referred to as a *subquery search condition* or a *subquery predicate*.

The statement in this figure retrieves all the invoices from the Invoices table that have invoice totals greater than the average of all the invoices. To do that, the subquery calculates the average of all the invoices. Then, the search condition tests each invoice to see if its invoice total is greater than that average.

When a subquery returns a single value as it does in this example, you can use it anywhere you would normally use an expression. However, a subquery can also return a single-column result set with two or more rows. In that case, it can be used in place of a list of values, such as the list for an IN operator. In addition, if a subquery is coded within a FROM clause, it can return a result set with two or more columns. You'll learn about all of these different types of subqueries in this chapter.

You can also code a subquery within another subquery. In that case, the subqueries are said to be nested. Because *nested subqueries* can be difficult to read and can result in poor performance, you should use them only when necessary.

Four ways to introduce a subquery in a SELECT statement

1. In a WHERE clause as a search condition
2. In a HAVING clause as a search condition
3. In the FROM clause as a table specification
4. In the SELECT clause as a column specification

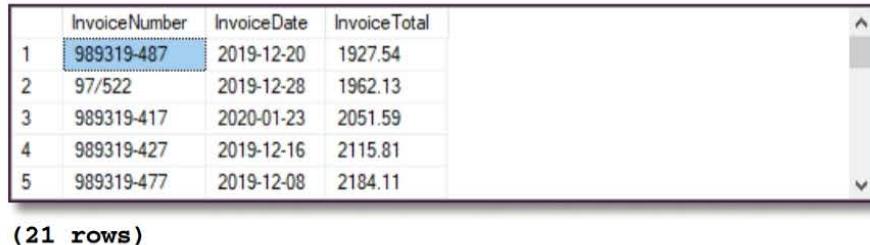
A SELECT statement that uses a subquery in the WHERE clause

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
WHERE InvoiceTotal >
    (SELECT AVG(InvoiceTotal)
     FROM Invoices)
ORDER BY InvoiceTotal;
```

The value returned by the subquery

1879.7413

The result set



The screenshot shows a table with three columns: InvoiceNumber, InvoiceDate, and InvoiceTotal. The data consists of five rows, each containing a unique invoice number, its date of issue, and its total amount. The first row is highlighted with a blue background.

	InvoiceNumber	InvoiceDate	InvoiceTotal
1	989319-487	2019-12-20	1927.54
2	97/522	2019-12-28	1962.13
3	989319-417	2020-01-23	2051.59
4	989319-427	2019-12-16	2115.81
5	989319-477	2019-12-08	2184.11

(21 rows)

Description

- A *subquery* is a SELECT statement that's coded within another SQL statement.
- A subquery can return a single value, a result set that contains a single column, or a result set that contains one or more columns.
- A subquery that returns a single value can be coded, or *introduced*, anywhere an expression is allowed. A subquery that returns a single column can be introduced in place of a list of values, such as the values for an IN phrase. And a subquery that returns one or more columns can be introduced in place of a table in the FROM clause.
- The syntax for a subquery is the same as for a standard SELECT statement. However, a subquery doesn't typically include the GROUP BY or HAVING clause, and it can't include an ORDER BY clause unless the TOP phrase is used.
- A subquery that's used in a WHERE or HAVING clause is called a *subquery search condition* or a *subquery predicate*. This is the most common use for a subquery.
- Although you can introduce a subquery in a GROUP BY or ORDER BY clause, you usually won't need to.
- Subqueries can be *nested* within other subqueries. However, subqueries that are nested more than two or three levels deep can be difficult to read and can result in poor performance.

How subqueries compare to joins

In the last figure, you saw an example of a subquery that returns an aggregate value that's used in the search condition of a WHERE clause. This type of subquery provides for processing that can't be done any other way. However, most subqueries can be restated as joins, and most joins can be restated as subqueries. This is illustrated by the SELECT statements in figure 6-2.

Both of the SELECT statements in this figure return a result set that consists of selected rows and columns from the Invoices table. In this case, only the invoices for vendors in California are returned. The first statement uses a join to combine the Vendors and Invoices table so the VendorState column can be tested for each invoice. By contrast, the second statement uses a subquery to return a result set that consists of the VendorID column for each vendor in California. Then, that result set is used with the IN operator in the search condition so that only invoices with a VendorID in that result set are included in the final result set.

So if you have a choice, which technique should you use? In general, I recommend you use the technique that results in the most readable code. For example, I think that a join tends to be more intuitive than a subquery when it uses an existing relationship between two tables. That's the case with the Vendors and Invoices tables used in the examples in this figure. On the other hand, a subquery tends to be more intuitive when it uses an ad hoc relationship.

As your queries get more complex, you may find that they're easier to code by using subqueries, regardless of the relationships that are involved. On the other hand, a query with an inner join typically performs faster than the same query with a subquery. So if system performance is an issue, you may want to use inner joins instead of subqueries.

You should also realize that when you use a subquery in a search condition, its results can't be included in the final result set. For instance, the second example in this figure can't be changed to include the VendorName column from the Vendors table. That's because the Vendors table isn't named in the FROM clause of the outer query. So if you need to include information from both tables in the result set, you need to use a join.

A query that uses an inner join

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
WHERE VendorState = 'CA'
ORDER BY InvoiceDate;
```

The same query restated with a subquery

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
WHERE VendorID IN
    (SELECT VendorID
    FROM Vendors
    WHERE VendorState = 'CA')
ORDER BY InvoiceDate;
```

The result set returned by both queries

	InvoiceNumber	InvoiceDate	InvoiceTotal
1	125520-1	2019-10-24	95.00
2	97/488	2019-10-24	601.95
3	111-92R-10096	2019-10-30	16.33
4	25022117	2019-11-01	6.00

(40 rows)

Advantages of joins

- The result of a join operation can include columns from both tables. The result of a query that includes a subquery can only include columns from the table named in the outer query. It can't include columns from the table named in the subquery.
- A join tends to be more intuitive when it uses an existing relationship between the two tables, such as a primary key to foreign key relationship.
- A query with a join typically performs faster than the same query with a subquery, especially if the query uses only inner joins.

Advantages of subqueries

- You can use a subquery to pass an aggregate value to the outer query.
- A subquery tends to be more intuitive when it uses an ad hoc relationship between the two tables.
- Long, complex queries can sometimes be easier to code using subqueries.

Description

- Like a join, a subquery can be used to code queries that work with two or more tables.
- Most subqueries can be restated as joins and most joins can be restated as subqueries.

How to code subqueries in search conditions

You can use a variety of techniques to work with a subquery in a search condition. You'll learn about those techniques in the topics that follow. As you read these topics, keep in mind that although all of the examples illustrate the use of subqueries in a WHERE clause, all of this information applies to the HAVING clause as well.

How to use subqueries with the IN operator

In chapter 3, you learned how to use the IN operator to test whether an expression is contained in a list of values. One way to provide that list of values is to use a subquery. This is illustrated in figure 6-3.

The example in this figure retrieves the vendors from the Vendors table that don't have invoices in the Invoices table. To do that, it uses a subquery to retrieve the VendorID of each vendor in the Invoices table. The result is a result set like the one shown in this figure that contains just the VendorID column. Then, this result set is used to filter the vendors that are included in the final result set.

You should notice two things about this subquery. First, it returns a single column. That's a requirement when a subquery is used with the IN operator. Second, the subquery includes the DISTINCT keyword. That way, if more than one invoice exists for a vendor, the VendorID for that vendor will be included only once. Note, however, that when the query is analyzed by SQL Server, this keyword will be added automatically. So you can omit it if you'd like to.

In the previous figure, you saw that a query that uses a subquery with the IN operator can be restated using an inner join. Similarly, a query that uses a subquery with the NOT IN operator can typically be restated using an outer join. The first query shown in this figure, for example, can be restated as shown in the second query. In this case, though, I think the query with the subquery is more readable. In addition, a query with a subquery will sometimes execute faster than a query with an outer join. That of course, depends on a variety of factors. In particular, it depends on the sizes of the tables and the relative number of unmatched rows. So if performance is an issue, you may want to test your query both ways to see which one executes faster.

The syntax of a WHERE clause that uses an IN phrase with a subquery

```
WHERE test_expression [NOT] IN (subquery)
```

A query that returns vendors without invoices

```
SELECT VendorID, VendorName, VendorState
FROM Vendors
WHERE VendorID NOT IN
    (SELECT DISTINCT VendorID
     FROM Invoices);
```

The result of the subquery

	VendorID
1	34
2	37
3	48
4	72
5	80
6	81

(34 rows)

The result set

	VendorID	VendorName	VendorState
32	33	Nielson	OH
33	35	Cal State Termite	CA
34	36	Graylift	CA
35	38	Venture Communications Int'l	NY
36	39	Custom Printing Company	MO
37	40	Nat Assoc of College Stores	OH

(88 rows)

The query restated without a subquery

```
SELECT Vendors.VendorID, Vendors.VendorName, Vendors.VendorState
FROM Vendors LEFT JOIN Invoices
    ON Vendors.VendorID = Invoices.VendorID
WHERE Invoices.VendorID IS NULL;
```

Description

- You can introduce a subquery with the IN operator to provide the list of values that are tested against the test expression.
- When you use the IN operator, the subquery must return a single column of values.
- A query that uses the NOT IN operator with a subquery can typically be restated using an outer join.

How to compare the result of a subquery with an expression

Figure 6-4 illustrates how you can use the comparison operators to compare an expression with the result of a subquery. In the example in this figure, the subquery returns the average balance due of the invoices in the Invoices table that have a balance due greater than zero. Then, it uses that value to retrieve all the invoices that have a balance due that's less than the average.

When you use a comparison operator as shown in this figure, the subquery must return a single value. In most cases, that means that it uses an aggregate function. However, you can also use the comparison operators with subqueries that return two or more values. To do that, you use the SOME, ANY, or ALL keyword to modify the comparison operator. You'll learn more about these keywords in the next two topics.

The syntax of a WHERE clause that compares an expression with the value returned by a subquery

```
WHERE expression comparison_operator [SOME|ANY|ALL] (subquery)
```

A query that returns invoices with a balance due less than the average

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,  
       InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue  
FROM Invoices  
WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0  
      AND InvoiceTotal - PaymentTotal - CreditTotal <  
          (SELECT AVG(InvoiceTotal - PaymentTotal - CreditTotal)  
           FROM Invoices  
          WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0)  
ORDER BY InvoiceTotal DESC;
```

The value returned by the subquery

2910.9472

The result set

	InvoiceNumber	InvoiceDate	InvoiceTotal	BalanceDue
1	31361833	2020-01-21	579.42	579.42
2	9982771	2020-01-24	503.20	503.20
3	547480102	2020-02-01	224.00	224.00
4	134116	2020-01-28	90.36	90.36
5	39104	2020-01-10	85.31	85.31

(9 rows)

Description

- You can use a comparison operator in a search condition to compare an expression with the results of a subquery.
- If you code a search condition without the ANY, SOME, and ALL keywords, the subquery must return a single value.
- If you include the ANY, SOME, or ALL keyword, the subquery can return a list of values. See figures 6-5 and 6-6 for more information on using these keywords.

Figure 6-4 How to compare the result of a subquery with an expression

How to use the ALL keyword

Figure 6-5 shows you how to use the ALL keyword. This keyword modifies the comparison operator so the condition must be true for all the values returned by a subquery. This is equivalent to coding a series of conditions connected by AND operators. The table at the top of this figure describes how this works for some of the comparison operators.

If you use the greater than operator ($>$), the expression must be greater than the maximum value returned by the subquery. Conversely, if you use the less than operator ($<$), the expression must be less than the minimum value returned by the subquery. If you use the equal operator ($=$), the expression must be equal to all of the values returned by the subquery. And if you use the not equal operator (\neq), the expression must not equal any of the values returned by the subquery. Note that a not equal condition could be restated using a NOT IN condition.

The query in this figure illustrates the use of the greater than operator with the ALL keyword. Here, the subquery selects the InvoiceTotal column for all the invoices with a VendorID value of 34. This results in a table with two rows, as shown in this figure. Then, the outer query retrieves the rows from the Invoices table that have invoice totals greater than all of the values returned by the subquery. In other words, this query returns all the invoices that have totals greater than the largest invoice for vendor number 34.

When you use the ALL operator, you should realize that if the subquery doesn't return any rows, the comparison operation will always be true. By contrast, if the subquery returns only null values, the comparison operation will always be false.

In many cases, a condition with the ALL keyword can be rewritten so it's easier to read and maintain. For example, the condition in the query in this figure could be rewritten to use the MAX function like this:

```
WHERE InvoiceTotal >
      (SELECT MAX(InvoiceTotal)
       FROM Invoices
       WHERE VendorID = 34)
```

Whenever you can, then, I recommend you replace the ALL keyword with an equivalent condition.

How the ALL keyword works

Condition	Equivalent expression	Description
<code>x > ALL (1, 2)</code>	<code>x > 2</code>	x must be greater than all the values returned by the subquery, which means it must be greater than the maximum value.
<code>x < ALL (1, 2)</code>	<code>x < 1</code>	x must be less than all the values returned by the subquery, which means it must be less than the minimum value.
<code>x = ALL (1, 2)</code>	<code>(x = 1) AND (x = 2)</code>	This condition can evaluate to True only if the subquery returns a single value or if all the values returned by the subquery are the same. Otherwise, it evaluates to False.
<code>x <> ALL (1, 2)</code>	<code>(x <> 1) AND (x <> 2)</code>	This condition is equivalent to: <code>x NOT IN (1, 2)</code>

A query that returns invoices larger than the largest invoice for vendor 34

```
SELECT VendorName, InvoiceNumber, InvoiceTotal
FROM Invoices JOIN Vendors ON Invoices.VendorID = Vendors.VendorID
WHERE InvoiceTotal > ALL
    (SELECT InvoiceTotal
     FROM Invoices
     WHERE VendorID = 34)
ORDER BY VendorName;
```

The result of the subquery

	InvoiceTotal
1	116.54
2	1083.58

The result set

	VendorName	InvoiceNumber	InvoiceTotal
1	Bertelsmann Industry Svcs. Inc	509786	6940.25
2	Cahners Publishing Company	587056	2184.50
3	Computerworld	367447	2433.00
4	Data Reproductions Corp	40318	21842.00
5	Dean Witter Reynolds	75C-90227	1367.50

(25 rows)

Description

- You can use the ALL keyword to test that a comparison condition is true for all of the values returned by a subquery. This keyword is typically used with the comparison operators `<`, `>`, `<=`, and `>=`.
- If no rows are returned by the subquery, a comparison that uses the ALL keyword is always true.
- If all of the rows returned by the subquery contain a null value, a comparison that uses the ALL keyword is always false.

Figure 6-5 How to use the ALL keyword

How to use the ANY and SOME keywords

Figure 6-6 shows how to use the ANY and SOME keywords. You use these keywords to test if a comparison is true for any, or some, of the values returned by a subquery. This is equivalent to coding a series of conditions connected with OR operators. Because these keywords are equivalent, you can use whichever one you prefer. The table at the top of this figure describes how these keywords work with some of the comparison operators.

The example in this figure shows how you can use the ANY keyword with the less than operator. This statement is similar to the one you saw in the previous figure, except that it retrieves invoices with invoice totals that are less than at least one of the invoice totals for a given vendor. Like the statement in the previous figure, this condition could be rewritten using the MAX function like this:

```
WHERE InvoiceTotal <
      (SELECT MAX(InvoiceTotal)
       FROM Invoices
      WHERE VendorID = 115)
```

Because you can usually replace an ANY condition with an equivalent condition that's more readable, you probably won't use ANY often.

How the ANY and SOME keywords work

Condition	Equivalent expression	Description
<code>x > ANY (1, 2)</code>	<code>x > 1</code>	<code>x</code> must be greater than at least one of the values returned by the subquery list, which means that it must be greater than the minimum value returned by the subquery.
<code>x < ANY (1, 2)</code>	<code>x < 2</code>	<code>x</code> must be less than at least one of the values returned by the subquery list, which means that it must be less than the maximum value returned by the subquery.
<code>x = ANY (1, 2)</code>	<code>(x = 1) OR (x = 2)</code>	This condition is equivalent to: <code>x IN (1, 2)</code>
<code>x <> ANY (1, 2)</code>	<code>(x <> 1) OR (x <> 2)</code>	This condition will evaluate to True for any non-empty result set containing at least one non-null value that isn't equal to <code>x</code> .

A query that returns invoices smaller than the largest invoice for vendor 115

```
SELECT VendorName, InvoiceNumber, InvoiceTotal
FROM Vendors JOIN Invoices ON Vendors.VendorID = Invoices.VendorID
WHERE InvoiceTotal < ANY
    (SELECT InvoiceTotal
     FROM Invoices
     WHERE VendorID = 115);
```

The result of the subquery

InvoiceTotal
1 6.00
2 6.00
3 25.67
4 6.00

The result set

VendorName	InvoiceNumber	InvoiceTotal
1 Abbey Office Furnishings	203339-13	17.50
2 Pacific Bell	111-92R-10096	16.33
3 Pacific Bell	111-92R-10097	16.33
4 Pacific Bell	111-92R-10094	19.67
5 Compuserve	21-4923721	9.95

(17 rows)

Description

- You can use the ANY or SOME keyword to test that a condition is true for one or more of the values returned by a subquery.
- ANY and SOME are equivalent keywords. SOME is the ANSI-standard keyword, but ANY is more commonly used.
- If no rows are returned by the subquery or all of the rows returned by the subquery contain a null value, a comparison that uses the ANY or SOME keyword is always false.

Figure 6-6 How to use the ANY and SOME keywords

How to code correlated subqueries

The subqueries you've seen so far in this chapter have been subqueries that are executed only once for the entire query. However, you can also code subqueries that are executed once for each row that's processed by the outer query. This type of query is called a *correlated subquery*, and it's similar to using a loop to do repetitive processing in a procedural programming language.

Figure 6-7 illustrates how correlated subqueries work. The example in this figure retrieves rows from the Invoices table for those invoices that have an invoice total that's greater than the average of all the invoices for the same vendor. To do that, the search condition in the WHERE clause of the subquery refers to the VendorID value of the current invoice. That way, only the invoices for the current vendor will be included in the average.

Each time a row in the outer query is processed, the value in the VendorID column for that row is substituted for the column reference in the subquery. Then, the subquery is executed based on the current value. If the VendorID value is 95, for example, this subquery will be executed:

```
SELECT AVG(InvoiceTotal)
  FROM Invoices AS Inv_Sub
 WHERE Inv_Sub.VendorID = 95;
```

After this subquery is executed, the value it returns is used to determine whether the current invoice is included in the result set. For example, the value returned by the subquery for vendor 95 is 28.5016. Then, that value is compared with the invoice total of the current invoice. If the invoice total is greater than that value, the invoice is included in the result set. Otherwise, it's not. This process is repeated until each of the invoices in the Invoices table has been processed.

As you study this example, notice how the column names in the WHERE clause of the inner query are qualified to indicate whether they refer to a column in the inner query or the outer query. In this case, the same table is used in both the inner and outer queries, so aliases, or correlation names, have been assigned to the tables. Then, those correlation names are used to qualify the column names. Although you have to qualify a reference to a column in the outer query, you don't have to qualify a reference to a column in the inner query. However, it's common practice to qualify both names, particularly if they refer to the same table.

Because correlated subqueries can be difficult to code, you may want to test a subquery separately before using it within another SELECT statement. To do that, however, you'll need to substitute a constant value for the variable that refers to a column in the outer query. That's what I did to get the average invoice total for vendor 95. Once you're sure that the subquery works on its own, you can replace the constant value with a reference to the outer query so you can use it within a SELECT statement.

A query that uses a correlated subquery to return each invoice that's higher than the vendor's average invoice

```
SELECT VendorID, InvoiceNumber, InvoiceTotal  
FROM Invoices AS Inv_Main  
WHERE InvoiceTotal >  
    (SELECT AVG(InvoiceTotal)  
     FROM Invoices AS Inv_Sub  
     WHERE Inv_Sub.VendorID = Inv_Main.VendorID)  
ORDER BY VendorID, InvoiceTotal;
```

The value returned by the subquery for vendor 95

28.5016

The result set

	VendorID	InvoiceNumber	InvoiceTotal
6	83	31359783	1575.00
7	95	111-92R-10095	32.70
8	95	111-92R-10093	39.77
9	95	111-92R-10092	46.21
10	110	P-0259	26881.40

(36 rows)

Description

- A *correlated subquery* is a subquery that is executed once for each row processed by the outer query. By contrast, a *noncorrelated subquery* is executed only once. All of the subqueries you've seen so far have been noncorrelated subqueries.
- A correlated subquery refers to a value that's provided by a column in the outer query. Because that value varies depending on the row that's being processed, each execution of the subquery returns a different result.
- To refer to a value in the outer query, a correlated subquery uses a qualified column name that includes the table name from the outer query. If the subquery uses the same table as the outer query, an alias, or *correlation name*, must be assigned to one of the tables to remove ambiguity.

Note

- Because a correlated subquery is executed once for each row processed by the outer query, a query with a correlated subquery typically takes longer to run than a query with a noncorrelated subquery.

How to use the EXISTS operator

Figure 6-8 shows you how to use the EXISTS operator with a subquery. This operator tests whether or not the subquery returns a result set. In other words, it tests whether the result set exists. When you use this operator, the subquery doesn't actually return a result set to the outer query. Instead, it simply returns an indication of whether any rows satisfy the search condition of the subquery. Because of that, queries that use this operator execute quickly.

You typically use the EXISTS operator with a correlated subquery as illustrated in this figure. This query retrieves all the vendors in the Vendors table that don't have invoices in the Invoices table. Notice that this query returns the same vendors as the two queries you saw in figure 6-3 that use the IN operator with a subquery and an outer join. However, the query in this figure executes more quickly than either of the queries in figure 6-3.

In this example, the correlated subquery selects all of the invoices that have the same VendorID value as the current vendor in the outer query. Because the subquery doesn't actually return a result set, it doesn't matter what columns are included in the SELECT clause. So it's customary to just code an asterisk. That way, SQL Server will determine what columns to select for optimum performance.

After the subquery is executed, the search condition in the WHERE clause of the outer query uses NOT EXISTS to test whether any invoices were found for the current vendor. If not, the vendor row is included in the result set. Otherwise, it's not.

The syntax of a subquery that uses the EXISTS operator

```
WHERE [NOT] EXISTS (subquery)
```

A query that returns vendors without invoices

```
SELECT VendorID, VendorName, VendorState  
FROM Vendors  
WHERE NOT EXISTS  
(SELECT *  
FROM Invoices  
WHERE Invoices.VendorID = Vendors.VendorID);
```

The result set

	VendorID	VendorName	VendorState
32	33	Nielson	OH
33	35	Cal State Termite	CA
34	36	Graylift	CA
35	38	Venture Communications Int'l	NY
36	39	Custom Printing Company	MO
37	40	Nat Assoc of College Stores	OH

(88 rows)

Description

- You can use the EXISTS operator to test that one or more rows are returned by the subquery. You can also use the NOT operator along with the EXISTS operator to test that no rows are returned by the subquery.
- When you use the EXISTS operator with a subquery, the subquery doesn't actually return any rows. Instead, it returns an indication of whether any rows meet the specified condition.
- Because no rows are returned by the subquery, it doesn't matter what columns you specify in the SELECT clause. So you typically just code an asterisk (*).
- Although you can use the EXISTS operator with either a correlated or a noncorrelated subquery, it's used most often with correlated subqueries. That's because it's usually better to use a join than a noncorrelated subquery with EXISTS.

Other ways to use subqueries

Although you'll typically use subqueries in the WHERE or HAVING clause of a SELECT statement, you can also use them in the FROM and SELECT clauses. You'll learn how to do that in the topics that follow.

How to code subqueries in the FROM clause

Figure 6-9 shows you how to code a subquery in a FROM clause. As you can see, you can code a subquery in place of a table specification. In this example, the results of the subquery, called a *derived table*, are joined with another table. When you use a subquery in this way, it can return any number of rows and columns.

Subqueries are typically used in the FROM clause to create derived tables that provide summarized data to a summary query. The subquery in this figure, for example, creates a derived table that contains the VendorID values and the average invoice totals for the five vendors with the top invoice averages. To do that, it groups the invoices by VendorID, sorts them in descending sequence by average invoice total, and then returns the top five rows. The derived table is then joined with the Invoices table, and the resulting rows are grouped by VendorID. Finally, the maximum invoice date and average invoice total are calculated for the grouped rows, and the results are sorted by the maximum invoice date in descending sequence.

You should notice four things about this query. First, the derived table is assigned a table alias so it can be referred to from the outer query. Second, the result of the AVG function in the subquery is assigned a column alias. This is because a derived table can't have unnamed columns. Third, since the subquery uses a TOP phrase, it also includes an ORDER BY clause. Fourth, although you might think that you could use the average invoice totals calculated by the subquery in the select list of the outer query, you can't. That's because the outer query includes a GROUP BY clause, so only aggregate functions, columns named in the GROUP BY clause, and constant values can be included in this list. Because of that, the AVG function is repeated in the select list.

When used in the FROM clause, a subquery is similar to a view. As you learned in chapter 1, a view is a predefined SELECT statement that's saved with the database. Because it's saved with the database, a view typically performs more efficiently than a derived table. However, it's not always practical to use a view. In those cases, derived tables can be quite useful. In addition, derived tables can be useful for testing possible solutions before creating a view. Then, once the derived table works the way you want it to, you can define the view based on the subquery you used to create the derived table.

A query that uses a derived table to retrieve the top 5 vendors by average invoice total

```
SELECT Invoices.VendorID, MAX(InvoiceDate) AS LatestInv,
       AVG(InvoiceTotal) AS AvgInvoice
  FROM Invoices JOIN
       (SELECT TOP 5 VendorID, AVG(InvoiceTotal) AS AvgInvoice
        FROM Invoices
        GROUP BY VendorID
        ORDER BY AvgInvoice DESC) AS TopVendor
    ON Invoices.VendorID = TopVendor.VendorID
 GROUP BY Invoices.VendorID
 ORDER BY LatestInv DESC;
```

The derived table generated by the subquery

	VendorID	AvgInvoice
1	110	23978.482
2	72	10963.655
3	104	7125.34
4	99	6940.25
5	119	4901.26

The result set

	VendorID	LatestInv	AvgInvoice
1	110	2020-01-31	23978.482
2	72	2020-01-10	10963.655
3	99	2019-12-18	6940.25
4	104	2019-11-21	7125.34
5	119	2019-11-11	4901.26

Description

- A subquery that's coded in the FROM clause returns a result set called a *derived table*. When you create a derived table, you must assign an alias to it. Then, you can use the derived table within the outer query just as you would any other table.
- When you code a subquery in the FROM clause, you must assign names to any calculated values in the result set.
- Derived tables are most useful when you need to further summarize the results of a summary query.
- A derived table is like a view in that it retrieves selected rows and columns from one or more base tables. Because views are stored as part of the database, they're typically more efficient to use than derived tables. However, it may not always be practical to construct and save a view in advance.

How to code subqueries in the SELECT clause

Figure 6-10 shows you how to use subqueries in the SELECT clause. As you can see, you can use a subquery in place of a column specification. Because of that, the subquery must return a single value.

In most cases, the subqueries you use in the SELECT clause will be correlated subqueries. The subquery in this figure, for example, calculates the maximum invoice date for each vendor in the Vendors table. To do that, it refers to the VendorID column from the Invoices table in the outer query.

Because subqueries coded in the SELECT clause are difficult to read, and because correlated subqueries are typically inefficient, you shouldn't use them unless you can't find another solution. In most cases, though, you can replace the subquery with a join. The first query shown in this figure, for example, could be restated as shown in the second query. This query joins the Vendors and Invoices tables, groups the rows by VendorName, and then uses the MAX function to calculate the maximum invoice date for each vendor. As you can see, this query is much easier to read than the one with the subquery. It will also execute more quickly.

A query that uses a correlated subquery in its SELECT clause to retrieve the most recent invoice for each vendor

```
SELECT DISTINCT VendorName,
    (SELECT MAX(InvoiceDate) FROM Invoices
     WHERE Invoices.VendorID = Vendors.VendorID) AS LatestInv
  FROM Vendors
 ORDER BY LatestInv DESC;
```

The result set

	VendorName	LatestInv
1	Federal Express Corporation	2020-02-02
2	Blue Cross	2020-02-01
3	Malloy Lithographing Inc	2020-01-31
4	Cardinal Business Media, Inc.	2020-01-28
5	Zylka Design	2020-01-25
6	Ford Motor Credit Company	2020-01-24
7	United Parcel Service	2020-01-24
8	Ingram	2020-01-21
9	Wakefield Co	2020-01-20

(122 rows)

The same query restated using a join

```
SELECT VendorName, MAX(InvoiceDate) AS LatestInv
  FROM Vendors LEFT JOIN Invoices ON Vendors.VendorID = Invoices.VendorID
 GROUP BY VendorName
 ORDER BY LatestInv DESC;
```

Description

- When you code a subquery for a column specification in the SELECT clause, the subquery must return a single value.
- A subquery that's coded within a SELECT clause is typically a correlated subquery.
- A query that includes a subquery in its SELECT clause can typically be restated using a join instead of the subquery. Because a join is usually faster and more readable, subqueries are seldom coded in the SELECT clause.

Figure 6-10 How to code subqueries in the SELECT clause

Guidelines for working with complex queries

So far, the examples you've seen of queries that use subqueries have been relatively simple. However, these types of queries can get complicated in a hurry, particularly if the subqueries are nested. Because of that, you'll want to be sure that you plan and test these queries carefully. You'll learn a procedure for doing that in a moment. But first, you'll see a complex query that illustrates the type of query I'm talking about.

A complex query that uses subqueries

Figure 6-11 presents a query that uses three subqueries. The first subquery is used in the FROM clause of the outer query to create a derived table that contains the state, name, and total invoice amount for each vendor in the Vendors table. The second subquery is also used in the FROM clause of the outer query to create a derived table that's joined with the first table. This derived table contains the state and total invoice amount for the vendor in each state that has the largest invoice total. To create this table, a third subquery is nested within the FROM clause of the subquery. This subquery is identical to the first subquery.

After the two derived tables are created, they're joined based on the columns in each table that contain the state and the total invoice amount. The final result set includes the state, name, and total invoice amount for the vendor in each state with the largest invoice total. This result set is sorted by state.

As you can see, this query is quite complicated and difficult to understand. In fact, you might be wondering if there isn't an easier solution to this problem. For example, you might think that you could solve the problem simply by joining the Vendors and Invoices table and creating a grouped aggregate. If you grouped by vendor state, however, you wouldn't be able to include the name of the vendor in the result set. And if you grouped by vendor state and vendor name, the result set would include all the vendors, not just the vendor from each state with the largest invoice total.

If you think about how else you might solve this query, I think you'll agree that the solution presented here is fairly straightforward. However, in figure 6-13, you'll learn how to use a feature called common table expressions to simplify this query. In particular, this feature allows you to code a single Summary subquery instead of coding the Summary1 and Summary2 subqueries shown here.

A query that uses three subqueries

```

SELECT Summary1.VendorState, Summary1.VendorName, TopInState.SumOfInvoices
FROM
    (SELECT V_Sub.VendorState, V_Sub.VendorName,
        SUM(I_Sub.InvoiceTotal) AS SumOfInvoices
    FROM Invoices AS I_Sub JOIN Vendors AS V_Sub
        ON I_Sub.VendorID = V_Sub.VendorID
    GROUP BY V_Sub.VendorState, V_Sub.VendorName) AS Summary1
JOIN
    (SELECT Summary2.VendorState,
        MAX(Summary2.SumOfInvoices) AS SumOfInvoices
    FROM
        (SELECT V_Sub.VendorState, V_Sub.VendorName,
            SUM(I_Sub.InvoiceTotal) AS SumOfInvoices
        FROM Invoices AS I_Sub JOIN Vendors AS V_Sub
            ON I_Sub.VendorID = V_Sub.VendorID
        GROUP BY V_Sub.VendorState, V_Sub.VendorName) AS Summary2
    GROUP BY Summary2.VendorState) AS TopInState
ON Summary1.VendorState = TopInState.VendorState AND
Summary1.SumOfInvoices = TopInState.SumOfInvoices
ORDER BY Summary1.VendorState;

```

The result set

	VendorState	VendorName	SumOfInvoices
1	AZ	Wells Fargo Bank	662.00
2	CA	Digital Dreamworks	7125.34
3	DC	Reiter's Scientific & Pro Books	600.00
4	MA	Dean Witter Reynolds	1367.50
5	MI	Malloy Lithographing Inc	119892.41
6	NV	United Parcel Service	23177.96
7	OH	Edward Data Services	207.78
8	PA	Cardinal Business Media, Inc.	265.36

(10 rows)

How the query works

- This query retrieves the vendor from each state that has the largest invoice total. To do that, it uses three subqueries: Summary1, Summary2, and TopInState. The Summary1 and TopInState subqueries are joined together in the FROM clause of the outer query, and the Summary2 subquery is nested within the FROM clause of the TopInState subquery.
- The Summary1 and Summary2 subqueries are identical. They join data from the Vendors and Invoices tables and produce a result set that includes the sum of invoices for each vendor grouped by vendor name within state.
- The TopInState subquery produces a result set that includes the vendor state and the largest sum of invoices for any vendor in that state. This information is retrieved from the results of the Summary2 subquery.
- The columns listed in the SELECT clause of the outer query are retrieved from the result of the join between the Summary1 and TopInState subqueries, and the results are sorted by state.

Figure 6-11 A complex query that uses subqueries

A procedure for building complex queries

To build a complex query like the one in the previous figure, you can use a procedure like the one in figure 6-12. To start, you should state the problem to be solved so you're clear about what you want the query to accomplish. In this case, the question is, "Which vendor in each state has the largest invoice total?"

Once you're clear about the problem, you should outline the query using *pseudocode*. Pseudocode is simply code that represents the intent of the query, but doesn't necessarily use SQL code. The pseudocode shown in this figure, for example, uses part SQL code and part English. Notice that this pseudocode identifies the two main subqueries. Because these subqueries define derived tables, the pseudocode also indicates the alias that will be used for each: Summary1 and TopInState. That way, you can use these aliases in the pseudocode for the outer query to make it clear where the data it uses comes from.

If it's not clear from the pseudocode how each subquery will be coded, or, as in this case, if a subquery is nested within another subquery, you can also write pseudocode for the subqueries. For example, the pseudocode for the TopInState query is presented in this figure. Because this subquery has a subquery nested in its FROM clause, that subquery is identified in this pseudocode as Summary2.

The next step in the procedure is to code and test the actual subqueries to be sure they work the way you want them to. For example, the code for the Summary1 and Summary2 queries is shown in this figure, along with the results of these queries and the results of the TopInState query. Once you're sure that the subqueries work the way you want them to, you can code and test the final query.

If you follow the procedure presented in this figure, I think you'll find it easier to build complex queries that use subqueries. Before you can use this procedure, of course, you need to have a thorough understanding of how subqueries work and what they can do. So you'll want to be sure to experiment with the techniques you learned in this chapter before you try to build a complex query like the one shown here.

A procedure for building complex queries

1. State the problem to be solved by the query in English.
2. Use pseudocode to outline the query. The pseudocode should identify the subqueries used by the query and the data they return. It should also include aliases used for any derived tables.
3. If necessary, use pseudocode to outline each subquery.
4. Code the subqueries and test them to be sure that they return the correct data.
5. Code and test the final query.

The problem to be solved by the query in figure 6-11

- Which vendor in each state has the largest invoice total?

Pseudocode for the query

```
SELECT Summary1.VendorState, Summary1.VendorName, TopInState.SumOfInvoices
FROM (Derived table returning VendorState, VendorName, SumOfInvoices)
      AS Summary1
  JOIN (Derived table returning VendorState, MAX(SumOfInvoices))
        AS TopInState
  ON Summary1.VendorState = TopInState.VendorState AND
     Summary1.SumOfInvoices = TopInState.SumOfInvoices
ORDER BY Summary1.VendorState;
```

Pseudocode for the TopInState subquery

```
SELECT Summary2.VendorState, MAX(Summary2.SumOfInvoices)
FROM (Derived table returning VendorState, VendorName, SumOfInvoices)
      AS Summary2
GROUP BY Summary2.VendorState;
```

The code for the Summary1 and Summary2 subqueries

```
SELECT V_Sub.VendorState, V_Sub.VendorName,
       SUM(I_Sub.InvoiceTotal) AS SumOfInvoices
  FROM Invoices AS I_Sub JOIN Vendors AS V_Sub
    ON I_Sub.VendorID = V_Sub.VendorID
 GROUP BY V_Sub.VendorState, V_Sub.VendorName;
```

The result of the Summary1 and Summary2 subqueries

VendorState	VendorName	SumOfInvoices
10 MA	Dean Witter Reynolds	1367.50
11 CA	Digital Dreamworks	7125.34
12 CA	Driftas Groom & McCormick	220.00
13 OH	Edward Data Services	207.78

(34 rows)

The result of the TopInState subquery

VendorState	SumOfInvoices
1 AZ	662.00
2 CA	7125.34
3 DC	600.00
4 MA	1367.50

(10 rows)

Figure 6-12 A procedure for building complex queries

How to work with common table expressions

A *common table expression (CTE)* is a feature that allows you to code an expression that defines a derived table. You can use CTEs to simplify complex queries that use subqueries. This can make your code easier to read and maintain. In addition, you can use a CTE to loop through nested structures.

How to code a CTE

Figure 6-13 shows how to use a CTE to simplify the complex query presented in figure 6-11. To start, the statement for the query begins with the WITH keyword to indicate that you are about to define a CTE. Then, it specifies Summary as the name for the first table, followed by the AS keyword, followed by an opening parenthesis, followed by a SELECT statement that defines the table, followed by a closing parenthesis. In this figure, for example, this statement returns the same result set as the subqueries named Summary1 and Summary2 that were presented in figure 6-11.

After the first CTE is defined, this example continues by defining a second CTE named TopInState. To start, a comma is coded to separate the two CTEs. Then, this query specifies TopInState as the name for the second table, followed by the AS keyword, followed by an opening parenthesis, followed by a SELECT statement that defines the table, followed by a closing parenthesis. Here, this SELECT statement refers to the Summary table that was defined by the first CTE. When coding multiple CTEs like this, a CTE can refer to any CTEs in the same WITH clause that are coded before it, but it can't refer to CTEs coded after it. As a result, this statement wouldn't work if the two CTEs were coded in the reverse order.

Finally, the SELECT statement that's coded immediately after the two CTEs uses both of these CTEs just as if they were tables. To do that, this SELECT statement joins the two tables, specifies the columns to retrieve, and specifies the sort order. To avoid ambiguous references, each column is qualified by the name for the CTE. If you compare figure 6-13 with figure 6-11, I think you'll agree that the code in figure 6-13 is easier to read. That's partly because the tables defined by the subqueries aren't nested within the SELECT statement. In addition, I think you'll agree that the code in figure 6-13 is easier to maintain. That's because this query reduces code duplication by only coding the Summary query in one place, not in two places.

When using the syntax shown here to define CTEs, you must supply distinct names for all columns defined by the SELECT statement, including calculated values. That way, it's possible for other statements to refer to the columns in the result set. Most of the time, that's all you need to know to be able to work with CTEs. For more information about working with CTEs, you can look up "WITH common_table_expression" in the documentation for SQL Server.

The syntax of a CTE

```
WITH cte_name1 AS (query_definition1)
[, cte_name2 AS (query_definition2)]
[...]
sql_statement
```

Two CTEs and a query that uses them

```
WITH Summary AS
(
    SELECT VendorState, VendorName, SUM(InvoiceTotal) AS SumOfInvoices
    FROM Invoices
    JOIN Vendors ON Invoices.VendorID = Vendors.VendorID
    GROUP BY VendorState, VendorName
),
TopInState AS
(
    SELECT VendorState, MAX(SumOfInvoices) AS SumOfInvoices
    FROM Summary
    GROUP BY VendorState
)
SELECT Summary.VendorState, Summary.VendorName, TopInState.SumOfInvoices
FROM Summary JOIN TopInState
    ON Summary.VendorState = TopInState.VendorState AND
        Summary.SumOfInvoices = TopInState.SumOfInvoices
ORDER BY Summary.VendorState;
```

The result set

	VendorState	VendorName	SumOfInvoices
1	AZ	Wells Fargo Bank	662.00
2	CA	Digital Dreamworks	7125.34
3	DC	Reiter's Scientific & Pro Books	600.00
4	MA	Dean Witter Reynolds	1367.50
5	MI	Malloy Lithographing Inc	119892.41
6	NV	United Parcel Service	23177.96
7	OH	Edward Data Services	207.78
8	PA	Cardinal Business Media, Inc.	265.36

(10 rows)

Description

- A *common table expression (CTE)* is an expression (usually a SELECT statement) that creates one or more temporary tables that can be used by the following query.
- To use a CTE with a query, you code the WITH keyword followed by the definition of the CTE. Then, immediately after the CTE, you code the statement that uses it.
- To code multiple CTEs, separate them with commas. Then, each CTE can refer to itself and any previously defined CTEs in the same WITH clause.
- You can use CTEs with SELECT, INSERT, UPDATE, and DELETE statements. However, you're most likely to use them with SELECT statements as shown in this figure and in figure 6-14.

Figure 6-13 How to code a CTE

How to code a recursive CTE

A *recursive query* is a query that is able to loop through a result set and perform processing to return a final result set. Recursive queries are commonly used to return hierarchical data such as an organizational chart in which a parent element may have one or more child elements, and each child element may have one or more child elements. To code a recursive query, you can use a *recursive CTE*. Figure 6-14 shows how.

The top of this figure shows an Employees table where the ManagerID column is used to identify the manager for each employee. Here, Cindy Smith is the top level manager since she doesn't have a manager, Elmer Jones and Paulo Locario report to Cindy, and so on.

The recursive CTE shown in this figure returns each employee according to their level in the organization chart for the company. To do that, this statement begins by defining a CTE named EmployeesCTE. Within this CTE, two SELECT statements are joined by the UNION ALL operator. Here, the first SELECT statement uses the IS NULL operator to return the first row of the result set. This statement is known as the *anchor member* of the recursive CTE.

Then, the second SELECT statement creates a loop by referencing itself. In particular, this query joins the Employees table to the EmployeesCTE table that's defined by the CTE. This statement is known as the *recursive member* and it loops through each row in the Employees table. With each loop, it adds 1 to the rank column and appends the current result set to the final result set. For example, on the first loop, it appends Elmer Jones and Paulo Locario to the final result set. On the second loop, it appends Ralph Simonian, Thomas Hardy, Olivia Hernandez, and Rhea O'Leary to the final result set. And so on.

When coding a recursive CTE, you must follow some rules. First, you must supply a name for each column defined by the CTE. To do that, you just need to make sure to specify a name for each column in the anchor member. Second, the rules for coding a union that you learned in chapter 4 still apply. In particular, the anchor member and the recursive member must have the same number of columns and the columns must have compatible data types.

Most of the time, that's all you need to know to be able to work with recursive CTEs. However, the goal of this figure is to show a simple recursive CTE to give you a general idea of how they work. If necessary, you can code much more complex recursive CTEs. For example, you can code multiple anchor members and multiple recursive members. For more information about working with recursive CTEs, you can start by looking up "WITH common_table_expression" in the documentation for SQL Server.

If you find that you're often using recursive CTEs to return hierarchical data, you may want to learn more about the hierarchyid data type that was introduced with SQL Server 2008. This data type makes it easier to work with hierarchical data such as organization charts. To learn more about this data type, you can look up "hierarchyid (Transact-SQL)" in the documentation for SQL Server.

The Employees table

	EmployeeID	LastName	FirstName	ManagerID
1	1	Smith	Cindy	NULL
2	2	Jones	Elmer	1
3	3	Simonian	Ralph	2
4	4	Hemandez	Olivia	2
5	5	Aaronsen	Robert	3
6	6	Watson	Denise	3
7	7	Hardy	Thomas	2
8	8	O'Leary	Rhea	2
9	9	Locario	Paulo	1

A recursive CTE that returns hierarchical data

```
WITH EmployeesCTE AS
(
    -- Anchor member
    SELECT EmployeeID,
        FirstName + ' ' + LastName As EmployeeName,
        1 As Rank
    FROM Employees
    WHERE ManagerID IS NULL
    UNION ALL
    -- Recursive member
    SELECT Employees.EmployeeID,
        FirstName + ' ' + LastName,
        Rank + 1
    FROM Employees
    JOIN EmployeesCTE
        ON Employees.ManagerID = EmployeesCTE.EmployeeID
)
SELECT *
FROM EmployeesCTE
ORDER BY Rank, EmployeeID;
```

The final result set

	EmployeeID	EmployeeName	Rank
1	1	Cindy Smith	1
2	2	Elmer Jones	2
3	9	Paulo Locario	2
4	3	Ralph Simonian	3
5	4	Olivia Hemandez	3
6	7	Thomas Hardy	3
7	8	Rhea O'Leary	3
8	5	Robert Aaronsen	4
9	6	Denise Watson	4

Description

- A *recursive query* is a query that is able to loop through a result set and perform processing to return a final result set. A *recursive CTE* can be used to create a recursive query.
- A recursive CTE must contain at least two query definitions, an *anchor member* and a *recursive member*, and these members must be connected by the UNION ALL operator.

Figure 6-14 How to code a recursive CTE

Perspective

As you've seen in this chapter, subqueries provide a powerful tool for solving difficult problems. Before you use a subquery, however, remember that a subquery can often be restated more clearly by using a join. In addition, a query with a join often executes more quickly than a query with a subquery. Because of that, you'll typically use a subquery only when it can't be restated as a join or when it makes the query easier to understand without slowing it down significantly.

If you find yourself coding the same subqueries over and over, you should consider creating a view for that subquery as described in chapter 13. This will help you develop queries more quickly since you can use the view instead of coding the subquery again. In addition, since views execute more quickly than subqueries, this may improve the performance of your queries.

Terms

subquery	derived table
introduce a subquery	pseudocode
subquery search condition	common table expression (CTE)
subquery predicate	recursive query
nested subquery	recursive CTE
correlated subquery	anchor member
noncorrelated subquery	recursive member

Exercises

1. Write a SELECT statement that returns the same result set as this SELECT statement. Substitute a subquery in a WHERE clause for the inner join.

```
SELECT DISTINCT VendorName  
FROM Vendors JOIN Invoices  
    ON Vendors.VendorID = Invoices.VendorID  
ORDER BY VendorName;
```

2. Write a SELECT statement that answers this question: Which invoices have a PaymentTotal that's greater than the average PaymentTotal for all paid invoices? Return the InvoiceNumber and InvoiceTotal for each invoice.
3. Write a SELECT statement that answers this question: Which invoices have a PaymentTotal that's greater than the median PaymentTotal for all paid invoices? (The median marks the midpoint in a set of values; an equal number of values lie above and below it.) Return the InvoiceNumber and InvoiceTotal for each invoice.

Hint: Begin with the solution to exercise 2, then use the ALL keyword in the WHERE clause and code "TOP 50 PERCENT PaymentTotal" in the subquery.

4. Write a SELECT statement that returns two columns from the GLAccounts table: AccountNo and AccountDescription. The result set should have one row for each account number that has never been used. Use a correlated subquery introduced with the NOT EXISTS operator. Sort the final result set by AccountNo.
5. Write a SELECT statement that returns four columns: VendorName, InvoiceID, InvoiceSequence, and InvoiceLineItemAmount for each invoice that has more than one line item in the InvoiceLineItems table.

Hint: Use a subquery that tests for `InvoiceSequence > 1`.
6. Write a SELECT statement that returns a single value that represents the sum of the largest unpaid invoices submitted by each vendor. Use a derived table that returns `MAX(InvoiceTotal)` grouped by `VendorID`, filtering for invoices with a balance due.
7. Write a SELECT statement that returns the name, city, and state of each vendor that's located in a unique city and state. In other words, don't include vendors that have a city and state in common with another vendor.
8. Write a SELECT statement that returns four columns: VendorName, InvoiceNumber, InvoiceDate, and InvoiceTotal. Return one row per vendor, representing the vendor's invoice with the earliest date.
9. Rewrite exercise 6 so it uses a common table expression (CTE) instead of a derived table.

How to insert, update, and delete data

In the last four chapters, you learned how to code the SELECT statement to retrieve and summarize data. Now, you'll learn how to code the INSERT, UPDATE, and DELETE statements to modify the data in a table. When you're done with this chapter, you'll know how to code the four statements that are used every day by professional SQL programmers.

How to create test tables	216
How to use the SELECT INTO statement	216
How to use a copy of the database.....	216
How to insert new rows	218
How to insert a single row	218
How to insert multiple rows.....	218
How to insert default values and null values.....	220
How to insert rows selected from another table	222
How to modify existing rows.....	224
How to perform a basic update operation.....	224
How to use subqueries in an update operation.....	226
How to use joins in an update operation	228
How to delete existing rows	230
How to perform a basic delete operation.....	230
How to use subqueries and joins in a delete operation.....	232
How to merge rows	234
How to perform a basic merge operation	234
How to code more complex merge operations	234
Perspective	236

How to create test tables

As you learn to code INSERT, UPDATE, and DELETE statements, you need to make sure that your experimentation won't affect "live" data or a classroom database that is shared by other students. Two ways to get around that are presented next.

How to use the **SELECT INTO** statement

Figure 7-1 shows how to use the SELECT INTO statement to create test tables that are derived from the tables in a database. Then, you can experiment all you want with the test tables and delete them when you're done. When you use the SELECT INTO statement, the result set that's defined by the SELECT statement is simply copied into a new table.

The three examples in this figure show some of the ways you can use this statement. Here, the first example copies all of the columns from all of the rows in the Invoices table into a new table named InvoiceCopy. The second example copies all of the columns in the Invoices table into a new table, but only for rows where the balance due is zero. And the third example creates a table that contains summary data from the Invoices table.

For the examples in the rest of this chapter, I used the SELECT INTO statement to make copies of the Vendors and Invoices tables, and I named these tables VendorCopy and InvoiceCopy. If you do the same, you'll avoid corrupting the original database. Then, when you're done experimenting, you can use the DROP TABLE statement that's shown in this figure to delete the test tables.

When you use this technique to create tables, though, only the column definitions and data are copied, which means that definitions like those of primary keys, foreign keys, and default values aren't retained. As a result, the test results that you get with the copied tables may be slightly different than the results you would get with the original tables. You'll understand that better after you read chapters 10 and 11.

How to use a copy of the database

If you download the files for this book as described in appendix A, you can create copies of the databases used in this book on your local server by running the provided database creation scripts. As a result, you can modify the tables within these databases without worrying about how much you change them. Then, when you're done testing, you can restore these databases by running the database creation scripts again.

However, when you create a copy of the entire database instead of making copies of tables within a database, the definitions of primary keys, foreign keys, and default values are retained, so your results may be slightly different than the ones shown in the examples. If, for example, you try to add a row with an invalid foreign key, SQL Server won't let you do that. You'll learn more about that in chapters 10 and 11.

The syntax of the SELECT INTO statement

```
SELECT select_list  
INTO table_name  
FROM table_source  
[WHERE search_condition]  
[GROUP BY group_by_list]  
[HAVING search_condition]  
[ORDER BY order_by_list]
```

A statement that creates a complete copy of the Invoices table

```
SELECT *  
INTO InvoiceCopy  
FROM Invoices;  
(114 rows affected)
```

A statement that creates a partial copy of the Invoices table

```
SELECT *  
INTO OldInvoices  
FROM Invoices  
WHERE InvoiceTotal - PaymentTotal - CreditTotal = 0;  
(103 rows affected)
```

A statement that creates a table with summary rows

```
SELECT VendorID, SUM(InvoiceTotal) AS SumOfInvoices  
INTO VendorBalances  
FROM Invoices  
WHERE InvoiceTotal - PaymentTotal - CreditTotal <> 0  
GROUP BY VendorID;  
(7 rows affected)
```

A statement that deletes a table

```
DROP TABLE InvoiceCopy;
```

Description

- The INTO clause is a SQL Server extension that lets you create a new table based on the result set defined by the SELECT statement. Since the definitions of the columns in the new table are based on the columns in the result set, the column names assigned in the SELECT clause must be unique.
- You can code the other clauses of the SELECT INTO statement just as you would for any other SELECT statement.
- If you use calculated values in the select list, you must name the column since that name is used in the definition of the new table.
- The table you name in the INTO clause must not exist. If it does, you must delete it using the DROP TABLE statement before you execute the SELECT INTO statement.

Warning

- When you use the SELECT INTO statement to create a table, only the column definitions and data are copied. That means that definitions of primary keys, foreign keys, indexes, default values, and so on are not included in the new table.

How to insert new rows

To add new rows to a table, you use the INSERT statement. This statement lets you insert a single row or multiple rows.

How to insert a single row

Figure 7-2 shows how to code an INSERT statement to insert a single row. The two examples in this figure insert a row into the InvoiceCopy table. The data this new row contains is defined near the top of this figure.

In the first example, you can see that you name the table in which the row will be inserted in the INSERT clause. Then, the VALUES clause lists the values to be used for each column. You should notice three things about this list. First, it includes a value for every column in the table except for the InvoiceID column. This value is omitted because the InvoiceID column is defined as an identity column. Because of that, its value will be generated by SQL Server. Second, the values are listed in the same sequence that the columns appear in the table. That way, SQL Server knows which value to assign to which column. And third, a null value is assigned to the last column, PaymentDate, using the NULL keyword. You'll learn more about using this keyword in the next topic.

The second INSERT statement in this figure includes a column list in the INSERT clause. Notice that this list doesn't include the PaymentDate column since it allows a null value. In addition, the columns aren't listed in the same sequence as the columns in the InvoiceCopy table. When you include a list of columns, you can code the columns in any sequence you like. Then, you just need to be sure that the values in the VALUES clause are coded in the same sequence.

When you specify the values for the columns to be inserted, you must be sure that those values are compatible with the data types of the columns. For example, you must enclose literal values for dates and strings within single quotes. However, you don't need to enclose literal values for numbers in single quotes. You'll learn more about data types and how to work with them in the next chapter. For now, just realize that if any of the values aren't compatible with the data types of the corresponding columns, an error will occur and the row won't be inserted.

How to insert multiple rows

SQL Server 2008 extended the syntax for the INSERT statement to allow a single INSERT statement to insert multiple rows. To do that, you just use a comma to separate the multiple value lists as shown in the third INSERT statement in figure 7-2. Although this syntax doesn't provide a performance gain, it does provide a more concise way to write the code than coding multiple INSERT statements.

The syntax of the INSERT statement

```
INSERT [INTO] table_name [(column_list)]
[DEFAULT] VALUES (expression_1 [, expression_2]...)
[, (expression_1 [, expression_2]...)...]
```

The values for a new row to be added to the Invoices table

Column	Value	Column	Value
InvoiceID	(Next available unique ID)	PaymentTotal	0
VendorID	97	CreditTotal	0
InvoiceNumber	456789	TermsID	1
InvoiceDate	3/01/2020	InvoiceDueDate	3/31/2020
InvoiceTotal	8,344.50	PaymentDate	null

An INSERT statement that adds the new row without using a column list

```
INSERT INTO InvoiceCopy
VALUES (97, '456789', '2020-03-01', 8344.50, 0, 0, 1, '2020-03-31', NULL);
```

An INSERT statement that adds the new row using a column list

```
INSERT INTO InvoiceCopy
    (VendorID, InvoiceNumber, InvoiceTotal, PaymentTotal, CreditTotal,
     TermsID, InvoiceDate, InvoiceDueDate)
VALUES
    (97, '456789', 8344.50, 0, 0, 1, '2030-03-01', '2020-03-31');
```

The response from the system

(1 row affected)

An INSERT statement that adds three new rows

```
INSERT INTO InvoiceCopy
VALUES
    (95, '111-10098', '2020-03-01', 219.50, 0, 0, 1, '2020-03-31', NULL),
    (102, '109596', '2020-03-01', 22.97, 0, 0, 1, '2020-03-31', NULL),
    (72, '40319', '2020-03-01', 173.38, 0, 0, 1, '2020-03-31', NULL);
```

The response from the system

(3 rows affected)

Description

- You specify the values to be inserted in the VALUES clause. The values you specify depend on whether you include a column list.
- If you don't include a column list, you must specify the column values in the same order as they appear in the table, and you must code a value for each column in the table. The exception is an identity column, which must be omitted.
- If you include a column list, you must specify the column values in the same order as they appear in the column list. You can omit columns with default values and columns that accept null values, and you must omit identity columns.

How to insert default values and null values

If a column allows null values, you'll want to know how to insert a null value into that column. Similarly, if a column is defined with a default value, you'll want to know how to insert that value. The technique you use depends on whether the INSERT statement includes a column list, as shown by the examples in figure 7-3.

All of these INSERT statements use a table named ColorSample. This table contains the three columns shown at the top of this figure. The first column, ID, is defined as an identity column. The second column, ColorNumber, is defined with a default value of 0. And the third column, ColorName, is defined so it allows null values.

The first two statements illustrate how you assign a default value or a null value using a column list. To do that, you simply omit the column from the list. In the first statement, for example, the column list names only the ColorNumber column, so the ColorName column is assigned a null value. Similarly, the column list in the second statement names only the ColorName column, so the ColorNumber is assigned its default value.

The next three statements show how you assign a default or null value to a column without including a column list. As you can see, you do that by using the DEFAULT and NULL keywords. For example, the third statement specifies a value for the ColorName column, but uses the DEFAULT keyword for the ColorNumber column. Because of that, SQL Server will assign a value of zero to this column. The fourth statement assigns a value of 808 to the ColorNumber column, and it uses the NULL keyword to assign a null value to the ColorName column. The fifth statement uses both the DEFAULT and NULL keywords.

Finally, in the sixth statement, the DEFAULT keyword is coded in front of the VALUES clause. When you use the DEFAULT keyword this way, any column that has a default value will be assigned that value, and all other columns (except the identity column) will be assigned a null value. Because of that, you can use this technique only when every column in the table is defined as either an identity column, a column with a default value, or a column that allows null values.

The definition of the ColorSample table

Column name	Data Type	Length	Identity	Allow Nulls	Default Value
ID	Int	4	Yes	No	No
ColorNumber	Int	4	No	No	0
ColorName	VarChar	10	No	Yes	No

Six INSERT statements for the ColorSample table

```

INSERT INTO ColorSample (ColorNumber)
VALUES (606);

INSERT INTO ColorSample (ColorName)
VALUES ('Yellow');

INSERT INTO ColorSample
VALUES (DEFAULT, 'Orange');

INSERT INTO ColorSample
VALUES (808, NULL);

INSERT INTO ColorSample
VALUES (DEFAULT, NULL);

INSERT INTO ColorSample
DEFAULT VALUES;

```

The ColorSample table after the rows are inserted

	ID	ColorNumber	ColorName
1	1	606	NULL
2	2	0	Yellow
3	3	0	Orange
4	4	808	NULL
5	5	0	NULL
6	6	0	NULL

Description

- If a column is defined so it allows null values, you can use the NULL keyword in the list of values to insert a null value into that column.
- If a column is defined with a default value, you can use the DEFAULT keyword in the list of values to insert the default value for that column.
- If all of the columns in a table are defined as either identity columns, columns with default values, or columns that allow null values, you can code the DEFAULT keyword at the beginning of the VALUES clause and then omit the list of values.
- If you include a column list, you can omit columns with default values and null values. Then, the default value or null value is assigned automatically.

How to insert rows selected from another table

Instead of using the VALUES clause of the INSERT statement to specify the values for a single row, you can use a subquery to select the rows you want to insert from another table. Figure 7-4 shows you how to do that.

Both examples in this figure retrieve rows from the InvoiceCopy table and insert them into a table named InvoiceArchive. This table is defined with the same columns as the InvoiceCopy table. However, the InvoiceID column isn't defined as an identity column, and the PaymentTotal and CreditTotal columns aren't defined with default values. Because of that, you must include values for these columns.

The first example in this figure shows how you can use a subquery in an INSERT statement without coding a column list. In this example, the SELECT clause of the subquery is coded with an asterisk so that all the columns in the InvoiceCopy table will be retrieved. Then, after the search condition in the WHERE clause is applied, all the rows in the result set are inserted into the InvoiceArchive table.

The second example shows how you can use a column list in the INSERT clause when you use a subquery to retrieve rows. Just as when you use the VALUES clause, you can list the columns in any sequence. However, the columns must be listed in the same sequence in the SELECT clause of the subquery. In addition, you can omit columns that are defined with default values or that allow null values.

Notice that the subqueries in these statements aren't coded within parentheses as a subquery in a SELECT statement is. That's because they're not coded within a clause of the INSERT statement. Instead, they're coded in place of the VALUES clause.

Before you execute INSERT statements like the ones shown in this figure, you'll want to be sure that the rows and columns retrieved by the subquery are the ones you want to insert. To do that, you can execute the SELECT statement by itself. Then, when you're sure it retrieves the correct data, you can add the INSERT clause to insert the rows in the derived table into another table.

The syntax of the INSERT statement for inserting rows selected from another table

```
INSERT [INTO] table_name [(column_list)]
SELECT column_list
FROM table_source
[WHERE search_condition]
```

An INSERT statement that inserts paid invoices in the InvoiceCopy table into the InvoiceArchive table

```
INSERT INTO InvoiceArchive
SELECT *
FROM InvoiceCopy
WHERE InvoiceTotal - PaymentTotal - CreditTotal = 0;
(103 rows affected)
```

The same INSERT statement with a column list

```
INSERT INTO InvoiceArchive
(InvoiceID, VendorID, InvoiceNumber, InvoiceTotal, CreditTotal,
PaymentTotal, TermsID, InvoiceDate, InvoiceDueDate)
SELECT
InvoiceID, VendorID, InvoiceNumber, InvoiceTotal, CreditTotal,
PaymentTotal, TermsID, InvoiceDate, InvoiceDueDate
FROM InvoiceCopy
WHERE InvoiceTotal - PaymentTotal - CreditTotal = 0;
(103 rows affected)
```

Description

- To insert rows selected from one or more tables into another table, you can code a subquery in place of the VALUES clause. Then, the rows in the derived table that result from the subquery are inserted into the table.
- If you don't code a column list in the INSERT clause, the subquery must return values for all the columns in the table where the rows will be inserted, and the columns must be returned in the same order as they appear in that table. The exception is an identity column, which must be omitted.
- If you include a column list in the INSERT clause, the subquery must return values for those columns in the same order as they appear in the column list. You can omit columns with default values and columns that accept null values, and you must omit identity columns.

How to modify existing rows

To modify the data in one or more rows of a table, you use the UPDATE statement. Although most of the UPDATE statements you code will perform simple updates like the ones you'll see in the next figure, you can also code more complex UPDATE statements that include subqueries and joins. You'll learn how to use these features after you learn how to perform a basic update operation.

How to perform a basic update operation

Figure 7-5 presents the syntax of the UPDATE statement. As you can see in the examples, most UPDATE statements include just the UPDATE, SET, and WHERE clauses. The UPDATE clause names the table to be updated, the SET clause names the columns to be updated and the values to be assigned to those columns, and the WHERE clause specifies the condition a row must meet to be updated. Although the WHERE clause is optional, you'll almost always include it. If you don't, all of the rows in the table will be updated, which usually isn't what you want.

The first UPDATE statement in this figure modifies the values of two columns in the InvoiceCopy table: PaymentDate and PaymentTotal. Because the WHERE clause in this statement identifies a specific invoice number, only the columns in that invoice will be updated. Notice in this example that the values to be assigned to the two columns are coded as literals. You should realize, however, that you can assign any valid expression to a column as long as it results in a value that's compatible with the data type of the column. You can also use the NULL keyword to assign a null value to a column that allows nulls, and you can use the DEFAULT keyword to assign the default value to a column that's defined with one.

The second UPDATE statement modifies a single column in the InvoiceCopy table: TermsID. This time, however, the WHERE clause specifies that all the rows for vendor 95 should be updated. Because this vendor has six rows in the InvoiceCopy table, all six rows will be updated.

The third UPDATE statement illustrates how you can use an expression to assign a value to a column. In this case, the expression increases the value of the CreditTotal column by 100. Like the first UPDATE statement, this statement updates a single row.

Before you execute an UPDATE statement, you'll want to be sure that you've selected the correct rows. To do that, you can execute a SELECT statement with the same search condition. Then, if the SELECT statement returns the correct rows, you can change it to an UPDATE statement.

In addition to the UPDATE, SET, and WHERE clauses, an UPDATE statement can also include a FROM clause. This clause is an extension to the SQL standards, and you'll see how to use it in the next two figures.

The syntax of the UPDATE statement

```
UPDATE table_name  
SET column_name_1 = expression_1 [, column_name_2 = expression_2]...  
[FROM table_source [[AS] table_alias]  
[WHERE search_condition]
```

An UPDATE statement that assigns new values to two columns of a single row in the InvoiceCopy table

```
UPDATE InvoiceCopy  
SET PaymentDate = '2020-03-21',  
    PaymentTotal = 19351.18  
WHERE InvoiceNumber = '97/522';  
  
(1 row affected)
```

An UPDATE statement that assigns a new value to one column of all the invoices for a vendor

```
UPDATE InvoiceCopy  
SET TermsID = 1  
WHERE VendorID = 95;  
  
(6 rows affected)
```

An UPDATE statement that uses an arithmetic expression to assign a value to a column

```
UPDATE InvoiceCopy  
SET CreditTotal = CreditTotal + 100  
WHERE InvoiceNumber = '97/522';  
  
(1 row affected)
```

Description

- You use the UPDATE statement to modify one or more rows in the table named in the UPDATE clause.
- You name the columns to be modified and the value to be assigned to each column in the SET clause. You can specify the value for a column as a literal or an expression.
- You can provide additional criteria for the update operation in the FROM clause, which is a SQL Server extension. See figures 7-6 and 7-7 for more information.
- You can specify the conditions that must be met for a row to be updated in the WHERE clause.
- You can use the DEFAULT keyword to assign the default value to a column that has one, and you can use the NULL keyword to assign a null value to a column that allows nulls.
- You can't update an identity column.

Warning

- If you omit the WHERE clause, all the rows in the table will be updated.

How to use subqueries in an update operation

Figure 7-6 presents four more UPDATE statements that illustrate how you can use subqueries in an update operation. In the first statement, a subquery is used in the SET clause to retrieve the maximum invoice due date from the InvoiceCopy table. Then, that value is assigned to the InvoiceDueDate column for invoice number 97/522.

In the second statement, a subquery is used in the WHERE clause to identify the invoices to be updated. This subquery returns the VendorID value for the vendor in the VendorCopy table with the name “Pacific Bell.” Then, all the invoices with that VendorID value are updated.

The third UPDATE statement also uses a subquery in the WHERE clause. This subquery returns a list of the VendorID values for all the vendors in California, Arizona, and Nevada. Then, the IN operator is used to update all the invoices with VendorID values in that list. Note that although the subquery returns 80 vendors, many of these vendors don’t have invoices. As a result, the UPDATE statement only affects 51 invoices.

The fourth example in this figure shows how you can use a subquery in the FROM clause of an UPDATE statement to create a derived table. In this case, the subquery returns a table that contains the InvoiceID values of the ten invoices with the largest balances of \$100 or more. (Because this UPDATE statement will apply a credit of \$100 to these invoices, you don’t want to retrieve invoices with balances less than that amount.) Then, the WHERE clause specifies that only those invoices should be updated. You can also use a column from a derived table in an expression in the SET clause to update a column in the base table.

An UPDATE statement that assigns the maximum due date in the InvoiceCopy table to a specific invoice

```
UPDATE InvoiceCopy
SET CreditTotal = CreditTotal + 100,
    InvoiceDueDate = (SELECT MAX(InvoiceDueDate) FROM InvoiceCopy)
WHERE InvoiceNumber = '97/522';
(1 row affected)
```

An UPDATE statement that updates all the invoices for a vendor based on the vendor's name

```
UPDATE InvoiceCopy
SET TermsID = 1
WHERE VendorID =
    (SELECT VendorID
     FROM VendorCopy
     WHERE VendorName = 'Pacific Bell');
(6 rows affected)
```

An UPDATE statement that changes the terms of all invoices for vendors in three states

```
UPDATE InvoiceCopy
SET TermsID = 1
WHERE VendorID IN
    (SELECT VendorID
     FROM VendorCopy
     WHERE VendorState IN ('CA', 'AZ', 'NV'));
(51 rows affected)
```

An UPDATE statement that applies a \$100 credit to the 10 invoices with the largest balances

```
UPDATE InvoiceCopy
SET CreditTotal = CreditTotal + 100
FROM
    (SELECT TOP 10 InvoiceID
     FROM InvoiceCopy
     WHERE InvoiceTotal - PaymentTotal - CreditTotal >= 100
     ORDER BY InvoiceTotal - PaymentTotal - CreditTotal DESC) AS TopInvoices
WHERE InvoiceCopy.InvoiceID = TopInvoices.InvoiceID;
(5 rows affected)
```

Description

- You can code a subquery in the SET, FROM, or WHERE clause of an UPDATE statement.
- You can use a subquery in the SET clause to return the value that's assigned to a column.
- You can use a subquery in the FROM clause to identify the rows that are available for update. Then, you can refer to the derived table in the SET and WHERE clauses.
- You can code a subquery in the WHERE clause to provide one or more values used in the search condition.

How to use joins in an update operation

In addition to subqueries, you can use joins in the FROM clause of an UPDATE statement. Joins provide an easy way to base an update on data in a table other than the one that's being updated. The two examples in figure 7-7 illustrate how this works.

The first example in this figure updates the TermsID column in all the invoices in the InvoiceCopy table for the vendor named "Pacific Bell." This is the same update operation you saw in the second example in the previous figure. Instead of using a subquery to retrieve the VendorID value for the vendor, however, this UPDATE statement joins the InvoiceCopy and VendorCopy tables on the VendorID column in each table. Then, the search condition in the WHERE clause uses the VendorName column in the VendorCopy table to identify the invoices to be updated.

The second example in this figure shows how you can use the columns in a table that's joined with the table being updated to specify values in the SET clause. Here, the VendorCopy table is joined with a table named ContactUpdates. As you can see in the figure, this table includes VendorID, LastName, and FirstName columns. After the two tables are joined on the VendorID column, the SET clause uses the LastName and FirstName columns from the ContactUpdates table to update the VendorContactLName and VendorContactFName columns in the VendorCopy table.

An UPDATE statement that changes the terms of all the invoices for a vendor

```
UPDATE InvoiceCopy
SET TermsID = 1
FROM InvoiceCopy JOIN VendorCopy
    ON InvoiceCopy.VendorID = VendorCopy.VendorID
WHERE VendorName = 'Pacific Bell';
(6 rows affected)
```

An UPDATE statement that updates contact names in the VendorCopy table based on data in the ContactUpdates table

```
UPDATE VendorCopy
SET VendorContactLName = LastName,
    VendorContactFName = FirstName
FROM VendorCopy JOIN ContactUpdates
    ON VendorCopy.VendorID = ContactUpdates.VendorID;
(8 rows affected)
```

The ContactUpdates table

	VendorID	LastName	FirstName
1	5	Davison	Michelle
2	12	Mayteh	Kendall
3	17	Onandonga	Bruce
4	44	Antavius	Anthony
5	76	Bradlee	Danny
6	94	Suscipe	Reynaldo
7	101	O'Sullivan	Geraldine
8	123	Bucket	Charles

Description

- If you need to specify column values or search conditions that depend on data in a table other than the one named in the UPDATE clause, you can use a join in the FROM clause.
- You can use columns from the joined tables in the values you assign to columns in the SET clause or in the search condition of a WHERE clause.

How to delete existing rows

To delete one or more rows from a table, you use the DELETE statement. Just as you can with the UPDATE statement, you can use subqueries and joins in a DELETE statement to help identify the rows to be deleted. You'll learn how to use subqueries and joins after you learn how to perform a basic delete operation.

How to perform a basic delete operation

Figure 7-8 presents the syntax of the DELETE statement, along with three examples that illustrate some basic delete operations. As you can see, you specify the name of the table that contains the rows to be deleted in the DELETE clause. You can also code the FROM keyword in this clause, but this keyword is optional and is usually omitted.

To identify the rows to be deleted, you code a search condition in the WHERE clause. Although this clause is optional, you'll almost always include it. If you don't, all of the rows in the table are deleted. This is a common coding mistake, and it can be disastrous.

You can also include a FROM clause in the DELETE statement to join additional tables with the base table. Then, you can use the columns of the joined tables in the search condition of the WHERE clause. The FROM clause is an extension to the standard SQL syntax. You'll see how to use it in the next figure.

The first DELETE statement in this figure deletes a single row from the InvoiceCopy table. To do that, it specifies the InvoiceID value of the row to be deleted in the search condition of the WHERE clause. The second statement is similar, but it deletes all the invoices with a VendorID value of 37. In this case, three rows are deleted.

The third DELETE statement shows how you can use an expression in the search condition of the WHERE clause. In this case, the InvoiceTotal, PaymentTotal, and CreditTotal columns are used to calculate the balance due. Then, if the balance due is zero, the row is deleted. You might use a statement like this after inserting the paid invoices into another table as shown in figure 7-4.

Finally, the fourth DELETE statement shows how easy it is to delete all the rows from a table. Because the WHERE clause has been omitted from this statement, all the rows in the InvoiceCopy table will be deleted, which probably isn't what you want.

Because you can't restore rows once they've been deleted, you'll want to be sure that you've selected the correct rows. One way to do that is to issue a SELECT statement with the same search condition. Then, if the correct rows are retrieved, you can be sure that the DELETE statement will work as intended.

The syntax of the DELETE statement

```
DELETE [FROM] table_name  
[FROM table_source]  
[WHERE search_condition]
```

A DELETE statement that removes a single row from the InvoiceCopy table

```
DELETE InvoiceCopy  
WHERE InvoiceID = 115;  
(1 row affected)
```

A DELETE statement that removes all the invoices for a vendor

```
DELETE InvoiceCopy  
WHERE VendorID = 37;  
(3 rows affected)
```

A DELETE statement that removes all paid invoices

```
DELETE InvoiceCopy  
WHERE InvoiceTotal - PaymentTotal - CreditTotal = 0;  
(103 rows affected)
```

A DELETE statement that removes all the rows from the InvoiceCopy table

```
DELETE InvoiceCopy;  
(114 rows affected)
```

Description

- You can use the DELETE statement to delete one or more rows from the table you name in the DELETE clause.
- You specify the conditions that must be met for a row to be deleted in the WHERE clause.
- You can specify additional criteria for the delete operation in the FROM clause. See figure 7-9 for more information.

Warning

- If you omit the WHERE clause from a DELETE statement, all the rows in the table will be deleted.

How to use subqueries and joins in a delete operation

The examples in figure 7-9 illustrate how you can use subqueries and joins in a DELETE statement. Because you've seen code like this in other statements, you shouldn't have any trouble understanding these examples.

The first two examples delete all the invoices from the InvoiceCopy table for the vendor named "Blue Cross." To accomplish that, the first example uses a subquery in the WHERE clause to retrieve the VendorID value from the VendorCopy table for this vendor. By contrast, the second example joins the InvoiceCopy and VendorCopy tables. Then, the WHERE clause uses the VendorName column in the VendorCopy table to identify the rows to be deleted.

The third DELETE statement deletes all vendors that don't have invoices. To do that, it uses a subquery to return a list of the VendorID values in the InvoiceCopy table. Then, it deletes all vendors that aren't in that list.

The fourth DELETE statement shows how you can use the FROM clause to join the base table named in the DELETE clause with a derived table. Here, the subquery creates a derived table based on the InvoiceCopy table. This subquery groups the invoices in this table by vendor and calculates the total invoice amount for each vendor. Then, after the derived table is joined with the VendorCopy table, the results are filtered by the total invoice amount. Because of that, only those vendors that have invoices totaling \$100 or less will be deleted from the VendorCopy table.

A DELETE statement that deletes all invoices for a vendor based on the vendor's name

```
DELETE InvoiceCopy
WHERE VendorID =
  (SELECT VendorID
   FROM VendorCopy
   WHERE VendorName = 'Blue Cross');

(3 rows affected)
```

The same DELETE statement using a join

```
DELETE InvoiceCopy
FROM InvoiceCopy JOIN VendorCopy
  ON InvoiceCopy.VendorID = VendorCopy.VendorID
WHERE VendorName = 'Blue Cross';

(3 rows affected)
```

A DELETE statement that deletes vendors that don't have invoices

```
DELETE VendorCopy
WHERE VendorID NOT IN
  (SELECT DISTINCT VendorID FROM InvoiceCopy);

(88 rows affected)
```

A DELETE statement that deletes vendors whose invoices total \$100 or less

```
DELETE VendorCopy
FROM VendorCopy JOIN
  (SELECT VendorID, SUM(InvoiceTotal) AS TotalOfInvoices
   FROM InvoiceCopy
   GROUP BY VendorID) AS InvoiceSum
  ON VendorCopy.VendorID = InvoiceSum.VendorID
WHERE TotalOfInvoices <= 100;

(6 rows affected)
```

Description

- You can use subqueries and joins in the FROM clause of a DELETE statement to base the delete operation on the data in tables other than the one named in the DELETE clause.
- You can use any of the columns returned by a subquery or a join in the WHERE clause of the DELETE statement.
- You can also use subqueries in the WHERE clause to provide one or more values used in the search condition.

Note

- The FROM clause is a SQL Server extension.

How to merge rows

SQL Server 2008 introduced a MERGE statement that allows you to merge multiple rows from one table into another table. Since this typically involves updating existing rows and inserting new rows, the MERGE statement is sometimes referred to as the *upsert* statement.

How to perform a basic merge operation

Figure 7-10 presents the syntax of the MERGE statement, along with an example that shows how it works. To start, you code the MERGE keyword, followed by the optional INTO keyword, followed by the target table. In this example, the target table is the InvoiceArchive table, and it has an alias of *ia*.

After the MERGE clause, you code the USING keyword followed by the source table. In this example, the source table is the InvoiceCopy table, and it has an alias of *ic*.

After the USING clause, you code an ON clause that specifies the condition that's used to join the two tables. In this figure, both tables use the InvoiceID column as the primary key, and the ON clause joins these tables on this column. However, if necessary, you can code more complex join conditions.

After the USING clause, you can code one or more WHEN clauses that control when a row is inserted, updated, or deleted. In this figure, for instance, the first WHEN clause checks if (a) the InvoiceID values are matched, (b) the PaymentDate column in the InvoiceCopy table is not a NULL value, and (c) the PaymentTotal column in the InvoiceCopy table is greater than the PaymentTotal column in the InvoiceArchive table. Here, the second and third conditions prevent every row with a matching InvoiceID from being updated. If all three conditions are true, this WHEN clause updates the columns in the InvoiceArchive table with the corresponding values in the InvoiceCopy table. To do that, this code uses the table aliases to qualify each column name.

The second WHEN clause checks if the InvoiceID values are unmatched. If so, it inserts the row into the InvoiceArchive table. To do that, this code uses the table alias for the InvoiceCopy table to qualify each column name.

Finally, to signal the end of a MERGE statement, you must code a semi-colon (;). Otherwise, you will get an error when you attempt to run it.

How to code more complex merge operations

The example in this figure shows a typical use of the MERGE statement. However, you can also use the MERGE statement to delete rows in the target table that aren't matched in the source table. To do that, you can add a WHEN clause like the third one in figure 7-10. In addition, the MERGE statement supports other more complex functionality such as a TOP clause. For more information, you can look up the MERGE statement in the documentation for SQL Server.

The syntax of the MERGE statement

```
MERGE [INTO] table_target
USING table_source
ON join_condition
[WHEN MATCHED [AND search_condition]...
    THEN dml_statement ]
[WHEN NOT MATCHED [BY TARGET ][AND search_condition]...
    THEN dml_statement ]
[WHEN NOT MATCHED BY SOURCE [AND search_condition]...
    THEN dml_statement ]
;
;
```

A MERGE statement that inserts and updates rows

```
MERGE InvoiceArchive AS ia
USING InvoiceCopy AS ic
ON ic.InvoiceID = ia.InvoiceID
WHEN MATCHED AND
    ic.PaymentDate IS NOT NULL AND
    ic.PaymentTotal > ia.PaymentTotal
THEN
    UPDATE SET
        ia.PaymentTotal = ic.PaymentTotal,
        ia.CreditTotal = ic.CreditTotal,
        ia.PaymentDate = ic.PaymentDate
WHEN NOT MATCHED THEN
    INSERT (InvoiceID, VendorID, InvoiceNumber,
        InvoiceTotal, PaymentTotal, CreditTotal,
        TermsID, InvoiceDate, InvoiceDueDate)
    VALUES (ic.InvoiceID, ic.VendorID, ic.InvoiceNumber,
        ic.InvoiceTotal, ic.PaymentTotal, ic.CreditTotal,
        ic.TermsID, ic.InvoiceDate, ic.InvoiceDueDate)
;
;
```

A WHEN clause that deletes rows that aren't matched by the source

```
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
```

Description

- The MERGE statement merges multiple rows from one table into another table. Since this often involves updating existing rows and inserting new rows, the MERGE statement is sometimes referred to as the *upsert* statement.
- When coding search conditions, you can use the AND keyword to create compound search conditions.
- You can code one or more WHEN clauses that control when and how a row is inserted, updated, or deleted.
- Within a WHEN clause, you can code a simplified INSERT, UPDATE, or DELETE statement that doesn't include a table name or a WHERE clause.
- To signal the end of a MERGE statement, you must code a semicolon (;).

Perspective

In this chapter, you learned how to use the INSERT, UPDATE, and DELETE statements to modify the data in a database. Now, if you want to practice using these statements, please use one of the two options presented at the start of this chapter so you won't corrupt a live database or a database shared by others.

In chapters 10 and 11, you'll learn more about the table definitions that can affect the way these statements work. If, for example, you delete a row in a Vendors table that has related rows in an Invoices table, SQL Server may delete all of the related rows in the Invoices table. But that depends on how the relationship is defined. So, to complete your understanding of the INSERT, UPDATE, and DELETE statements, you need to read chapters 10 and 11.

Finally, you should know that you can use the TOP clause with the INSERT, UPDATE, and DELETE statements to limit the number of rows that are inserted, updated, or deleted. This works similarly to using the TOP clause with a SELECT statement as described in chapter 3. As a result, if you read chapter 3, you shouldn't have much trouble using the TOP clause with the INSERT, UPDATE, and DELETE statements. For coding details, you can search for "TOP (Transact-SQL)" in the documentation for SQL Server.

Exercises

1. Write SELECT INTO statements to create two test tables named VendorCopy and InvoiceCopy that are complete copies of the Vendors and Invoices tables. If VendorCopy and InvoiceCopy already exist, first code two DROP TABLE statements to delete them.
2. Write an INSERT statement that adds a row to the InvoiceCopy table with the following values:

VendorID: 32
InvoiceTotal: \$434.58
TermsID: 2
InvoiceNumber: AX-014-027
PaymentTotal: \$0.00
InvoiceDueDate: 05/8/2020
InvoiceDate: 4/21/2020
CreditTotal: \$0.00
PaymentDate: null
3. Write an INSERT statement that adds a row to the VendorCopy table for each non-California vendor in the Vendors table. (This will result in duplicate vendors in the VendorCopy table.)
4. Write an UPDATE statement that modifies the VendorCopy table. Change the default account number to 403 for each vendor that has a default account number of 400.

5. Write an UPDATE statement that modifies the InvoiceCopy table. Change the PaymentDate to today's date and the PaymentTotal to the balance due for each invoice with a balance due. Set today's date with a literal date string, or use the GETDATE() function.
6. Write an UPDATE statement that modifies the InvoiceCopy table. Change TermsID to 2 for each invoice that's from a vendor with a DefaultTermsID of 2. Use a subquery.
7. Solve exercise 6 using a join rather than a subquery.
8. Write a DELETE statement that deletes all vendors in the state of Minnesota from the VendorCopy table.
9. Write a DELETE statement for the VendorCopy table. Delete the vendors that are located in states from which no vendor has ever sent an invoice.

Hint: Use a subquery coded with “SELECT DISTINCT VendorState” introduced with the NOT IN operator.

8

How to work with data types

So far, you have been using SQL statements to work with the three most common types of data: strings, numbers, and dates. Now, this chapter takes a more in-depth look at the data types that are available with SQL Server and shows some basic skills for working with them. When you complete this chapter, you'll have a thorough understanding of the data types, and you'll know how to use some functions to convert one data type to another.

A review of the SQL data types.....	240
Data type overview	240
The numeric data types	242
The string data types	244
The date/time data types	246
The large value data types	248
How to convert data	250
How data conversion works	250
How to convert data using the CAST function.....	252
How to convert data using the CONVERT function.....	254
How to use the TRY_CONVERT function	256
How to use other data conversion functions.....	258
Perspective	260

A review of the SQL data types

A column's *data type* specifies the kind of information the column is intended to store. In addition, a column's data type determines the operations that can be performed on the column.

Data type overview

The SQL Server data types can be divided into the four categories shown in the first table in figure 8-1. The *string data types* are intended for storing a string of one or more characters, which can include letters, numbers, symbols, or special characters. The terms *character*, *string*, and *text* are used interchangeably to describe this type of data.

The *numeric data types* are intended for storing numbers that can be used for mathematical calculations. As you'll see in the next topic, SQL Server can store numbers in a variety of formats.

The *temporal data types* are used to store dates and times. These data types are typically referred to as *date/time*, or *date, data types*.

Historically, most databases have stored string, numeric, and temporal data. That's why this book focuses on working with these data types. However, it's becoming more common to store other types of data such as images, sound, and video in databases. That's why SQL Server 2005 introduced new data types for working with these types of data. For more information about working with large character and binary values, see figure 8-5.

SQL Server 2005 also introduced a rowversion data type. This data type is meant to replace the timestamp data type, which has been deprecated. The rowversion data type is used to store unique 8-byte binary values that are generated by SQL Server and that are typically used to identify various versions of the rows in a table. The value of a column with the rowversion data type is updated any time a row is inserted or updated in the table.

Finally, SQL Server 2005 introduced an xml data type that provides new functionality for storing XML in a database. To learn more about working with the xml data type, see chapter 18.

Beyond that, SQL Server 2008 introduced several more data types. It introduced several new date/time data types that are described later in this chapter. In addition, it introduced a geometry data type for storing geometric data such as points, lines, and polygons. It introduced a geography data type that works similarly to the geometry type, except that it uses longitude and latitude to specify points on the earth's surface. And, it introduced the hierarchyid data type for storing hierarchical data such as organization charts.

Most of the SQL Server data types correspond to the ANSI-standard data types. These data types are listed in the second table in this figure. Here, the second column lists the SQL Server data type names, and the first column lists the synonyms SQL Server provides for the ANSI-standard data types. Although you can use these synonyms instead of the SQL Server data types, you're not likely to do that. If you do, SQL Server simply maps the synonyms to the corresponding SQL Server data types.

The four data type categories

Category	Description
String	Strings of character data
Numeric	Integers, floating point numbers, currency, and other numeric data
Temporal (date/time)	Dates, times, or both
Other	Large character and binary values, XML, geometric data, geographic data, hierarchical data

ANSI-standard data types and SQL Server equivalents

Synonym for ANSI-standard data type	SQL Server data type used
binary varying	varbinary
char varying	varchar
character varying	
character	char
dec	decimal
double precision	float
float	real or float
integer	int
national char	nchar
national character	
national char varying	nvarchar
national character varying	
national text	ntext
timestamp	rowversion

Description

- SQL Server defines dozens of *data types* that are divided into the four categories shown above.
- The *temporal data types* are typically referred to as *date/time data types*, or simply *date data types*.
- SQL Server supports most, but not all, of the ANSI-standard data types.
- SQL Server provides a synonym for each of the supported ANSI-standard data types. Although you can use these synonyms, I recommend you use the SQL Server data types instead.
- When you use the synonym for an ANSI data type, it's mapped to the appropriate SQL Server data type indicated in the table above.

The numeric data types

Figure 8-2 presents the numeric data types supported by SQL Server. As you can see, these can be divided into three groups: integer, decimal, and real.

Integer data types store whole numbers, which are numbers with no digits to the right of the decimal point. The five integer data types differ in the amount of storage they use and the range of values they can store. Notice that the bigint, int, and smallint data types can store positive or negative numbers. By contrast, the tinyint and bit data types can store only positive numbers or zero.

To store numbers with digits to the right of the decimal point, you use the *decimal data types*. These data types have a fixed decimal point, which means that the number of digits to the right of the decimal point doesn't vary. The number of digits a value has to the right of the decimal point is called its *scale*, and the total number of digits is called its *precision*. Notice that the money and smallmoney data types have a fixed precision and scale. These data types are intended for storing units of currency. By contrast, you can customize the precision and scale of the decimal and numeric data types so they're right for the data to be stored. Although the decimal and numeric data types are synonymous, decimal is more commonly used.

In contrast to the *fixed-point numbers* stored by the decimal data types, the *real data types* are used to store *floating-point numbers*. These data types provide for very large and very small numbers, but with a limited number of *significant digits*. The real data type can be used to store a *single-precision number*, which provides for numbers with up to 7 significant digits. And the float data type can be used to store a *double-precision number*, which provides for numbers with up to 15 significant digits. Because the real data type is equivalent to float(24), the float data type is typically used for floating-point numbers.

To express the value of a floating-point number, you can use *scientific notation*. To use this notation, you type the letter E followed by a power of 10. For instance, 3.65E+9 is equal to 3.65×10^9 , or 3,650,000,000. If you have a mathematical background, of course, you're already familiar with this notation.

Because the precision of all the integer and decimal data types is exact, these data types are considered *exact numeric data types*. By contrast, the real data types are considered *approximate numeric data types* because they may not represent a value exactly. That can happen, for example, when a number is rounded to the appropriate number of significant digits. For business applications, you will most likely use only the exact numeric types, as there's seldom the need to work with the very large and very small numbers that the real data types are designed for.

The integer data types

Type	Bytes	Description
bigint	8	Large integers from -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807.
int	4	Integers from -2,147,483,648 through 2,147,483,647.
smallint	2	Small integers from -32,768 through 32,767.
tinyint	1	Very small positive integers from 0 through 255.
bit	1	Integers with a value of 1 or 0.

The decimal data types

Type	Bytes	Description
decimal[(p[,s])]	5-17	Decimal numbers with fixed precision (p) and scale (s) from $-10^{38}+1$ through $10^{38}-1$. The precision can be any number between 1 and 38; the default is 18. The scale can be any number between 0 and the precision; the default is 0.
numeric[(p[,s])]	5-17	Synonymous with decimal.
money	8	Monetary values with four decimal places from -922,337,203,685,477.5808 through 922,337,203,685,477.5807. Synonymous with decimal(19,4).
smallmoney	4	Monetary values with four decimal places from -214,748.3648 through 214,748.3647. Synonymous with decimal(10,4).

The real data types

Type	Bytes	Description
float[(n)]	4 or 8	Double-precision floating-point numbers from -1.79×10^{308} through 1.79×10^{308} . n represents the number of bits used to store the decimal portion of the number (the mantissa): n=24 is single-precision; n=53 is double-precision. The default is 53.
real	4	Single-precision floating point numbers from -3.4×10^{38} through 3.4×10^{38} . Synonymous with float(24).

Description

- The *integer data types* are used to store whole numbers, which are numbers without any digits to the right of the decimal point.
- The *decimal data types* are used to store decimal values, which can include digits to the right of the decimal point. The *precision* of a decimal value indicates the total number of digits that can be stored, and the *scale* indicates the number of digits that can be stored to the right of the decimal point.
- The integer and decimal data types are considered *exact numeric data types* because their precision is exact.
- The *real data types* are used to store *floating-point numbers*, which have a limited number of *significant digits*. These data types are considered *approximate numeric data types* because they may not represent a value exactly.

Figure 8-2 The numeric data types

The string data types

Figure 8-3 presents the four most common string data types supported by SQL Server. The number of bytes that are used to store each character for these data types depends on the collation that's used. You'll learn more about collations in chapter 11. For now, we'll assume you're using the default collation.

With the default collation for SQL Server, the char and varchar data types use one byte per character. This allows them to store most characters used by Western European languages. On the other hand, the nchar and nvarchar data types use two bytes per character. This allows them to store most *Unicode characters*, which include most characters used by the world's writing systems. In the ANSI standards, two-byte characters are known as *national characters*. That's why the names of these data types start with the letter *n*, and that's why you prefix string literals with an *N* when working with these data types.

You use the char and nchar data types to store *fixed-length strings*. These data types always use the same number of bytes regardless of the length of the string. To make that possible, SQL Server appends spaces to the string whenever necessary. These data types are typically used to define columns that have a fixed number of characters. For example, the VendorState column in the Vendors table is defined with the char(2) type because it always contains two characters.

You use the varchar and nvarchar data types to store *variable-length strings*. These data types only use the number of bytes needed to store the string, plus two bytes to store the length of the string. They're typically used to define columns whose lengths vary from one row to the next. In general, variable-length strings use less storage than fixed-length strings. As a result, you should only use fixed-length strings when the length of the string is always going to be the same.

Although you typically store numeric values using numeric data types, the string data types may be a better choice for some numeric values. For example, you typically store zip codes, telephone numbers, and social security numbers in string columns even if they contain only numbers. That's because their values aren't used in arithmetic operations. In addition, if you stored these numbers in numeric columns, leading zeros would be stripped, which isn't what you want.

As I said, the char and varchar types store each character with a single byte when you use the default collation. Because a byte consists of eight *bits* and because eight bits can be combined in 256 different ways, each byte can represent one of 256 different characters. These characters are assigned numeric codes from 0 to 255. Most systems use the same codes for the first 128 characters. These are the codes defined by the *ASCII (American Standard Code for Information Interchange)* system. The other codes, however, may be unique on your system.

When you use the nchar and nvarchar types to store Unicode characters with the default collation, each character is stored in two bytes (16 bits). This provides for 63,536 different characters, which allows for most characters used by the world's writing systems. Because of that, you should use the nchar and nvarchar data types if your database needs to support a multi-language environment. However, if your database primarily needs to support Western European languages, you should use the char and varchar data types to keep storage requirements to a minimum.

The string data types used to store standard characters

Type	Bytes	Description
char(n)	n	Fixed-length string of characters where n is the number of bytes ranging from 1 to 8000.
varchar(n)		Variable-length string of characters where n is the maximum number of bytes ranging from 1 to 8000.

The string data types used to store Unicode characters

Type	Bytes	Description
nchar(n)	2xn	Fixed-length string of Unicode characters where n is the number of byte-pairs ranging from 1 to 4000.
nvarchar(n)		Variable-length string of Unicode characters where n is the maximum number of byte-pairs ranging from 1 to 4000.

The string data types with the default collation for most systems

Type	Original value	Value stored	Bytes used
CHAR(2)	'CA'	'CA'	2
CHAR(10)	'CA'	'CA' '	10
VARCHAR(20)	'CA'	'CA' '	4 (2 + 2)
VARCHAR(20)	'New York'	'New York'	10 (8 + 2)
NCHAR(2)	N'CA'	N'CA'	4
NCHAR(10)	N'CA'	N'CA' '	20
NVARCHAR(20)	N'CA'	N'CA' '	6 (4 + 2)
NVARCHAR(20)	N'New York'	N'New York'	18 (16 + 2)

Description

- The number of bytes that are used to store characters depends on the collation that's used. For more information about collations, see chapter 11.
- With the default collation, the char and varchar types use one byte per character. This allows these types to store the characters used by most Western European languages.
- With the default collation, the nchar and nvarchar types use one *byte-pair*, or two bytes, per character. This allows these types to store most *Unicode characters*, which includes most characters used by the world's writing systems.
- The char and nchar types store *fixed-length strings* that use the same amount of storage for each value. To do that, they append spaces to the end of the string whenever necessary.
- The varchar and nvarchar types store *variable-length strings* that use a varying amount of storage for each value. To do that, they store the characters for the string plus two bytes that store the length of the string.
- With the default collation, the number specified within the parentheses for a data type matches the number of characters that can be stored in the data type. However, that's not true for collations that allow these data types to store the full range of Unicode characters.

Figure 8-3 The string data types

The date/time data types

Figure 8-4 starts by presenting the two date/time data types supported prior to SQL Server 2008. These data types differ by the amount of storage they use and the range of values they can store. In particular, the datetime type supports a wider range of dates and more precise time values than the smalldatetime type. However, it also requires more bytes.

After the older data types, this figure presents the four date/time data types that were introduced with SQL Server 2008. These data types offer several advantages over the older date/time types. As a result, for new development, you will probably want to use the data types that were introduced with SQL Server 2008 instead of the older date/time data types. Of course, for existing databases, you can continue to use the older date/time types.

If you want to store date values without storing a time component, you can use the date data type. This data type is appropriate for many date/time values where the time component isn't necessary such as birthdays. In addition, compared to the datetime type, the date type reduces storage requirements, allows for a wider range of dates, and makes it easier to search a range of dates.

Conversely, if you want to store a time value without storing a date component, you can use the time data type. When you work with this type, you can specify the precision for the fractional seconds from 0 to 7 digits. Then, the amount of disk space that's required varies from 3 to 5 bytes depending on the precision that's specified. For example, the time(1) type can only store 1 digit of fractional seconds and requires 3 bytes. At the other end of the spectrum, the time(7) type can store 7 digits of fractional seconds and requires 5 bytes.

The datetime2 data type combines the date and time data types into a single type that stores both a date and a time component. This allows you to use the techniques for specifying the precision for fractional seconds with the datetime2 data type. For example, compared to the datetime type, the datetime2(3) type stores a wider range of date values with more precise time values and uses less storage (only 6 bytes).

The datetimeoffset data type works like the datetime2 data type. However, this data type also stores a time zone offset that specifies the number of hours that the value is ahead or behind *Greenwich Mean Time (GMT)*, which is also known as *Universal Time Coordinate (UTC)*. Although this can make it easier to work with time zones, it requires another 2 bytes of storage.

When you work with date/time values, you need to know how to code date and time literals. This figure presents some of the most common formats for dates and times. All SQL Server systems recognize dates in the yyyy-mm-dd format, which is why I've used this format in most of the examples in this book. Most systems recognize the other date and time formats shown here as well. Since the supported formats depend on system settings, however, you may need to check and see which ones are acceptable on your system.

You also need to be aware of the two-digit year cutoff that's defined on your system when you work with date literals. When you code a two-digit year, the two-digit year cutoff determines whether the year is interpreted as a year

The date/time data types prior to SQL Server 2008

Type	Bytes	Description
<code>datetime</code>	8	Dates and times from January 1, 1753 through December 31, 9999, with an accuracy of 3.33 milliseconds.
<code>smalldatetime</code>	4	Dates and times from January 1, 1900 through June 6, 2079, with an accuracy of one minute.

The date/time data types introduced with SQL Server 2008

Type	Bytes	Description
<code>date</code>	3	Dates only (no time part) from January 1, 0001 through December 31, 9999.
<code>time(n)</code>	3-5	Times only (no date part) from 00:00:00.0000000 through 23:59:59.9999999, with an accuracy of .0000001 seconds. <i>n</i> is the number of digits from 0 to 7 that are used for fractional second precision.
<code>datetime2(n)</code>	6-8	Dates from January 1, 0001 through December 31, 9999 with time values from 00:00:00.0000000 through 23:59:59.9999999.
<code>datetimeoffset(n)</code>	8-10	An extension of the datetime2 type that also includes a time zone offset from -14 to +14.

Common date and time formats

Date format	Example
<code>yyyy-mm-dd</code>	<code>2020-04-30</code>
<code>mm/dd/yyyy</code>	<code>4/30/2020</code>
<code>mm-dd-yy</code>	<code>4-30-20</code>
<code>Month dd, yyyy</code>	<code>April 30, 2020</code>
<code>Mon dd, yy</code>	<code>Apr 30, 20</code>
<code>dd Mon yy</code>	<code>30 Apr 20</code>

Time format	Example
<code>hh:mi</code>	<code>16:20</code>
<code>hh:mi am/pm</code>	<code>4:20 pm</code>
<code>hh:mi:ss</code>	<code>4:20:36</code>
<code>hh:mi:ss:mmm</code>	<code>4:20:36:12</code>
<code>hh:mi:ss.nnnnnnnn</code>	<code>4:20:36.1234567</code>

Description

- You can specify a date/time value by coding a date/time literal. To code a date/time literal, enclose the date/time value in single quotes.
- If you don't specify a time when storing a date/time value, the time defaults to 12:00 a.m. If you don't specify a date when storing a date/time value, the date defaults to January 1, 1900.
- By default, the two-digit year cutoff is 50, which means that 00 to 49 are interpreted as 2000 to 2049 and 50 through 99 are interpreted as 1950 through 1999.
- You can specify a time using either a 12-hour or a 24-hour clock. For a 12-hour clock, am is the default.

Figure 8-4 The date/time data types

in the 20th or the 21st century. By default, SQL Server interprets the years 00 through 49 as 2000 through 2049, and it interprets the years 50 through 99 as 1950 through 1999. Because the two-digit year cutoff can be modified, however, you'll want to find out what it is on your system before you use date literals with two-digit years. Then, if you code a date literal outside the range, you'll have to use a four-digit year. Of course, you can always code all of your date literals with four-digit years, just to be sure.

The large value data types

Figure 8-5 presents the data types introduced in SQL Server 2005 that make it easier to work with large values, such as image and sound files. To start, SQL Server provides a max specifier that can be used with the varchar and nvarchar data types described in figure 8-3. This allows you to store up to 2 gigabytes of character data in a column. In addition, the max specifier can be used with the varbinary data type. This allows you to store up to 2 gigabytes of binary data in a column. Since these data types allow you to store large values, they're known as the *large value data types*.

Prior to SQL Server 2005, you could use the text, ntext, and image data types to store this type of data. However, these data types have been deprecated and will be removed in a future version of Microsoft SQL Server. As a result, you should avoid using these data types for any new development. Instead, you should use the corresponding large value data type.

One advantage of the large value data types is that they work like their smaller counterparts. As a result, once you learn how to use the smaller counterparts, you can use the same skills to work with the large value data types. For example, once you understand how to work with the varchar data type, you can use the same skills for working with the varchar(max) data type. However, since the large value data types may store up to 2 gigabytes of data in a column, you may not want to read or write the entire value at once. In that case, you can use the SUBSTRING function that you'll learn about in the next chapter to read the data in chunks, and you can use the .WRITE clause of the UPDATE statement to update the value in chunks. For more information about using the .WRITE clause of the UPDATE statement, you can look up the UPDATE statement in the documentation for SQL Server.

A second advantage of the large value data types is that they don't have as many restrictions as the old text, ntext, and image data types. For example, they can be used as variables in batches and scripts.

In recent years, it has become increasingly common to store large binary values such as images, sounds, and video within a database. That's why chapter 19 shows how to use the varbinary(max) type to store large binary values within a database. Once you learn how to work with this type, you can apply similar skills to the varchar(max) and nvarchar(max) types if you need to do that.

The large value data types for SQL Server 2005 and later

Type	Description
varchar(max)	Works the same as the varchar type described in figure 8-3, but the max specifier allows this data type to store up to 2,147,483,648 bytes of data.
nvarchar(max)	Works the same as the nvarchar type described in figure 8-3, but the max specifier allows this data type to store up to 2,147,483,648 bytes of data.
varbinary(max)	Stores variable-length binary data up to a maximum of 2,147,483,648 bytes. The number of bytes used to store the data depends on the actual length of the data.

How the large value data types map to the old large object types

SQL Server 2005 and later	Prior to 2005
varchar(max)	text
nvarchar(max)	ntext
varbinary(max)	image

Description

- The max specifier can be used with the varchar, nvarchar, and varbinary data types to increase the storage capacity of the column so it can store up to 2 gigabytes of data.
- The varchar(max), nvarchar(max), and varbinary(max) data types are known as the *large value data types*, and these data types can be used to store images and other types of large character or binary data. For more information about working with the varbinary(max) data type, see chapter 19.
- The text, ntext, and image data types that were used prior to SQL Server 2005 have been deprecated and will be removed in a future version of Microsoft SQL Server. As a result, you should avoid using these data types for any new development.
- The large value data types work like their smaller counterparts. As a result, once you learn how to use the smaller counterparts, you can use the same skills to work with the large value data types. In addition, the large value data types don't have as many restrictions as the old text, ntext, and image data types.

How to convert data

As you work with the various data types, you'll find that you frequently need to convert a value with one data type to another data type. Although SQL Server does many conversions automatically, it doesn't always do them the way you want. Because of that, you need to be aware of how data conversion works, and you need to know when and how to specify the type of conversion you want to do.

How data conversion works

Before SQL Server can operate on two values, it must convert those values to the same data type. To do that, it converts the value that has the data type with the lowest precedence to the data type of the other value. Figure 8-6 presents the order of precedence for some common SQL Server data types.

To illustrate how this works, consider the three expressions shown in this figure. The first expression multiplies the `InvoiceTotal` column, which is defined with the `money` data type, by a decimal value. Because the `decimal` data type has a higher precedence than the `money` data type, the value in the `InvoiceTotal` column is converted to a `decimal` before the multiplication is performed. Then, the result of the operation is also a `decimal` value. Similarly, the integer literal in the second expression is converted to the `money` data type before it's subtracted from the `PaymentTotal` column, and the result of the operation is a `money` value.

The third example shows that data conversion is also used when a value is assigned to a column. In this case, a date literal is assigned to the `PaymentDate` column. Because this column is defined with the `date` data type, the literal must be converted to this data type before it can be assigned to the column.

When SQL Server performs a conversion automatically, it's called an *implicit conversion*. However, not all conversions can be done implicitly. Some of the conversions that can't be done implicitly are listed in this figure. These conversions must be done explicitly. To perform an *explicit conversion*, you use the `CAST` and `CONVERT` functions you'll learn about in the next two topics.

Before I go on, you should realize that you won't usually code expressions with literal values like the ones shown in this figure. Instead, you'll use column names that contain the appropriate values. I used literal values here so it's clear what data types are being evaluated.

Order of precedence for common SQL Server data types

Precedence	Category	Data type
Highest	Date/time	datetime2 date time
	Numeric	float real decimal money smallmoney int smallint tinyint bit
	String	nvarchar nchar varchar char
Lowest		

Conversions that can't be done implicitly

From data type	To data type
char, varchar, nchar, nvarchar	money, smallmoney
money, smallmoney	char, varchar, nchar, nvarchar

Expressions that use implicit conversion

```

InvoiceTotal * .0775      -- InvoiceTotal (money) converted to decimal
PaymentTotal - 100        -- Numeric literal converted to money
PaymentDate = '2020-04-05' -- Date literal converted to date value

```

Description

- If you assign a value with one data type to a column with another data type, SQL Server converts the value to the data type of the column using *implicit conversion*. Not all data types can be converted implicitly to all other data types.
- SQL Server also uses implicit conversion when it evaluates an expression that involves values with different data types. In that case, it converts the value whose data type has lower precedence to the data type that has higher precedence. The result of the expression is returned in this same data type.
- Each combination of precision and scale for the decimal and numeric values is considered a different data type, with higher precision and scale taking precedence.
- If you want to perform a conversion that can't be done implicitly or you want to convert a data type with higher precedence to a data type with lower precedence, you can use the CAST or CONVERT function to perform an *explicit conversion*.
- Not all data types can be converted to another data type. For example, the datetime2, date, and time data types can't be converted to any of the numeric data types.

Figure 8-6 How data conversion works

How to convert data using the CAST function

Figure 8-7 presents the syntax of the CAST function. This function lets you convert, or *cast*, an expression to the data type you specify.

The SELECT statement in this figure illustrates how this works. Here, the third column in the result set shows that when the date values that are stored in the InvoiceDate column are cast as varchar values, they're displayed the same as they are without casting. If you want to display them with a different format, then, you can use the CONVERT function as shown in the next figure.

The fourth column in the result set shows what happens when the money values in the InvoiceTotal column are cast as integer values. Before the digits to the right of the decimal point are dropped, the numbers are rounded to the nearest whole number. Finally, the last column in the result set shows the values from the InvoiceTotal column cast as varchar values. In this case, the result looks the same even though the data has been converted from the money type to the varchar type.

This figure also illustrates a problem that can occur when you perform integer division without explicit conversion. In the first example, the number 50 is divided by the number 100 giving a result of 0. This happens because the result of the division of two integers must be an integer. For this operation to return an accurate result, then, you must explicitly convert one of the numbers to a decimal. Then, because the data type of the other value will be lower in the order of precedence, that value will be converted to a decimal value as well and the result will be a decimal. This is illustrated in the second example, where the value 100 is converted to a decimal. As you can see, the result is .5, which is what you want.

The syntax of the CAST function

```
CAST(expression AS data_type)
```

A SELECT statement that uses the CAST function

```
SELECT InvoiceDate, InvoiceTotal,  
       CAST(InvoiceDate AS varchar) AS varcharDate,  
       CAST(InvoiceTotal AS int) AS integerTotal,  
       CAST(InvoiceTotal AS varchar) AS varcharTotal  
FROM Invoices;
```

	InvoiceDate	InvoiceTotal	varcharDate	integerTotal	varcharTotal
1	2019-10-08	3813.33	2019-10-08	3813	3813.33
2	2019-10-10	40.20	2019-10-10	40	40.20
3	2019-10-13	138.75	2019-10-13	139	138.75
4	2019-10-16	144.70	2019-10-16	145	144.70

How to convert data when performing integer division

Operation	Result
50/100	0
50/CAST(100 AS decimal(3))	.500000

Description

- You can use the CAST function to explicitly convert, or *cast*, an expression from one data type to another.
- When you perform a division operation on two integers, the result is an integer. To get a more accurate result, you can cast one of the integer values as a decimal. That way, the result will be a decimal.
- CAST is an ANSI-standard function and is used more frequently than CONVERT, which is unique to SQL Server. You should use CONVERT when you need the additional formatting capabilities it provides. See figure 8-8 for details.

Figure 8-7 How to convert data using the CAST function

How to convert data using the CONVERT function

Although the CAST function is an ANSI-standard function, SQL Server provides another function you can use to convert data: CONVERT. Figure 8-8 shows you how to use this function.

In the syntax at the top of this figure, you can see that the CONVERT function provides an optional style argument. You can use this argument to specify the format you want to use when you convert date/time, real, or money data to character data. Some of the common style codes are presented in this figure. For a complete list of codes, please refer to the documentation for SQL Server.

The SELECT statement in this figure shows several examples of the CONVERT function. The first thing you should notice here is that if you don't code a style argument, the CONVERT function works just like CAST. This is illustrated by the first and fourth columns in the result set. Because of that, you'll probably use CONVERT only when you need to use one of the formats provided by the style argument.

The second and third columns both use the CONVERT function to format the InvoiceDate column. The second column uses a style code of 1, so the date is returned in the mm/dd/yy format. By contrast, the third column uses a style code of 107, so the date is returned in the Mon dd, yyyy format. Finally, the fifth column uses the CONVERT function to format the InvoiceTotal column with two digits to the right of the decimal point and commas to the left.

The syntax of the CONVERT function

```
CONVERT(data_type, expression [, style])
```

A SELECT statement that uses the CONVERT function

```
SELECT CONVERT(varchar, InvoiceDate) AS varcharDate,
       CONVERT(varchar, InvoiceDate, 1) AS varcharDate_1,
       CONVERT(varchar, InvoiceDate, 107) AS varcharDate_107,
       CONVERT(varchar, InvoiceTotal) AS varcharTotal,
       CONVERT(varchar, InvoiceTotal, 1) AS varcharTotal_1
FROM Invoices;
```

	varcharDate	varcharDate_1	varcharDate_107	varcharTotal	varcharTotal_1
1	2019-10-08	10/08/19	Oct 08, 2019	3813.33	3,813.33
2	2019-10-10	10/10/19	Oct 10, 2019	40.20	40.20
3	2019-10-13	10/13/19	Oct 13, 2019	138.75	138.75
4	2019-10-16	10/16/19	Oct 16, 2019	144.70	144.70

Common style codes for converting date/time data to character data

Code	Output format
0 or 100	Mon dd yyyy hh:miAM/PM
1 or 101	mm/dd/yy or mm/dd/yyyy
7 or 107	Mon dd, yy or Mon dd, yyyy
8 or 108	hh:mi:ss
10 or 110	mm-dd-yy or mm-dd-yyyy
12 or 112	yyymmdd or yyyyymmdd
14 or 114	hh:mi:ss:mmm (24-hour clock)

Common style codes for converting real data to character data

Code	Output
0 (default)	6 digits maximum
1	8 digits; must use scientific notation
2	16 digits; must use scientific notation

Common style codes for converting money data to character data

Code	Output
0 (default)	2 digits to the right of the decimal point; no commas to the left
1	2 digits to the right of the decimal point; commas to the left
2	4 digits to the right of the decimal point; no commas to the left

Description

- You can use the CONVERT function to explicitly convert an expression from one data type to another.
- You can use the optional style argument to specify the format to be used for date/time, real, and money values converted to character data. For a complete list of codes, search for “cast and convert” in the SQL Server documentation.

How to use the TRY_CONVERT function

When you use the CAST or CONVERT function, SQL Server returns an error if the expression can't be converted to the data type you specify. If that's not what you want, you can use the TRY_CONVERT function instead. Figure 8-9 shows how this function works.

If you compare the SELECT statement in this figure to the one in figure 8-8, you'll notice two differences. First, it uses the TRY_CONVERT function instead of the CONVERT function. If you look at the results of the first five columns, you'll see that they're identical to the results of the CONVERT function.

Second, a sixth column has been added to the SELECT clause. This column uses a TRY_CONVERT function that attempts to convert an invalid date to the date data type. As you can see, this function returns a value of NULL instead of generating an error. This is particularly useful if you need to test the value of a variable within a script. For more information on coding scripts, please see chapter 14.

Note that you can only use the TRY_CONVERT function with a data type that can be converted to the specified data type. As mentioned earlier, for example, the datetime2, date, and time data types can't be converted to any of the numeric data types. If you try to perform this type of conversion, SQL Server will return an error just like it does if you use the CAST or CONVERT function.

The syntax of the TRY_CONVERT function

```
TRY_CONVERT(data_type, expression [, style ])
```

A SELECT statement that uses the CONVERT function

```
SELECT TRY_CONVERT(varchar, InvoiceDate) AS varcharDate,  
       TRY_CONVERT(varchar, InvoiceDate, 1) AS varcharDate_1,  
       TRY_CONVERT(varchar, InvoiceDate, 107) AS varcharDate_107,  
       TRY_CONVERT(varchar, InvoiceTotal) AS varcharTotal,  
       TRY_CONVERT(varchar, InvoiceTotal, 1) AS varcharTotal_1,  
       TRY_CONVERT(date, 'Feb 29 2019') AS invalidDate  
FROM Invoices;
```

	varcharDate	varcharDate_1	varcharDate_107	varcharTotal	varcharTotal_1	invalidDate
1	2019-10-08	10/08/19	Oct 08, 2019	3813.33	3,813.33	NULL
2	2019-10-10	10/10/19	Oct 10, 2019	40.20	40.20	NULL
3	2019-10-13	10/13/19	Oct 13, 2019	138.75	138.75	NULL
4	2019-10-16	10/16/19	Oct 16, 2019	144.70	144.70	NULL
5	2019-10-16	10/16/19	Oct 16, 2019	15.50	15.50	NULL
6	2019-10-16	10/16/19	Oct 16, 2019	42.75	42.75	NULL
7	2019-10-21	10/21/19	Oct 21, 2019	172.50	172.50	NULL
8	2019-10-24	10/24/19	Oct 24, 2019	95.00	95.00	NULL

Description

- You can use the TRY_CONVERT function to explicitly convert an expression from one data type to another.
- If the TRY_CONVERT function can't convert the expression to the specified data type, it returns a NULL value instead of generating an error.
- You can use the optional style argument to specify the format to be used for date/time, real, and money values converted to character data. See figure 8-8 for more information.
- The TRY_CONVERT function can only be used with an expression that has a data type that can be explicitly converted to the specified data type.

Figure 8-9 How to convert data using the TRY_CONVERT function

How to use other data conversion functions

Although CAST, CONVERT, and TRY_CONVERT are the conversion functions you'll use most often, SQL Server provides some additional functions to perform special types of conversions. These functions are presented in figure 8-10.

You can use the first function, STR, to convert a floating-point value to a character value. You can think of this function as two conversion functions combined into one. First, it converts a floating-point value to a decimal value with the specified length and number of digits to the right of the decimal point. Then, it converts the decimal value to a character value. The function in this figure, for example, converts the number 1234.5678 to a string with a maximum length of seven characters and one digit to the right of the decimal point. Notice that the decimal digits are rounded rather than truncated.

The other four functions are used to convert characters to their equivalent numeric code and vice versa. CHAR and ASCII work with standard character strings that are stored one byte per character. The CHAR function shown in this figure, for example, converts the number 79 to its equivalent ASCII code, the letter O. Conversely, the ASCII function converts the letter O to its numeric equivalent of 79. Notice that although the string in the ASCII function can include more than one character, only the first character is converted.

The NCHAR and UNICODE functions convert Unicode characters to and from their numeric equivalents. You can see how these functions work in the examples. Notice in the last example that to code a Unicode character as a literal value, you have to precede the literal with the letter N.

CHAR is frequently used to output ASCII control characters that can't be typed on your keyboard. The three most common control characters are presented in this figure. These characters can be used to format output so it's easy to read. The SELECT statement in this figure, for example, uses the CHAR(13) and CHAR(10) control characters to start new lines after the vendor name and vendor address in the output.

Other data conversion functions

Function	Description
STR(float [,length[,decimal]])	Converts a floating-point number to a character string with the given length and number of digits to the right of the decimal point. The length must include one character for the decimal point and one character for the sign. The sign is blank if the number is positive.
CHAR(integer)	Converts the ASCII code represented by an integer between 0 and 255 to its character equivalent.
ASCII(string)	Converts the leftmost character in a string to its equivalent ASCII code.
NCHAR(integer)	Converts the Unicode code represented by an integer between 0 and 65535 to its character equivalent.
UNICODE(string)	Converts the leftmost character in a UNICODE string to its equivalent UNICODE code.

Examples that use the data conversion functions

Function	Result
STR(1234.5678, 7, 1)	1234.6
CHAR(79)	O
ASCII('Orange')	79
NCHAR(332)	O
UNICODE(N'Or')	332

ASCII codes for common control characters

Control character	Value
Tab	Char(9)
Line feed	Char(10)
Carriage return	Char(13)

A SELECT statement that uses the CHAR function to format output

```
SELECT VendorName + CHAR(13) + CHAR(10)
      + VendorAddress1 + CHAR(13) + CHAR(10)
      + VendorCity + ', ' + VendorState + ' ' + VendorZipCode
FROM Vendors
WHERE VendorID = 1;
```

US Postal Service
Attn: Supt. Window Services
Madison, WI 53707

Description

- The CHAR function is typically used to insert control characters into a character string.
- To code a Unicode value as a literal, precede the value with the character N.

Figure 8-10 How to use other data conversion functions

Perspective

In this chapter, you learned about the different SQL Server data types. In addition, you learned how to use some functions for converting data from one type to another. In the next chapter, you'll learn about some of the additional functions for working with data.

Terms

data type	exact numeric data types
string data type	approximate numeric data types
numeric data type	Unicode character
temporal data type	supplementary characters
date/time data type	national character
date data type	fixed-length string
integer data type	variable-length string
decimal data type	bit
scale	ASCII (American Standard Code for Information Interchange)
precision	large value data types
real data type	implicit conversion
fixed-point number	explicit conversion
floating-point number	cast
significant digits	Universal Time Coordinate (UTC)
single-precision number	Greenwich Mean Time
double-precision number	
scientific notation	

Exercises

1. Write a SELECT statement that returns four columns based on the InvoiceTotal column of the Invoices table:
 - Use the CAST function to return the first column as data type decimal with 2 digits to the right of the decimal point.
 - Use CAST to return the second column as a varchar.
 - Use the CONVERT function to return the third column as the same data type as the first column.
 - Use CONVERT to return the fourth column as a varchar, using style 1.
2. Write a SELECT statement that returns three columns based on the InvoiceDate column of the Invoices table:
 - Use the CAST function to return the first column as data type varchar.
 - Use the CONVERT function to return the second and third columns as a varchar, using style 1 and style 10, respectively.

9

How to use functions

In chapter 3, you were introduced to some of the scalar functions that you can use in a SELECT statement. Now, this chapter expands on that coverage by presenting many more of the scalar functions. When you complete this chapter, you'll have a thorough understanding of the functions that you can use with SQL Server.

How to work with string data.....	262
A summary of the string functions.....	262
How to solve common problems that occur with string data.....	266
How to work with numeric data.....	268
A summary of the numeric functions.....	268
How to solve common problems that occur with numeric data.....	270
How to work with date/time data	272
A summary of the date/time functions	272
How to parse dates and times	276
How to perform operations on dates and times.....	278
How to perform a date search.....	280
How to perform a time search	282
Other functions you should know about	284
How to use the CASE function.....	284
How to use the IIF and CHOOSE functions	286
How to use the COALESCE and ISNULL functions	288
How to use the GROUPING function	290
How to use the ranking functions.....	292
How to use the analytic functions	296
Perspective	300

How to work with string data

SQL Server provides a number of functions for working with string data. You'll learn how to use some of those functions in the topics that follow. In addition, you'll learn how to solve two common problems that can occur when you work with string data.

A summary of the string functions

Part 1 of figure 9-1 summarizes the string functions that are available with SQL Server. Most of these functions are used to perform string manipulation. For example, you can use the LEN function to get the number of characters in a string. Note that this function counts spaces at the beginning of the string (leading spaces), but not spaces at the end of the string (trailing spaces). If you want to remove leading or trailing spaces from a string, you can use the LTRIM or RTRIM function. If you want to remove both leading and trailing spaces, you can use the TRIM function.

You can use the LEFT and RIGHT functions to get the specified number of characters from the beginning and end of a string. You can use the SUBSTRING function to get the specified number of characters from anywhere in a string. You can use the REPLACE function to replace a substring within a string with another substring. You can use the TRANSLATE function to replace one or more characters in a string with other characters. In some cases, you can use this function instead of multiple REPLACE functions. And you can use the REVERSE function to reverse the order of the characters in a string.

The CHARINDEX function lets you locate the first occurrence of a substring within another string. The return value is a number that indicates the position of the substring. Note that you can start the search at a position other than the beginning of the string by including the start argument.

The PATINDEX function is similar to CHARINDEX. Instead of locating a string, however, it locates a string pattern. Like the string patterns you learned about in chapter 3 for use with the LIKE operator, the string patterns you use with the PATINDEX function can include wildcard characters. You can refer back to chapter 3 if you need to refresh your memory on how to use these characters.

The CONCAT function lets you concatenate two or more values into a single string. Although this function is similar to the concatenation operator, it lets you concatenate values other than strings. To do that, it implicitly converts all values to strings. That includes null values, which it converts to empty strings. The CONCAT_WS (concatenate with separator) function works like the CONCAT function, but it lets you specify a delimiter that's used to separate the values.

The last three functions should be self-explanatory. You use the LOWER and UPPER functions to convert the characters in a string to lower or upper case. And you use the SPACE function to return a string that has the specified number of spaces.

Some of the string functions

Function	Description
LEN(string)	Returns the number of characters in the string. Leading spaces are included, but trailing spaces are not.
LTRIM(string)	Returns the string with any leading spaces removed.
RTRIM(string)	Returns the string with any trailing spaces removed.
TRIM(string)	Returns the string with any leading and trailing spaces removed.
LEFT(string,length)	Returns the specified number of characters from the beginning of the string.
RIGHT(string,length)	Returns the specified number of characters from the end of the string.
SUBSTRING(string,start,length)	Returns the specified number of characters from the string starting at the specified position.
REPLACE(search,find,replace)	Returns the search string with all occurrences of the find string replaced with the replace string.
TRANSLATE(search,find,replace)	Returns the search string with characters in the find string replaced with the characters in the replace string.
REVERSE(string)	Returns the string with the characters in reverse order.
CHARINDEX(find,search[,start])	Returns an integer that represents the position of the first occurrence of the find string in the search string starting at the specified position. If the starting position isn't specified, the search starts at the beginning of the string. If the string isn't found, the function returns zero.
PATINDEX(find,search)	Returns an integer that represents the position of the first occurrence of the find pattern in the search string. If the pattern isn't found, the function returns zero. The find pattern can include wildcard characters. If the pattern begins with a wildcard, the value returned is the position of the first non-wildcard character.
CONCAT(value1,value2[,value3]...)	Returns a string that contains a concatenation of the specified values. The values are implicitly converted to strings. A null value is converted to an empty string.
CONCAT_WS(delimiter,value1, value2[,value3]...)	Same as CONCAT but the values are separated by the specified delimiter.
LOWER(string)	Returns the string converted to lowercase letters.
UPPER(string)	Returns the string converted to uppercase letters.
SPACE(integer)	Returns a string with the specified number of space characters (blanks).

Notes

- The start argument must be an integer from 1 to the length of the string.
- The TRIM, TRANSLATE, and CONCAT_WS functions were introduced with SQL Server 2017.

Figure 9-1 A summary of the string functions (part 1 of 2)

Part 2 of figure 9-1 presents examples of most of the string functions. If you study the examples at the top of this figure, you shouldn't have any trouble figuring out how they work. If you're confused by any of them, though, you can refer back to part 1 to check the syntax.

The SELECT statement shown in this figure illustrates how you can use the LEFT and RIGHT functions to format columns in a result set. In this case, the LEFT function is used to retrieve the first character of the VendorContactFName column in the Vendors table, which contains the first name of the vendor contact. In other words, this function retrieves the first initial of the vendor contact. Then, this initial is combined with the last name of the vendor contact and two literal values. You can see the result in the second column of the result set.

The third column in the result set lists the vendor's phone number without an area code. To accomplish that, this column specification uses the RIGHT function to extract the eight rightmost characters of the VendorPhone column. This assumes, of course, that all of the phone numbers are stored in the same format, which isn't necessarily the case since the VendorPhone column is defined as varchar(50).

This SELECT statement also shows how you can use a function in the search condition of a WHERE clause. This condition uses the SUBSTRING function to select only those rows with an area code of 559. To do that, it retrieves three characters from the VendorPhone column starting with the second character. Again, this assumes that the phone numbers are all in the same format and that the area code is enclosed in parentheses.

String function examples

Function	Result
LEN('SQL Server')	10
LEN(' SQL Server ')	12
LEFT('SQL Server', 3)	'SQL'
LTRIM(' SQL Server ')	'SQL Server '
RTRIM(' SQL Server ')	' SQL Server'
TRIM(' SQL Server ')	'SQL Server'
LOWER('SQL Server')	'sql server'
UPPER('ca')CA	
PATINDEX('%v_r%', 'SQL Server')	8
CHARINDEX('SQL', ' SQL Server')	3
CHARINDEX('-', '(559) 555-1212')	10
SUBSTRING('(559) 555-1212', 7, 8)	555-1212
REPLACE(RIGHT('(559) 555-1212', 13), ') ', '-')	559-555-1212
TRANSLATE('(XDG) 197.TS224', '().', '[]-')	[XDG] 197-TS224
CONCAT('Run time: ', 1.52, ' seconds')	Run time: 1.52 seconds
CONCAT_WS('.', 559, 555, 1212)	559.555.1212

A SELECT statement that uses the LEFT, RIGHT, and SUBSTRING functions

```
SELECT VendorName, VendorContactLName + ', ' + LEFT(VendorContactFName, 1)
      + '.' AS ContactName, RIGHT(VendorPhone, 8) AS Phone
FROM Vendors
WHERE SUBSTRING(VendorPhone, 2, 3) = 559
ORDER BY VendorName;
```

	VendorName	ContactName	Phone
1	Abbey Office Furnishings	Francis, K.	555-8300
2	BFI Industries	Kaleigh, E.	555-1551
3	Bill Marvin Electric Inc	Hostley, K.	555-5106
4	Cal State Temite	Hunter, D.	555-1534
5	California Business Machines	Rohansen, A.	555-5570
6	California Data Marketing	Jonessen, M.	555-3801
7	City Of Fresno	Mayte, K.	555-9999
8	Coffee Break Service	Smitzen, J.	555-8700

Figure 9-1 A summary of the string functions (part 2 of 2)

How to solve common problems that occur with string data

Figure 9-2 presents solutions to two common problems that occur when you work with string data. The first problem occurs when you store numeric data in a character column and then want to sort the column in numeric sequence.

To illustrate, look at the first example in this figure. Here, the columns in the StringSample table are defined with character data types. The first SELECT statement shows the result of sorting the table by the first column, which contains a numeric ID. As you can see, the rows are not in numeric sequence. That's because SQL Server interprets the values as characters, not as numbers.

One way to solve this problem is to convert the values in the ID column to integers for sorting purposes. This is illustrated in the second SELECT statement in this example. As you can see, the rows are now sorted in numeric sequence.

Another way to solve this problem is to pad the numbers with leading zeros or spaces so the numbers are aligned on the right. This is illustrated by the AltID column in this table, which is padded with zeros. If you sorted by this column instead of the first column, the rows would be returned in numeric sequence.

The second problem you'll encounter when working with string data occurs when two or more values are stored in the same string. For example, both a first and a last name are stored in the Name column of the StringSample table. If you want to work with the first and last names independently, you have to parse the string using the string functions. This is illustrated by the SELECT statement in the second example in this figure.

To extract the first name, this statement uses the LEFT and CHARINDEX functions. First, it uses the CHARINDEX function to locate the first space in the Name column. Then, it uses the LEFT function to extract all of the characters up to that space. Notice that one is subtracted from the value that's returned by the CHARINDEX function, so the space itself isn't included in the first name.

To extract the last name, this statement uses the RIGHT, LEN, and CHARINDEX functions. It uses the LEN function to get the number of characters in the Name column. Then, it uses the CHARINDEX function to locate the first space in the Name column, and it subtracts that value from the value returned by the LEN function. The result is the number of characters in the last name. That value is then used in the RIGHT function to extract the last name from the Name column.

As you review this example, you should keep in mind that I kept it simple so you can focus on how the string functions are used. You should realize, however, that this code won't work for all names. If, for example, a first name contains a space, such as in the name Jean Paul, this code won't work properly. That illustrates the importance of designing a database so this type of problem doesn't occur. You'll learn more about that in the next chapter. For now, just realize that if a database is designed correctly, you won't have to worry about this type of problem. Instead, this problem should occur only if you're importing data from another file or database system.

How to use the CAST function to sort by a string column that contains numbers

The StringSample table sorted by the ID column

```
SELECT * FROM StringSample
ORDER BY ID;
```

	ID	Name	AltID
1	1	Lizbeth Darien	01
2	17	Lance Pinos-Potter	17
3	2	Damell O'Sullivan	02
4	20	Jean Paul Renard	20
5	3	Alisha von Strump	03

The StringSample table sorted by the ID column cast to an integer

```
SELECT * FROM StringSample
ORDER BY CAST(ID AS int);
```

	ID	Name	AltID
1	1	Lizbeth Darien	01
2	2	Damell O'Sullivan	02
3	3	Alisha von Strump	03
4	17	Lance Pinos-Potter	17
5	20	Jean Paul Renard	20

How to use the string functions to parse a string

```
SELECT Name,
       LEFT(Name, CHARINDEX(' ', Name) - 1) AS First,
       RIGHT(Name, LEN(Name) - CHARINDEX(' ', Name)) AS Last
  FROM StringSample;
```

	Name	First	Last
1	Lizbeth Darien	Lizbeth	Darien
2	Damell O'Sullivan	Damell	O'Sullivan
3	Lance Pinos-Potter	Lance	Pinos-Potter
4	Jean Paul Renard	Jean	Paul Renard
5	Alisha von Strump	Alisha	von Strump

Description

- If you sort by a string column that contains numbers, you may receive unexpected results. To avoid that, you can convert the string column to a numeric value in the ORDER BY clause.
- If a string consists of two or more components, you can parse it into its individual components. To do that, you can use the CHARINDEX function to locate the characters that separate the components. Then, you can use the LEFT, RIGHT, SUBSTRING, and LEN functions to extract the individual components.

How to work with numeric data

In addition to the string functions, SQL Server provides several numeric functions. Although you'll probably use only a couple of these functions on a regular basis, you should be aware of all of them in case you ever need them. After you learn about these functions, I'll show you how you can use them and some of the other functions you've learned about in this chapter to solve common problems that occur when you work with numeric data.

A summary of the numeric functions

Figure 9-3 summarizes eight of the numeric functions SQL Server provides. The function you'll probably use most often is ROUND. This function rounds a number to the precision specified by the length argument. Note that you can round the digits to the left of the decimal point by coding a negative value for this argument. However, you're more likely to code a positive number to round the digits to the right of the decimal point. You can also use the ROUND function to truncate a number to the specified length. To do that, you can code any integer value other than zero for the optional function argument.

The first set of examples in this figure shows how the ROUND function works. The first example rounds the number 12.5 to a precision of zero, which means that the result has no significant digits to the right of the decimal point. Note that this function does not change the precision of the value. The result still has one digit to the right of the decimal point. The number has just been rounded so the digit is insignificant. To make that point clear, the second example rounds a number with four decimal places to a precision of zero. Notice that the result still has four digits to the right of the decimal point; they're just all zero.

The next three examples show variations of the first two examples. The third example rounds a number with four decimal places to a precision of 1, and the fourth example rounds the digits to the left of the decimal point to a precision of one. Finally, the last example truncates a number to a precision of zero.

The other function you're likely to use is ISNUMERIC. This function returns a Boolean value that indicates if an expression is numeric. This is illustrated by the next set of examples in this figure. This function can be useful for testing the validity of a value before saving it in a table.

You can use the next three functions, ABS, CEILING, and FLOOR, to get the absolute value of a number, the smallest integer greater than or equal to a number, or the largest integer less than or equal to a number. If you study the examples, you shouldn't have any trouble figuring out how these functions work.

The next two functions, SQUARE and SQRT, are used to calculate the square and square root of a number. And the last function, RAND, generates a floating-point number with a random value between 0 and 1. SQL Server provides a variety of functions like these for performing mathematical calculations, but you're not likely to use them. For a complete list of these functions, you can search for "mathematical functions" in the SQL Server documentation.

Some of the numeric functions

Function	Description
ROUND(number,length [,function])	Returns the number rounded to the precision specified by length. If length is positive, the digits to the right of the decimal point are rounded. If it's negative, the digits to the left of the decimal point are rounded. To truncate the number rather than round it, code a non-zero value for function.
ISNUMERIC(expression)	Returns a value of 1 (true) if the expression is a numeric value; returns a value of 0 (false) otherwise.
ABS(number)	Returns the absolute value of the number.
CEILING(number)	Returns the smallest integer that is greater than or equal to the number.
FLOOR(number)	Returns the largest integer that is less than or equal to the number.
SQUARE(float_number)	Returns the square of a floating-point number.
SQRT(float_number)	Returns the square root of a floating-point number.
RAND([integer])	Returns a random floating-point number between 0 and 1. If integer is coded, it provides a starting value for the function. Otherwise, the function will return the same number each time it's invoked within the same query.

Examples that use the numeric functions

Function	Result
ROUND(12.5,0)	13.0
ROUND(12.4999,0)	12.0000
ROUND(12.4999,1)	12.5000
ROUND(12.4999,-1)	10.0000
ROUND(12.5,0,1)	12.0
ISNUMERIC(-1.25)	1
ISNUMERIC('SQL Server')	0
ISNUMERIC('2020-04-30')	0
ABS(-1.25)	1.25
CEILING(-1.25)	-1
FLOOR(-1.25)	-2
CEILING(1.25)	2
FLOOR(1.25)	1
SQUARE(5.2786)	27.86361796
SQRT(125.43)	11.199553562531
RAND()	0.243729

Note

- To calculate the square or square root of a number with a data type other than float or real, you must cast it to a floating-point number.

Figure 9-3 A summary of the numeric functions

How to solve common problems that occur with numeric data

In the previous chapter, you learned that numbers with the real data types don't contain exact values. The details of why that is are beyond the scope of this book. From a practical point of view, though, that means that you don't want to search for exact values when you're working with real numbers. If you do, you'll miss values that are in essence equal to the value you're looking for.

To illustrate, consider the RealSample table shown in figure 9-4. This table includes a column named R that's defined with the float(53) data type. Now, consider what would happen if you selected all the rows where the value of R is equal to 1. The result set would include only the second row, even though the table contains two other rows that have values approximately equal to 1.

When you perform a search on a column with a real data type, then, you usually want to search for an approximate value. This figure shows two ways to do that. First, you can search for a range of values. The first SELECT statement in this figure, for example, searches for values between .99 and 1.01. Second, you can search for values that round to an exact value. This is illustrated by the second SELECT statement. Both of these statements return the three rows in the RealSample table that are approximately equal to 1. In fact, the value in the first row is so close to 1 that the Management Studio removes the decimal places from this number when it displays the results.

Although both of the SELECT statements shown here return the same results, the first statement is more efficient than the second one. That's because SQL Server isn't able to optimize a query that uses a function in its search condition. Because of that, I recommend you use the range technique to search for a real value whenever possible.

Another problem you may face is formatting numeric values so they're easy to read. One way to do that is to format them so they're aligned on the right, as shown in the third SELECT statement. To do this, the real numbers in the R column are first cast as decimal numbers to give them a consistent scale. Then, the decimal values are cast as character data and padded on the left with spaces to right-align the data as shown in the column named R_Formatted.

If you look at the expression for the last column, you'll see that it's quite complicated. If you break it down into its component parts, however, you shouldn't have much trouble understanding how it works. To help you break it down, the third, fourth, and fifth columns in the result set show the interim results returned by portions of the expression.

To align the values at the right, the last column specification assumes a column width of nine characters. Then, the length of the number to be formatted is subtracted from nine, and the SPACE function is used to create a string with the resulting number of spaces. Finally, the number is concatenated to the string of spaces after it's converted to a string value. The result is a string column with the numbers aligned at the right. However, this formatting isn't displayed properly when you use the Management Studio to view the results in the grid. As a result, to view this formatting, you must click on the Results to Text button to view the results as text as shown in this figure.

The RealSample table

ID	R
1	1.00000000000000011
2	1
3	0.999999999999999
4	1234.56789012345
5	999.04440209348
6	24.04849

How to search for approximate real values

A SELECT statement that searches for a range of values

```
SELECT * FROM RealSample
WHERE R BETWEEN 0.99 AND 1.01;
```

A SELECT statement that searches for rounded values

```
SELECT * FROM RealSample
WHERE ROUND(R,2) = 1;
```

ID	R
1	1
2	1
3	0.999999999999999

A SELECT statement that formats real numbers

```
SELECT ID, R, CAST(R AS decimal(9,3)) AS R_decimal,
       CAST(CAST(R AS decimal(9,3)) AS varchar(9)) AS R_varchar,
       LEN(CAST(CAST(R AS decimal(9,3)) AS varchar(9))) AS R_LEN,
       SPACE(9 - LEN(CAST(CAST(R AS decimal(9,3)) AS varchar(9)))) +
       CAST(CAST(R AS decimal(9,3)) AS varchar(9)) AS R_Formatted
FROM RealSample;
```

R_decimal	R_varchar	R_LEN	R_Formatted
1.000	1.000	5	1.000
1.000	1.000	5	1.000
1.000	1.000	5	1.000
1234.568	1234.568	8	1234.568
999.044	999.044	7	999.044
24.048	24.048	6	24.048

Description

- Because real values are approximate, you'll want to search for approximate values when retrieving real data. To do that, you can specify a range of values, or you can use the ROUND function to search for rounded values.
- When you display real or decimal values, you may want to format them so they're aligned on the right.

How to work with date/time data

In the topics that follow, you'll learn how to use some of the functions SQL Server provides for working with dates and times. As you'll see, these include functions for extracting different parts of a date/time value and for performing operations on dates and times. In addition, you'll learn how to perform different types of searches on date/time values.

A summary of the date/time functions

Figure 9-5 presents a summary of the date/time functions and shows how some of them work. One of the functions you'll use frequently is GETDATE, which gets the current local date and time from your system. GETUTCDATE is similar, but it returns the Universal Time Coordinate (UTC) date, also known as Greenwich Mean Time (GMT).

Although you probably won't use the GETUTCDATE function often, it's useful if your system will operate in different time zones. That way, the date/time values will always reflect Greenwich Mean Time, regardless of the time zone in which they're entered. For example, a date/time value entered at 11:00 a.m. Los Angeles time would be given the same value as a date/time value entered at 2:00 p.m. New York time. That makes it easy to compare and operate on these values.

The next three functions (SYSDATETIME, SYSUTCDATETIME, and SYSDATETIMEOFFSET) work similarly to the first two functions. However, they return the datetime2 and datetimeoffset types that were introduced with SQL Server 2008. As a result, these functions return a more precise fractional second value. In addition, the SYSDATETIMEOFFSET function returns a value that includes a time zone offset. Note that the time zone offset is not adjusted for daylight savings time.

The next five functions (DAY, MONTH, YEAR, DATENAME, and DATEPART) let you extract different parts of a date value. For more information about these functions, you can refer to figure 9-6. For now, just realize that when you use the DATEPART and DATENAME functions, you can retrieve any of the date parts listed in part 2 of this figure.

The DATEADD and DATEDIFF functions let you perform addition and subtraction operations on date/time values. As you can see, these functions let you specify the date part to be added. For more information about these functions, you can refer to figure 9-7.

The TODATETIMEOFFSET and SWITCHOFFSET functions let you work with the datetimeoffset data type. In particular, you can use the TODATETIMEOFFSET function to add a time zone offset to a datetime2 value and return a datetimeoffset value. In addition, you can use the SWITCHOFFSET function to specify a new time zone offset value for a datetimeoffset value.

The next two functions, EOMONTH and DATEFROMPARTS, were introduced with SQL Server 2012. The EOMONTH function gets the last day of the month for the specified date. This can be helpful for determining what days are valid for a given month. The DATEFROMPARTS function lets you create

Some of the date/time functions

Function	Description
GETDATE()	Returns a datetime value for the current local date and time based on the system's clock.
GETUTCDATE()	Returns a datetime value for the current UTC date and time based on the system's clock and time zone setting.
SYSDATETIME()	Returns a datetime2(7) value for the current local date and time based on the system's clock.
SYSUTCDATETIME()	Returns a datetime2(7) value for the current UTC date and time based on the system's clock and time zone setting.
SYSDATETIMEOFFSET()	Returns a datetimeoffset(7) value for the current UTC date and time based on the system's clock and time zone setting with a time zone offset that is <i>not</i> adjusted for daylight savings time.
DAY(date)	Returns the day of the month as an integer.
MONTH(date)	Returns the month as an integer.
YEAR(date)	Returns the 4-digit year as an integer.
DATENAME(datepart,date)	Returns the part of the date specified by datepart as a character string.
DATEPART(datepart,date)	Returns the part of the date specified by datepart as an integer.
DATEADD(datepart,number,date)	Returns the date that results from adding the specified number of datepart units to the date.
DATEDIFF(datepart,startdate,enddate)	Returns the number of datepart units between the specified start and end dates.
TODATETIMEOFFSET(datetime2,tzoffset)	Returns a datetimeoffset value that results from adding the specified time zone offset to the specified datetime2 value.
SWITCHOFFSET(datetimeoffset,tzoffset)	Returns a datetimeoffset value that results from switching the time zone offset for the specified datetimeoffset value to the specified offset.
EOMONTH(startdate[,months])	Returns a date value for the last day of the month specified by the start date. If months is specified, the number of months is added to the start date before the end-of-month date is calculated.
DATEFROMPARTS(year,month,day)	Returns a date value for the specified year, month, and day.
ISDATE(expression)	Returns a value of 1 (true) if the expression is a valid date/time value; returns a value of 0 (false) otherwise.

Figure 9-5 A summary of the date/time functions (part 1 of 2)

a date value for a given year, month, and day. In addition to this function, SQL Server 2012 introduced other functions that let you create datetime, smalldatetime, time, datetime2, and datetimeoffset values. For more information, please search for “date and time functions” in the SQL Server documentation.

The last function (ISDATE) returns a Boolean value that indicates whether an expression can be cast as a valid date/time value. This function is useful for testing the validity of a date/time value before it’s saved to a table. This is illustrated by the last set of examples. Here, you can see that the first and third expressions are valid dates, but the second and fourth expressions aren’t. The second expression isn’t valid because the month of September has only 30 days. And the fourth expression isn’t valid because a time value can have a maximum of 59 minutes and 59 seconds. Note that this function checks for both a valid date/time format and a valid date/time value.

The first two sets of examples illustrate the differences between the functions that return date/time values. To start, there’s a 7 hour difference between the datetime value that’s returned by the GETDATE and GETUTCDATETIME functions. That’s because I ran these functions from California, which is 7 hours behind the Universal Time Coordinate (UTC). In addition, note that the datetime2(7) value that’s returned by the SYSDATETIME function provides more precise fractional second values than the datetime value that’s returned by the GETDATE and GETUTCDATETIME functions. Finally, note that the SYSDATETIMEOFFSET function returns a datetimeoffset value that includes a time zone offset.

The third set of examples shows how you can use the date parts with the DATEPART and DATENAME functions. To start, you don’t need to specify a date part when you use the MONTH function to return an integer value for the month. However, you can get the same result with the DATEPART function by specifying the month date part as the first argument. Or, if you want to return the name of the month as a string of characters, you can specify the month date part as the first argument of the DATENAME function. Finally, you can use an abbreviation for a date part whenever that makes sense. However, I generally prefer to avoid abbreviations as they tend to make the code more difficult to read and understand.

The fourth set of examples shows how to use the EOMONTH and DATEFROMPARTS functions that were introduced with SQL Server 2012. Here, the first expression uses the EOMONTH function to return a date for the last day of the month for February 1, 2020. Since 2020 is a leap year, this returns February 29, 2020. The second expression is similar, but it adds two months to the specified date. Finally, the last expression uses the DATEFROMPARTS function to create a date with a year value of 2020, a month value of 4, and a day value of 3.

Date part values and abbreviations

Argument	Abbreviations
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw
hour	hh
minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns
tzoffset	tz

Examples that use date/time functions

Function	Result
<code>GETDATE()</code>	2020-04-30 14:10:13.813
<code>GETUTCDATE()</code>	2020-04-30 21:10:13.813
<code>SYSDATETIME()</code>	2020-04-30 14:10:13.8160822
<code>SYSUTCDATETIME()</code>	2020-04-30 21:10:13.8160822
<code>SYSDATETIMEOFFSET()</code>	2020-04-30 14:10:13.8160822 -07.00
<code>MONTH('2020-04-30')</code>	4
<code>DATEPART(month, '2020-04-30')</code>	4
<code>DATENAME(month, '2020-04-30')</code>	April
<code>DATENAME(m, '2020-04-30')</code>	April
<code>EOMONTH('2020-02-01')</code>	2020-02-29
<code>EOMONTH('2020-02-01', 2)</code>	2020-04-30
<code>DATEFROMPARTS(2020, 4, 3)</code>	2020-04-03
<code>ISDATE('2020-04-30')</code>	1
<code>ISDATE('2020-04-31')</code>	0
<code>ISDATE('23:59:59')</code>	1
<code>ISDATE('23:99:99')</code>	0

Figure 9-5 A summary of the date/time functions (part 2 of 2)

How to parse dates and times

Figure 9-6 shows you how to use the DAY, MONTH, YEAR, DATEPART, and DATENAME functions to parse dates and times. If you just need to get an integer value for a day, month, or year, you should use the DAY, MONTH, and YEAR functions as shown in the examples at the top of this figure since these are ANSI-standard functions. If you need to extract another part of a date or time as an integer, however, you'll need to use the DATEPART function. And if you need to extract a date part as a string, you'll need to use the DATENAME function.

This figure shows the result of using each of the date part values with the DATEPART and DATENAME functions. As you can see, many of the values returned by the two functions appear to be the same. Keep in mind, however, that all of the values returned by DATEPART are integers. By contrast, all of the values returned by DATENAME are strings. That's why the month and week day are returned as names rather than numbers when you use DATENAME. The function you use, then, will depend on what you need to do with the date part. If you need to use it in an arithmetic operation, for example, you'll want to use the DATEPART function. But if you need to use it in a concatenation, you'll want to use the DATENAME function.

Finally, it's important to note the difference between the DATEPART and DATENAME functions when working with the tzoffset date part. With this part, the DATEPART function returns an integer value for the number of minutes for the time zone offset, and the DATENAME function returns a string value that specifies the hours and minutes.

Examples that use the DAY, MONTH, and YEAR functions

Function	Result
DAY('2020-04-30')	30
MONTH('2020-04-30')	4
YEAR('2020-04-30')	2020

Examples that use the DATEPART function

Function	Result
DATEPART(day, '2020-04-30 11:35:00')	30
DATEPART(month, '2020-04-30 11:35:00')	4
DATEPART(year, '2020-04-30 11:35:00')	2020
DATEPART(hour, '2020-04-30 11:35:00')	11
DATEPART(minute, '2020-04-30 11:35:00')	35
DATEPART(second, '2020-04-30 11:35:00')	0
DATEPART(quarter, '2020-04-30 11:35:00')	2
DATEPART(dayofyear, '2020-04-30 11:35:00')	121
DATEPART(week, '2020-04-30 11:35:00')	18
DATEPART(weekday, '2020-04-30 11:35:00')	5
DATEPART(millisecond, '11:35:00.1234567')	123
DATEPART(microsecond, '11:35:00.1234567')	123456
DATEPART(nanosecond, '11:35:00.1234567')	123456700
DATEPART(tzoffset, '11:35:00.1234567 -07:00')	-420

Examples that use the DATENAME function

Function	Result
DATENAME(day, '2020-04-30 11:35:00')	30
DATENAME(month, '2020-04-30 11:35:00')	April
DATENAME(year, '2020-04-30 11:35:00')	2020
DATENAME(hour, '2020-04-30 11:35:00')	11
DATENAME(minute, '2020-04-30 11:35:00')	35
DATENAME(second, '2020-04-30 11:35:00')	0
DATENAME(quarter, '2020-04-30 11:35:00')	2
DATENAME(dayofyear, '2020-04-30 11:35:00')	121
DATENAME(week, '2020-04-30 11:35:00')	18
DATENAME(weekday, '2020-04-30 11:35:00')	Thursday
DATENAME(millisecond, '11:35:00.1234567')	123
DATENAME(microsecond, '11:35:00.1234567')	123456
DATENAME(nanosecond, '11:35:00.1234567')	123456700
DATENAME(tzoffset, '11:35:00.1234567 -07:00')	-07:00

Notes

- When you use weekday with the DATEPART function, it returns an integer that indicates the day of the week where 1=Sunday, 2=Monday, etc.
- The DAY, MONTH, and YEAR functions are ANSI-standard functions. The DATEPART and DATENAME functions are more general-purpose functions provided by SQL Server.

How to perform operations on dates and times

Figure 9-7 shows you how to use the DATEADD and DATEDIFF functions to perform operations on dates and times. You can use the DATEADD function to add a specified number of date parts to a date. The first eight DATEADD functions in this figure, for example, show how you can add one day, month, year, hour, minute, second, quarter, and week to a date/time value. If you want to subtract date parts from a date/time value, you can do that with the DATEADD function too. Just code the number argument as a negative value, as illustrated by the next to last DATEADD function. The last DATEADD function illustrates that you can't add a fractional number of date parts to a date/time value. If you try to, the fractional portion is ignored.

If you need to find the difference between two date/time values, you can use the DATEDIFF function as illustrated by the second set of examples in this figure. As you can see, the result is expressed in the date part units you specify. The first function, for example, returns the number of days between two dates, and the second example returns the number of months between the same two dates.

In most cases, the earlier date is specified as the second argument in the DATEDIFF function and the later date is specified as the third argument. That way, the result of the function is a positive value. However, you can also code the later date first. Then, the result is a negative value as you can see in the last DATEDIFF function in this figure.

If you use the DATEDIFF function, you should realize that it returns the number of date/time boundaries crossed, which is not necessarily the same as the number of intervals between two dates. To understand this, consider the third DATEDIFF function. This function returns the difference in years between the dates 2019-07-01 and 2020-04-30. Since the second date is less than one year after the first date, you might expect this function to return a value of zero. As you can see, however, it returns a value of 1 because it crossed the one-year boundary between the years 2019 and 2020. Because this is not intuitive, you'll want to use this function carefully.

The last three examples in this figure show how you can perform operations on dates and times without using the DATEADD and DATEDIFF functions. The first expression, for example, adds one day to a date/time value, and the second expression subtracts one day from the same value. When you use this technique, SQL Server assumes you're adding or subtracting days. So you can't add or subtract other date parts unless you express them as multiples or fractions of days. In addition, you can't use the addition and subtraction operators with the date, time, datetime2, and datetimeoffset data types.

The last expression shows how you can subtract two date/time values to calculate the number of days between them. Notice that after the dates are subtracted, the result is converted to an integer. That's necessary because the result of the subtraction operation is implicitly cast as a date/time value that represents the number of days after January 1, 1900. For this reason, the integer difference of 304 days is interpreted as the following date/time value:
1900-10-31 00:00:00:000.

Examples that use the DATEADD function

Function	Result
DATEADD(day, 1, '2020-04-30 11:35:00')	2020-05-01 11:35:00.000
DATEADD(month, 1, '2020-04-30 11:35:00')	2020-05-30 11:35:00.000
DATEADD(year, 1, '2020-04-30 11:35:00')	2021-04-30 11:35:00.000
DATEADD(hour, 1, '2020-04-30 11:35:00')	2020-04-30 12:35:00.000
DATEADD(minute, 1, '2020-04-30 11:35:00')	2020-04-30 11:36:00.000
DATEADD(second, 1, '2020-04-30 11:35:00')	2020-04-30 11:35:01.000
DATEADD(quarter, 1, '2020-04-30 11:35:00')	2020-07-30 11:35:00.000
DATEADD(week, 1, '2020-04-30 11:35:00')	2020-05-07 11:35:00.000
DATEADD(month, -1, '2020-04-30 11:35:00')	2020-03-30 11:35:00.000
DATEADD(year, 1.5, '2020-04-30 11:35:00')	2021-04-30 11:35:00.000

Examples that use the DATEDIFF function

Function	Result
DATEDIFF(day, '2019-07-01', '2020-04-30')	304
DATEDIFF(month, '2019-07-01', '2020-04-30')	9
DATEDIFF(year, '2019-07-01', '2020-04-30')	1
DATEDIFF(hour, '06:46:45', '11:35:00')	5
DATEDIFF(minute, '06:46:45', '11:35:00')	289
DATEDIFF(second, '06:46:45', '11:35:00')	17295
DATEDIFF(quarter, '2019-07-01', '2020-04-30')	3
DATEDIFF(week, '2019-07-01', '2020-04-30')	43
DATEDIFF(day, '2020-04-30', '2019-07-01')	-304

Examples that use the addition and subtraction operators

Operation	Result
CAST('2020-04-30 11:35:00' AS smalldatetime) + 1	2020-05-01 11:35:00
CAST('2020-04-30 11:35:00' AS smalldatetime) - 1	2020-04-29 11:35:00
CAST(CAST('2020-04-30' AS datetime) - CAST('2019-07-01' AS datetime) AS int)	304

Description

- You can use the DATEADD function to subtract a specified number of date parts from a date by coding the number of date parts as a negative value.
- If the number of date parts you specify in the DATEADD function isn't an integer, the fractional portion of the number is ignored.
- If the end date you specify in a DATEDIFF function is before the start date, the function will return a negative value.
- You can also use the addition and subtraction operators to add and subtract days from a date value. To add and subtract days from a date string, cast the string to a date/time value.
- You can also calculate the number of days between two dates by subtracting the date/time values and converting the result to an integer.

Figure 9-7 How to perform operations on dates and times

How to perform a date search

Because date/time values often contain both a date and a time component, searching for specific dates and times can be difficult. In this topic, you'll learn a variety of ways to ignore the time component when you search for a date value. And in the next topic, you'll learn how to ignore date components when you search for time values.

Before I go on, you should realize that the problems described here can sometimes be avoided by designing the database differently. For example, if you don't need to include a time component, you can use the date data type for the column that stores the date. That way, there's no time component to complicate your date searches. Conversely, if you don't need to include a date component, you can use the time data type for the column that holds the time. Of course, since these data types were introduced with SQL Server 2008, you'll need to use a different technique for prior versions of SQL Server.

Figure 9-8 illustrates the problem you can encounter when searching for dates. The examples in this figure use a table named DateSample. This table includes an ID column that contains an integer value and a StartDate column that contains a datetime value. Notice that the time components in the first three rows in this table have a zero value. By contrast, the time components in the next three rows have non-zero time components.

The problem occurs when you try to search for a date value. The first SELECT statement in this figure, for example, searches for rows in the DateSample table with the date 2019-10-28. Because a time component isn't specified, a zero time component is added when the date string is converted to a datetime value. However, because the row with this date has a non-zero time value, no rows are returned by this statement.

To solve this problem, you can use one of the five techniques shown in this figure. Of these techniques, the first technique is usually the easiest to use. Here, you use the CONVERT function to convert the datetime value to a date value. Of course, this only works for SQL Server 2008 or later.

As a result, if you need to work with a prior version of SQL Server, you'll need to use one of the other techniques. For example, you can use the second technique to search for dates that are greater than or equal to the date you're looking for and less than the date that follows the date you're looking for. Or, you can use the third technique to search for the values that are returned by the MONTH, DAY, and YEAR functions.

The fourth technique is to use the CAST function to convert the value in the StartDate column to an 11-character string. That causes the time portion of the date to be truncated (if you look back at figure 8-7, you'll see that when a date/time data type is cast to a string data type, the date portion contains 11 characters in the format "Mon dd yyyy"). Then, the string is converted back to a datetime value, which adds a zero time component.

The last technique is similar, but it uses the CONVERT function instead of the CAST function. The style code used in this function converts the datetime value to a 10-character string that doesn't include the time. Then, the string is converted back to a date with a zero time component.

The contents of the DateSample table

ID	StartDate
1	1990-11-01 00:00:00.000
2	2010-10-28 00:00:00.000
3	2015-06-30 00:00:00.000
4	2016-10-28 10:00:00.000
5	2019-10-28 13:58:32.823
6	2019-11-01 09:02:25.000

A search condition that fails to return a row

```
SELECT * FROM DateSample
WHERE StartDate = '2019-10-28';
```

Five SELECT statements that ignore time values

A SELECT statement that uses the date type to remove time values (SQL Server 2008 or later)

```
SELECT * FROM DateSample
WHERE CONVERT(date, StartDate) = '2019-10-28';
```

A SELECT statement that searches for a range of dates

```
SELECT * FROM DateSample
WHERE StartDate >= '2019-10-28' AND StartDate < '2019-10-29';
```

A SELECT statement that searches for month, day, and year components

```
SELECT * FROM DateSample
WHERE MONTH(StartDate) = 10 AND
      DAY(StartDate) = 28 AND
      YEAR(StartDate) = 2019;
```

A SELECT statement that uses the CAST function to remove time values

```
SELECT * FROM DateSample
WHERE CAST(CAST(StartDate AS char(11)) AS datetime) = '2019-10-28';
```

A SELECT statement that uses the CONVERT function to remove time values

```
SELECT * FROM DateSample
WHERE CONVERT(datetime, CONVERT(char(10), StartDate, 110)) = '2019-10-28';
```

The result set

ID	StartDate
1	2019-10-28 13:58:32.823

Description

- If you perform a search using a date string that doesn't include the time, the date string is converted implicitly to a date/time value with a zero time component. Then, if the date columns you're searching have non-zero time components, you have to accommodate the times in the search condition.

Note that the second technique (searching for a range of dates) is the only technique that doesn't use any functions in the WHERE clause. Because of that, this is the most efficient technique for searching for dates. As a result, you may want to use it even if you're using SQL Server 2008 or later.

How to perform a time search

When you search for a time value without specifying a date component, SQL Server automatically uses the default date of January 1, 1900. That's why neither of the first two SELECT statements in figure 9-9 return any rows. Even though at least one row has the correct time value for each search condition, those rows don't have the correct date value.

To solve this problem, you can use a SELECT statement like the one shown in the third or fourth example. In the third example, the CONVERT function is used to convert the datetime value to a time value. Of course, since the time data type was introduced with SQL Server 2008, this won't work for earlier versions of SQL Server.

As a result, if you need to use an earlier version of SQL Server, you can use a SELECT statement like the one shown in the fourth example. In this statement, the search condition uses the CONVERT function to convert the datetime values in the StartDate column to string values without dates. To do that, it uses a style argument of 8. Then, it converts the string values back to datetime values, which causes the default date to be used. That way, the date will match the dates that are added to the date literals.

The contents of the DateSample table

ID	StartDate
1	1990-11-01 00:00:00.000
2	2010-10-28 00:00:00.000
3	2015-06-30 00:00:00.000
4	2016-10-28 10:00:00.000
5	2019-10-28 13:58:32.823
6	2019-11-01 09:02:25.000

Two search conditions that fail to return a row

```
SELECT * FROM DateSample
WHERE StartDate = CAST('10:00:00' AS datetime);

SELECT * FROM DateSample
WHERE StartDate >= '09:00:00' AND
      StartDate < '12:59:59:999';
```

Two SELECT statements that ignore date values

A SELECT statement that removes date values (SQL Server 2008 or later)

```
SELECT * FROM DateSample
WHERE CONVERT(time, StartDate) >= '09:00:00' AND
      CONVERT(time, StartDate) < '12:59:59:999';
```

A SELECT statement that removes date values (prior to SQL Server 2008)

```
SELECT * FROM DateSample
WHERE CONVERT(datetime, CONVERT(char(12), StartDate, 8)) >= '09:00:00' AND
      CONVERT(datetime, CONVERT(char(12), StartDate, 8)) < '12:59:59:999';
```

The result set

ID	StartDate
1	2016-10-28 10:00:00.000
2	2019-11-01 09:02:25.000

Description

- If you perform a search using a date string that includes only a time, the date is converted implicitly to a date/time value with a default date component of 1900-01-01. Then, if the date columns you're searching have other dates, you have to accommodate those dates in the search condition.

Other functions you should know about

In addition to the conversion functions and the functions for working with specific types of data, SQL Server provides some other general purpose functions you should know about. Several of these functions are described in the topics that follow.

How to use the CASE function

Figure 9-10 presents the two formats of the CASE function. This function returns a value that's determined by the conditions you specify. The easiest way to describe how this function works is to look at the two examples shown in this figure.

The first example uses a simple CASE function. When you use this function, SQL Server compares the input expression you code in the CASE clause with the expressions you code in the WHEN clauses. In this example, the input expression is a value in the TermsID column of the Invoices table, and the when expressions are the valid values for this column. When SQL Server finds a when expression that's equal to the input expression, it returns the expression specified in the matching THEN clause. If the value of the TermsID column is 3, for example, this function returns the value "Net due 30 days." Although it's not shown in this example, you can also code an ELSE clause at the end of the CASE function. Then, if none of the when expressions are equal to the input expression, the function returns the value specified in the ELSE clause.

The simple CASE function is typically used with columns that can contain a limited number of values, such as the TermsID column used in this example. By contrast, the searched CASE function can be used for a wide variety of purposes. For example, you can test for conditions other than equal with this function. In addition, each condition can be based on a different column or expression. The second example in this figure illustrates how this function works.

This example determines the status of the invoices in the Invoices table. To do that, the searched CASE function uses the DATEDIFF function to get the number of days between the current date and the invoice due date. If the difference is greater than 30, the CASE function returns the value "Over 30 days past due." Similarly, if the difference is greater than 0, the function returns the value "1 to 30 days past due." Notice that if an invoice is 45 days old, both of these conditions are true. In that case, the function returns the expression associated with the first condition since this condition is evaluated first. In other words, the sequence of the conditions is critical to getting logical results. If neither of the conditions is true, the function returns the value "Current."

Because the WHEN clauses in this example use greater than conditions, this CASE function couldn't be coded using the simple syntax. Of course, CASE functions can be more complicated than what's shown here, but this should give you an idea of what you can do with this function.

The syntax of the simple CASE function

```
CASE input_expression
    WHEN when_expression_1 THEN result_expression_1
    [WHEN when_expression_2 THEN result_expression_2]...
    [ELSE else_result_expression]
END
```

The syntax of the searched CASE function

```
CASE
    WHEN conditional_expression_1 THEN result_expression_1
    [WHEN conditional_expression_2 THEN result_expression_2]...
    [ELSE else_result_expression]
END
```

A SELECT statement that uses a simple CASE function

```
SELECT InvoiceNumber, TermsID,
CASE TermsID
    WHEN 1 THEN 'Net due 10 days'
    WHEN 2 THEN 'Net due 20 days'
    WHEN 3 THEN 'Net due 30 days'
    WHEN 4 THEN 'Net due 60 days'
    WHEN 5 THEN 'Net due 90 days'
END AS Terms
FROM Invoices;
```

InvoiceNumber	TermsID	Terms
6	963253261	3 Net due 30 days
7	963253237	3 Net due 30 days
8	125520-1	1 Net due 10 days

A SELECT statement that uses a searched CASE function

```
SELECT InvoiceNumber, InvoiceTotal, InvoiceDate, InvoiceDueDate,
CASE
    WHEN DATEDIFF(day, InvoiceDueDate, GETDATE()) > 30
        THEN 'Over 30 days past due'
    WHEN DATEDIFF(day, InvoiceDueDate, GETDATE()) > 0
        THEN '1 to 30 days past due'
    ELSE 'Current'
END AS Status
FROM Invoices
WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
```

InvoiceNumber	InvoiceTotal	InvoiceDate	InvoiceDueDate	Status
9	134116	2020-01-28	2020-02-17	Current
10	0-2436	2020-01-31	2020-02-29	Current
11	547480102	2020-02-01	2020-02-29	Current

Description

- The simple CASE function tests the expression in the CASE clause against the expressions in the WHEN clauses. Then, the function returns the result expression associated with the first test that results in an equal condition.
- The searched CASE function tests the conditional expression in each WHEN clause in sequence and returns the result expression for the first condition that evaluates to true.

Figure 9-10 How to use the CASE function

How to use the IIF and CHOOSE functions

Figure 9-11 shows how to use the two *logical functions* that were introduced with SQL Server 2012. The IIF function returns one of two values depending on the result of a conditional expression, and the CHOOSE function returns a value from a list of values depending on the index you specify.

The SELECT statement in the first example in this figure illustrates how the IIF function works. This statement groups the rows in the Invoices table by the VendorID column and returns three columns. The first column contains the VendorID for each vendor, and the second column contains the sum of the invoice totals for that vendor. Then, the third column contains a value that indicates if the sum of invoice totals is less than 1000 or greater than or equal to 1000. To do that, the first argument of the IIF function tests if the sum of invoice totals is less than 1000. If it is, a value of “Low” is returned. Otherwise, a value of “High” is returned.

If you compare the IIF function with the searched CASE function in the previous figure, you’ll see that it provides another way to test a conditional expression that can result in one of two values. In fact, SQL Server translates IIF functions to CASE functions before processing them. For example, SQL Server would translate the IIF function in this figure so it looks something like this:

```
SELECT VendorID, SUM(InvoiceTotal) AS SumInvoices,
CASE
    WHEN SUM(InvoiceTotal) < 1000
        THEN 'Low'
    ELSE
        'High'
END AS InvoiceRange
FROM Invoices
GROUP BY VendorID;
```

The technique you use is mostly a matter of preference.

The second SELECT statement in this figure illustrates how to use the CHOOSE function. Although this function isn’t as useful as some of the other functions presented in this book, it can be useful in certain situations. In this example, it’s used to return a description of the due days for the invoices in the Invoices table with a balance due based on the value of the TermsID column. This works because the TermsID column is an int type.

The syntax of the IIF function

```
IIF(conditional_expression, true_value, false_value)
```

The syntax of the CHOOSE function

```
CHOOSE(index, value1, value2 [,value3]...)
```

A SELECT statement that uses the IIF function

```
SELECT VendorID, SUM(InvoiceTotal) AS SumInvoices,
       IIF(SUM(InvoiceTotal) < 1000, 'Low', 'High') AS InvoiceRange
  FROM Invoices
 GROUP BY VendorID;
```

	VendorID	SumInvoices	InvoiceRange
1	34	1200.12	High
2	37	564.00	Low
3	48	856.92	Low
4	72	21927.31	High
5	80	265.36	Low
6	81	936.93	Low
7	82	600.00	Low
8	83	2154.42	High

A SELECT statement that uses the CHOOSE function

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
       CHOOSE(TermsID, '10 days', '20 days', '30 days', '60 days', '90 days')
             AS NetDue
  FROM Invoices
 WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
```

	InvoiceNumber	InvoiceDate	InvoiceTotal	NetDue
1	39104	2020-01-10	85.31	30 days
2	963253264	2020-01-18	52.25	30 days
3	31361833	2020-01-21	579.42	20 days
4	263253268	2020-01-21	59.97	30 days
5	263253270	2020-01-22	67.92	30 days

Description

- The IIF and CHOOSE functions are known as *logical functions*, and they were introduced with SQL Server 2012.
- The IIF function lets you test an expression and return one value if the expression is true and another value if the expression is false. It provides a shorthand way of coding a searched CASE function with a single WHEN clause and an ELSE clause.
- The CHOOSE function provides an index into a list of values. The index value must be a type that can be converted to an int value and it must range from 1 to the number of values in the list.

Figure 9-11 How to use the IIF and CHOOSE functions

How to use the COALESCE and ISNULL functions

Figure 9-12 presents two functions that you can use to work with null values: COALESCE and ISNULL. Both of these functions let you substitute non-null values for null values. Although these two functions are similar, COALESCE is more flexible because it lets you specify a list of values. Then, it returns the first non-null value in the list. By contrast, the ISNULL function uses only two expressions. It returns the first expression if that expression isn't null. Otherwise, it returns the second expression.

The examples in this figure illustrate how these functions work. The first example uses the COALESCE function to return the value of the PaymentDate column, if that column doesn't contain a null value. Otherwise, it returns the date 1900-01-01. The second example performs the operation using the ISNULL function. Note that when you use either of these functions, all of the expressions must have the same data type. So, for example, you couldn't substitute the string "Not Paid" for a null payment date.

The third example shows how you can work around this restriction. In this example, the value of the InvoiceTotal column is converted to a character value. That way, if the InvoiceTotal column contains a null value, the COALESCE function can substitute the string "No invoices" for this value. Notice that this example uses an outer join to combine all of the rows in the Vendors table with the rows for each vendor in the Invoices table. Because of that, a null value will be returned for the InvoiceTotal column for any vendor that doesn't have invoices. As you can see, then, this function is quite useful with outer joins.

The syntax of the COALESCE function

```
COALESCE(expression_1 [, expression_2]...)
```

The syntax of the ISNULL function

```
ISNULL(check_expression, replacement_value)
```

A SELECT statement that uses the COALESCE function

```
SELECT PaymentDate,
       COALESCE(PaymentDate, '1900-01-01') AS NewDate
  FROM Invoices;
```

The same SELECT statement using the ISNULL function

```
SELECT PaymentDate,
       ISNULL(PaymentDate, '1900-01-01') AS NewDate
  FROM Invoices;
```

The result set

	PaymentDate	NewDate
111	2020-03-03	2020-03-03
112	NULL	1900-01-01
113	NULL	1900-01-01
114	2020-03-04	2020-03-04

A SELECT statement that substitutes a different data type

```
SELECT VendorName,
       COALESCE(CAST(InvoiceTotal AS varchar), 'No invoices') AS InvoiceTotal
  FROM Vendors LEFT JOIN Invoices
    ON Vendors.VendorID = Invoices.VendorID
 ORDER BY VendorName;
```

	VendorName	InvoiceTotal
1	Abbey Office Furnishings	17.50
2	American Booksellers Assoc	No invoices
3	American Express	No invoices
4	ASC Signs	No invoices
5	Ascom Hasler Mailing Systems	No invoices

Description

- The COALESCE and ISNULL functions let you substitute non-null values for null values.
- The COALESCE function returns the first expression in a list of expressions that isn't null. All of the expressions in the list must have the same data type. If all of the expressions are null, this function returns a null value.
- The ISNULL function returns the expression if it isn't null. Otherwise, it returns the value you specify. The expression and the value must have the same data type.
- COALESCE is not an ANSI-standard function, but it's more widely supported than ISNULL, which is unique to SQL Server.

Figure 9-12 How to use the COALESCE and ISNULL functions

How to use the GROUPING function

In chapter 5, you learned how to use the ROLLUP and CUBE operators to add summary rows to a summary query. You may recall that when you do that, a null value is assigned to any column in a summary row that isn't being summarized. If you need to, you can refer back to figures 5-7 and 5-8 to refresh your memory on how this works.

If you want to assign a value other than null to these columns, you can do that using the GROUPING function as illustrated in figure 9-13. This function accepts the name of a column as its argument. The column you specify must be one of the columns named in a GROUP BY clause that includes the ROLLUP or CUBE operator.

The example in this figure shows how you can use the GROUPING function in a summary query that summarizes vendors by state and city. This is the same summary query you saw back in figure 5-7. Instead of simply retrieving the values of the VendorState and VendorCity columns from the base table, however, this query uses the GROUPING function within a CASE function to determine the values that are assigned to those columns. If a row is added to summarize the VendorState column, for example, the value of the GROUPING function for that column is 1. Then, the CASE function assigns the value "All" to that column. Otherwise, it retrieves the value of the column from the Vendors table. Similarly, if a row is added to summarize the VendorCity column, the value "All" is assigned to that column. As you can see in the result set shown here, this makes it more obvious what columns are being summarized.

This technique is particularly useful if the columns you're summarizing can contain null values. In that case, it would be difficult to determine which rows are summary rows and which rows simply contain null values. Then, you may not only want to use the GROUPING function to replace the null values in summary rows, but you may want to use the COALESCE or ISNULL function to replace null values retrieved from the base table.

The syntax of the GROUPING function

```
GROUPING(column_name)
```

A summary query that uses the GROUPING function

```

SELECT
    CASE
        WHEN GROUPING(VendorState) = 1 THEN 'All'
        ELSE VendorState
    END AS VendorState,
    CASE
        WHEN GROUPING(VendorCity) = 1 THEN 'All'
        ELSE VendorCity
    END AS VendorCity,
    COUNT(*) AS QtyVendors
FROM Vendors
WHERE VendorState IN ('IA', 'NJ')
GROUP BY VendorState, VendorCity WITH ROLLUP
ORDER BY VendorState DESC, VendorCity DESC;
```

The result set

	VendorState	VendorCity	QtyVendors
1	NJ	Washington	1
2	NJ	Fairfield	1
3	NJ	East Brunswick	2
4	NJ	All	4
5	IA	Washington	1
6	IA	Fairfield	1
7	IA	All	2
8	All	All	6

Description

- You can use the GROUPING function to determine when a null value is assigned to a column as the result of the ROLLUP or CUBE operator. The column you name in this function must be one of the columns named in the GROUP BY clause.
- If a null value is assigned to the specified column as the result of the ROLLUP or CUBE operator, the GROUPING function returns a value of 1. Otherwise, it returns a value of 0.
- You typically use the GROUPING function with the CASE function. Then, if the GROUPING function returns a value of 1, you can assign a value other than null to the column.

Figure 9-13 How to use the GROUPING function

How to use the ranking functions

Figure 9-14 shows how to use the four *ranking functions*. These functions provide a variety of ways that you can rank the rows that are returned by a result set. All four of these functions have a similar syntax and work similarly.

The first example shows how to use the ROW_NUMBER function. Here, the SELECT statement retrieves two columns from the Vendors table. The first column uses the ROW_NUMBER function to sort the result set by VendorName and to number each row in the result set. To show that the first column has been sorted and numbered correctly, the second column displays the VendorName.

To accomplish the sorting and numbering, you code the name of the ROW_NUMBER function, followed by a set of parentheses, followed by the OVER keyword and a second set of parentheses. Within the second set of parentheses, you code the required ORDER BY clause that specifies the sort order that's used by the function. In this example, for instance, the ORDER BY clause sorts by VendorName in ascending order. However, you can code more complex ORDER BY clauses whenever that's necessary. In addition, when necessary, you can code an ORDER BY clause that applies to the entire result set. In that case, the ORDER BY clause within the ranking function is used to number the rows and the ORDER BY clause outside the ranking function is used to sort the rows after the numbering has been applied.

The second example shows how to use the optional PARTITION BY clause of a ranking function. This clause allows you to specify a column that's used to divide the result set into groups. In this example, for instance, the PARTITION BY clause uses a column within the Vendors table to group vendors by state and to sort these vendors by name within each state.

However, you can also use the PARTITION BY clause when a SELECT statement joins one or more tables like this:

```
SELECT VendorName, InvoiceNumber,
       ROW_NUMBER() OVER(PARTITION BY VendorName
                         ORDER BY InvoiceNumber) As RowNumber
  FROM Vendors JOIN Invoices
    ON Vendors.VendorID = Invoices.VendorID;
```

Here, the invoices will be grouped by vendor and sorted within each vendor by invoice number. As a result, if a vendor has three invoices, these invoices will be sorted by invoice number and numbered from 1 to 3.

The syntax for the four ranking functions

<code>ROW_NUMBER()</code>	<code>OVER ([partition_by_clause] order_by_clause)</code>
<code>RANK()</code>	<code>OVER ([partition_by_clause] order_by_clause)</code>
<code>DENSE_RANK()</code>	<code>OVER ([partition_by_clause] order_by_clause)</code>
<code>NTILE(integer_expression)</code>	<code>OVER ([partition_by_clause] order_by_clause)</code>

A query that uses the ROW_NUMBER function

```
SELECT ROW_NUMBER() OVER(ORDER BY VendorName) AS RowNumber, VendorName
FROM Vendors;
```

The result set

RowNumber	VendorName
1	Abbey Office Furnishings
2	American Booksellers Assoc
3	American Express
4	ASC Signs
5	Ascom Hasler Mailing Systems

A query that uses the PARTITION BY clause

```
SELECT ROW_NUMBER() OVER(PARTITION BY VendorState
                        ORDER BY VendorName) AS RowNumber, VendorName, VendorState
FROM Vendors;
```

The result set

RowNumber	VendorName	VendorState
1	AT&T	AZ
2	Computer Library	AZ
3	Wells Fargo Bank	AZ
4	Abbey Office Furnishings	CA
5	American Express	CA
6	ASC Signs	CA

Description

- The ROW_NUMBER, RANK, DENSE_RANK, and NTILE functions are known as *ranking functions*.
- The ROW_NUMBER function returns the sequential number of a row within a partition of a result set, starting at 1 for the first row in each partition.
- The ORDER BY clause of a ranking function specifies the sort order in which the ranking function is applied.
- The optional PARTITION BY clause of a ranking function specifies the column that's used to divide the result set into groups.

Figure 9-14 How to use the ranking functions (part 1 of 2)

The third example shows how the RANK and DENSE_RANK functions work. You can use these functions to rank the rows in a result set. In this example, both the RANK and the DENSE_RANK functions sort all invoices in the Invoices table by the invoice total. Since the first three rows have the same invoice total, both of these functions give these three rows the same rank, 1. However, the fourth row has a different value. To calculate the value for this row, the RANK function adds 1 to the total number of previous rows. In other words, since the first three rows are tied for first place, the fourth row gets fourth place and is assigned a rank of 4.

The DENSE_RANK function, on the other hand, calculates the value for the fourth row by adding 1 to the rank for the previous row. As a result, this function assigns a rank of 2 to the fourth row. In other words, since the first three rows are tied for first place, the fourth row gets second place.

The fourth example shows how the NTILE function works. You can use this function to divide the rows in a partition into the specified number of groups. When the rows can be evenly divided into groups, this function is easy to understand. For example, if a result set returns 100 rows, you can use the NTILE function to divide this result set into 10 groups of 10. However, when the rows can't be evenly divided into groups, this function is a little more difficult to understand. In this figure, for example, the NTILE function is used to divide a result set that contains 5 rows. Here, the first NTILE function divides this result into 2 groups with the first having 3 rows and the second having 2 rows. The second NTILE function divides this result set into 3 groups with the first having 2 rows, the second having 2 rows, and the third having 1 row. And so on. Although this doesn't result in groups with even numbers of rows, the NTILE function creates the number of groups specified by its argument.

In this figure, the examples for the RANK, DENSE_RANK, and NTILE functions don't include PARTITION BY clauses. As a result, these functions are applied to the entire result set. However, whenever necessary, you can use the PARTITION BY clause to divide the result set into groups just as shown in the second example for the ROW_NUMBER function.

A query that uses the RANK and DENSE_RANK functions

```
SELECT RANK() OVER (ORDER BY InvoiceTotal) As Rank,
       DENSE_RANK() OVER (ORDER BY InvoiceTotal) As DenseRank,
       InvoiceTotal, InvoiceNumber
  FROM Invoices;
```

The result set

	Rank	DenseRank	InvoiceTotal	InvoiceNumber
1	1	1	6.00	25022117
2	1	1	6.00	24863706
3	1	1	6.00	24780512
4	4	2	9.95	21-4923721
5	4	2	9.95	21-4748363
6	6	3	10.00	4-321-2596

Description

- The RANK and DENSE_RANK functions both return the rank of each row within the partition of a result set.
- If there is a tie, both of these functions give the same rank to all rows that are tied.
- To determine the rank for the next distinct row, the RANK function adds 1 to the total number of rows, while the DENSE_RANK function adds 1 to the rank for the previous row.

A query that uses the NTILE function

```
SELECT TermsDescription,
       NTILE(2) OVER (ORDER BY TermsID) AS Tile2,
       NTILE(3) OVER (ORDER BY TermsID) AS Tile3,
       NTILE(4) OVER (ORDER BY TermsID) AS Tile4
  FROM Terms;
```

The result set

	TermsDescription	Tile2	Tile3	Tile4
1	Net due 10 days	1	1	1
2	Net due 20 days	1	1	1
3	Net due 30 days	1	2	2
4	Net due 60 days	2	2	3
5	Net due 90 days	2	3	4

Description

- The NTILE function divides the rows in a partition into the specified number of groups.
- If the rows can't be evenly divided into groups, the later groups may have one less row than the earlier groups.

Figure 9-14 How to use the ranking functions (part 2 of 2)

How to use the analytic functions

Figure 9-15 shows how to use the *analytic functions* that were introduced with SQL Server 2012. These functions let you perform calculations on ordered sets of data. Note that all of the examples in this figure use the SalesReps and SalesTotals tables that are summarized in this figure. These tables are related by the RepID column in each table.

The FIRST_VALUE and LAST_VALUE functions let you return the first and last values in an ordered set of values. The first example in this figure uses these functions to return the name of the sales rep with the highest and lowest sales for each year. To do that, the OVER clause is used to group the result set by year and sort the rows within each year by sales total in descending sequence. Then, the expression that's specified for the functions causes the name for the first rep within each year to be returned.

For the LAST_VALUE function to return the value you want, you also have to include the RANGE clause as shown here. This clause indicates that the rows should be unbounded within the partition. In other words, all of the rows in the partition should be included in the calculation. If you don't include this clause, the LAST_VALUE function will return the last value for each group specified by the ORDER BY clause. In this case, that means that the function would return the last rep name for each sales total. Since all of the sales totals are different, though, the function would simply return the name of the rep in each row, which isn't what you want. So, you would typically use this clause only if you sorted the result set by a column that contains duplicate values. In that case, you can typically omit the PARTITION BY clause.

Instead of the RANGE clause, you can code a ROWS clause on a FIRST_VALUE or LAST_VALUE function. This clause lets you specify the rows to include relative to the current row. For more information on how to code this clause and the RANGE clause, please search for it in the SQL Server documentation.

The syntax of the analytic functions

```
{FIRST_VALUE|LAST_VALUE}(scalar_expression)
    OVER ([partition_by_clause] order_by_clause [rows_range_clause])

{LEAD|LAG}(scalar_expression [, offset [, default]])
    OVER ([partition_by_clause] order_by_clause)

{PERCENT_RANK()|CUME_DIST} OVER ([partition_by_clause] order_by_clause)

{PERCENTILE_CONT|PERCENTILE_DISC}(numeric_literal)
    WITHIN GROUP (ORDER BY expression [ASC|DESC]) OVER (partition_by_clause)
```

The columns in the SalesReps and SalesTotals tables

Column name	Data type	Column name	Data type
RepID	int	RepID	int
RepFirstName	varchar(50)	SalesYear	char(4)
RepLastName	varchar(50)	SalesTotal	money

A query that uses the FIRST_VALUE and LAST_VALUE functions

```
SELECT SalesYear, RepFirstName + ' ' + RepLastName AS RepName, SalesTotal,
FIRST_VALUE(RepFirstName + ' ' + RepLastName)
    OVER (PARTITION BY SalesYear ORDER BY SalesTotal DESC)
    AS HighestSales,
LAST_VALUE(RepFirstName + ' ' + RepLastName)
    OVER (PARTITION BY SalesYear ORDER BY SalesTotal DESC
        RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
    AS LowestSales
FROM SalesTotals JOIN SalesReps
    ON SalesTotals.RepID = SalesReps.RepID;
```

SalesYear	RepID	SalesTotal	PctRank	CumeDist	PercentileCont	PercentileDisc
1 2017	2	978465.99	0	0.33333333333333	1032875.48	1032875.48
2 2017	3	1032875.48	0.5	0.666666666666667	1032875.48	1032875.48
3 2017	1	1274856.38	1	1	1032875.48	1032875.48
4 2018	5	422847.86	0	0.2	923746.85	923746.85
5 2018	4	655786.92	0.25	0.4	923746.85	923746.85
6 2018	1	923746.85	0.5	0.6	923746.85	923746.85
7 2018	2	974853.81	0.75	0.8	923746.85	923746.85
8 2018	3	1132744.56	1	1	923746.85	923746.85
9 2019	5	45182.44	0	0.25	480069.56	72443.37
10 2019	4	72443.37	0.3333...	0.5	480069.56	72443.37
11 2019	2	887695.75	0.6666...	0.75	480069.56	72443.37
12 2019	1	998337.46	1	1	480069.56	72443.37

Description

- The FIRST_VALUE, LAST_VALUE, LEAD, LAG, PERCENT_RANK, CUME_DIST, PERCENTILE_CONT, and PERCENTILE_DISC functions are known as *analytic functions*. They were introduced with SQL Server 2012.
- The FIRST_VALUE function returns the first value in a sorted set of values, and the LAST_VALUE function returns the last value in a sorted set of values. When you use the PARTITION BY clause with LAST_VALUE, you typically include the ROWS or RANGE clause as well.

Figure 9-15 How to use the analytic functions (part 1 of 2)

The LEAD and LAG functions let you refer to values in other rows of the result set. The LAG function is illustrated in the second example in this figure. Here, the OVER clause is used to group the result set by rep ID and sort it by year. Then, the LAG function in the fourth column gets the sales total from one row prior to the current row (the offset). Since the rows are sorted by year for each sales rep, that means that the function retrieves the sales rep's sales for the previous year. The fifth column uses the same function, but it subtracts the result of this function from the current sales to show the change in sales from the previous year. The LEAD function is similar, but it lets you refer to values in following rows rather than preceding rows.

Notice that the value of the LAG function for the first row for each sales rep is 0.00. That's because there isn't a row for the prior year. By default, this value is NULL. Because I wanted to calculate the change for each row in the result set, though, I set the third argument of the LAG function (default) to 0.

The third example in this figure shows how to use all four of the remaining functions. Each of these functions groups the rows by year and sorts them by sales total in ascending sequence. Notice, though, that the ORDER BY clause for the PERCENTILE_CONT and PERCENTILE_DISC functions isn't specified on the OVER clause. Instead, it's specified on the WITHIN GROUP clause, which, unlike the OVER clause, allows for the result set to be sorted only by a single column.

The PERCENT_RANK function calculates a percent that indicates the rank of each row within a group. The result of this function is always a value between 0 and 1. If you study the results in this example, you shouldn't have any trouble understanding how this function works.

The CUME_DIST function is similar, but it calculates the percent of values that are less than or equal to the current value. This function represents the *cumulative distribution* of the values. The cumulative distribution is calculated by dividing the number of rows with the current value or a lower value by the total number of rows in the group.

The PERCENTILE_CONT and PERCENTILE_DISC functions calculate the value at the percentile you specify. The difference between these two functions is that PERCENTILE_CONT is based on a *continuous distribution* of values, and PERCENTILE_DISC is based on a *discrete distribution* of values. This means that the value returned by PERCENTILE_CONT doesn't need to appear in the result set, but the value returned by PERCENTILE_DISC does.

In this example, these functions are used to calculate the median of the sales totals for each year (the value in the middle). Because there are an odd number of rows for 2017 and 2018, both functions return the value in the middle row. Because there are an even number of rows for 2019, though, there isn't a middle value. In that case, the PERCENTILE_CONT function calculates the median by adding the two middle values together and dividing by 2. As you can see, the resulting value doesn't exist in the result set. By contrast, the PERCENTILE_DISC function uses the CUME_DIST function to identify the row with a cumulative distribution of .5 (the same percentile specified by the PERCENTILE_DISC function), and it uses the value of that row as the result.

A query that uses the LAG function

```
SELECT RepID, SalesYear, SalesTotal AS CurrentSales,
       LAG(SalesTotal, 1, 0) OVER (PARTITION BY RepID ORDER BY SalesYear)
           AS LastSales,
       SalesTotal - LAG(SalesTotal, 1, 0)
           OVER (PARTITION BY RepID ORDER BY SalesYear) AS Change
FROM SalesTotals;
```

RepID	SalesYear	CurrentSales	LastSales	Change
1	2017	1274856.38	0.00	1274856.38
2	2018	923746.85	1274856.38	-351109.53
3	2019	998337.46	923746.85	74590.61
4	2017	978465.99	0.00	978465.99
5	2018	974853.81	978465.99	-3612.18
6	2019	887695.75	974853.81	-87158.06

A query that uses the PERCENT_RANK, CUME_DIST, PERCENTILE_CONT, and PERCENTILE_DISC functions

```
SELECT SalesYear, RepID, SalesTotal,
       PERCENT_RANK() OVER (PARTITION BY SalesYear ORDER BY SalesTotal)
           AS PctRank,
       CUME_DIST() OVER (PARTITION BY SalesYear ORDER BY SalesTotal)
           AS CumeDist,
       PERCENTILE_CONT(.5) WITHIN GROUP (ORDER BY SalesTotal)
           OVER (PARTITION BY SalesYear) AS PercentileCont,
       PERCENTILE_DISC(.5) WITHIN GROUP (ORDER BY SalesTotal)
           OVER (PARTITION BY SalesYear) AS PercentileDisc
FROM SalesTotals;
```

SalesYear	RepName	SalesTotal	HighestSales	LowestSales
2017	Jonathon Thomas	1274856.38	Jonathon Thomas	Sonja Martinez
2017	Andrew Markasian	1032875.48	Jonathon Thomas	Sonja Martinez
2017	Sonja Martinez	978465.99	Jonathon Thomas	Sonja Martinez
2018	Andrew Markasian	1132744.56	Andrew Markasian	Lydia Kramer
2018	Sonja Martinez	974853.81	Andrew Markasian	Lydia Kramer
2018	Jonathon Thomas	923746.85	Andrew Markasian	Lydia Kramer
2018	Phillip Winters	655786.92	Andrew Markasian	Lydia Kramer
2018	Lydia Kramer	422847.86	Andrew Markasian	Lydia Kramer
2019	Jonathon Thomas	998337.46	Jonathon Thomas	Lydia Kramer
2019	Sonja Martinez	887695.75	Jonathon Thomas	Lydia Kramer
2019	Phillip Winters	72443.37	Jonathon Thomas	Lydia Kramer
2019	Lydia Kramer	45182.44	Jonathon Thomas	Lydia Kramer

Description

- The LEAD function retrieves data from a subsequent row in a result set, and the LAG function retrieves data from a previous row in a result set.
- The PERCENT_RANK function calculates the rank of the values in a sorted set of values as a percent. The CUME_DIST function calculates the percent of the values in a sorted set of values that are less than or equal to the current value.
- The PERCENTILE_CONT and PERCENTILE_DISC functions calculate the value at the specified percentile for a sorted set of values. PERCENTILE_CONT returns an exact percentile, and PERCENTILE_DIST returns a value that exists in the sorted column.

Figure 9-15 How to use the analytic functions (part 2 of 2)

Perspective

In this chapter, you learned about many of the functions that you can use to operate on SQL Server data. At this point, you have all of the essential skills you need to develop SQL code at a professional level.

However, there's a lot more to learn about SQL Server. In the next section of this book, then, you'll learn the basic skills for designing a database. Even if you never need to design your own database, understanding this material will help you work more efficiently with databases that have been designed by others.

Terms

logical functions
ranking functions
analytic functions

cumulative distribution
continuous distribution
discrete distribution

Exercises

1. Write a SELECT statement that returns two columns based on the Vendors table. The first column, Contact, is the vendor contact name in this format: first name followed by last initial (for example, “John S.”) The second column, Phone, is the VendorPhone column without the area code. Only return rows for those vendors in the 559 area code. Sort the result set by first name, then last name.
2. Write a SELECT statement that returns the InvoiceNumber and balance due for every invoice with a non-zero balance and an InvoiceDueDate that’s less than 30 days from today.
3. Modify the search expression for InvoiceDueDate from the solution for exercise 2. Rather than 30 days from today, return invoices due before the last day of the current month.
4. Write a summary query that uses the CUBE operator to return LineItemSum (which is the sum of InvoiceLineItemAmount) grouped by Account (an alias for AccountDescription) and State (an alias for VendorState). Use the CASE and GROUPING function to substitute the literal value “*ALL*” for the summary rows with null values.
5. Add a column to the query described in exercise 2 that uses the RANK() function to return a column named BalanceRank that ranks the balance due in descending order.

Section 3

Database design and implementation

In large programming shops, database administrators are usually responsible for designing the databases that are used by production applications, and they may also be responsible for the databases that are used for testing those applications. Often, though, programmers are asked to design, create, or maintain small databases that are used for testing. And in a small shop, programmers may also be responsible for the production databases.

So whether you're a database administrator or a SQL programmer, you need the skills and knowledge presented in this section. That's true even if you aren't ever called upon to design or maintain a database. By understanding what's going on behind the scenes, you'll be able to use SQL more effectively.

So, in chapter 10, you'll learn how to design a SQL Server database. In chapter 11, you'll learn how to use the Data Definition Language (DDL) statements to create and maintain the SQL Server objects of a database. And in chapter 12, you'll learn how to use the Management Studio to do the same tasks.

10

How to design a database

In this chapter, you'll learn how to design a new database. This is useful information for the SQL programmer whether or not you ever design a database on your own. To illustrate this process, I'll use the accounts payable (AP) system that you've seen throughout this book because that will make it easier for you to understand the design techniques.

How to design a data structure.....	304
The basic steps for designing a data structure.....	304
How to identify the data elements	306
How to subdivide the data elements	308
How to identify the tables and assign columns	310
How to identify the primary and foreign keys	312
How to enforce the relationships between tables	314
How normalization works	316
How to identify the columns to be indexed.....	318
How to normalize a data structure.....	320
The seven normal forms	320
How to apply the first normal form	322
How to apply the second normal form	324
How to apply the third normal form.....	326
When and how to denormalize a data structure.....	328
Perspective	330

How to design a data structure

Databases are often designed by database administrators (DBAs) or design specialists. This is especially true for large, multiuser databases. How well this is done can directly affect your job as a SQL programmer. In general, a well designed database is easy to query, while a poorly designed database is difficult to work with. In fact, when you work with a poorly designed database, you will often need to figure out how it is designed before you can code your queries appropriately.

The topics that follow will teach you a basic approach for designing a *data structure*. We use that term to refer to a model of the database rather than the database itself. Once you design the data structure, you can use the techniques presented in the next two chapters to create a database with that design.

The basic steps for designing a data structure

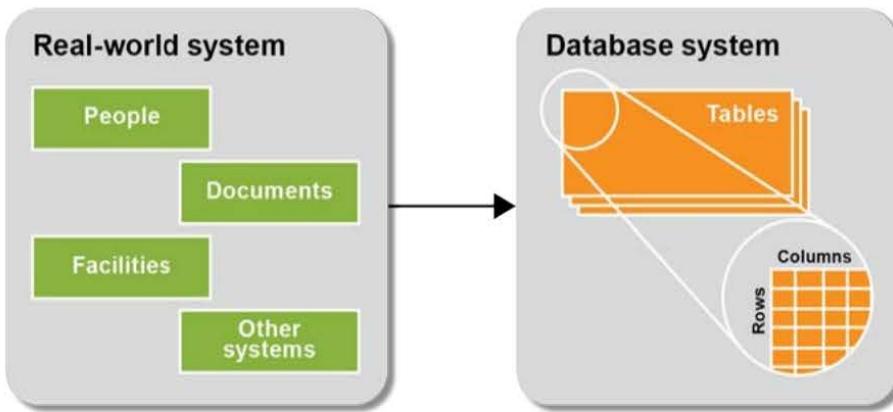
In many cases, you can design a data structure based on an existing real-world system. The illustration at the top of figure 10-1 presents a conceptual view of how this works. Here, you can see that all of the information about the people, documents, and facilities within a real-world system is mapped to the tables, columns, and rows of a database system.

As you design a data structure, each table represents one object, or *entity*, in the real-world system. Then, within each table, each column stores one item of information, or *attribute*, for the entity, and each row stores one occurrence, or *instance*, of the entity.

This figure also presents the six steps you can follow to design a data structure. You'll learn more about each of these steps in the topics that follow. In general, though, step 1 is to identify all the data elements that need to be stored in the database. Step 2 is to break complex elements down into smaller components whenever that makes sense. Step 3 is to identify the tables that will make up the system and to determine which data elements are assigned as columns in each table. Step 4 is to define the relationships between the tables by identifying the primary and foreign keys. Step 5 is to normalize the database to reduce data redundancy. And step 6 is to identify the indexes that are needed for each table.

To model a database system after a real-world system, you can use a technique called *entity-relationship (ER) modeling*. Because this is a complex subject of its own, I won't present it in this book. However, I have applied some of the basic elements of this technique to the design diagrams presented in this chapter. In effect, then, you'll be learning some of the basics of this modeling technique.

A database system is modeled after a real-world system



The six basic steps for designing a data structure

- Step 1: Identify the data elements
- Step 2: Subdivide each element into its smallest useful components
- Step 3: Identify the tables and assign columns
- Step 4: Identify the primary and foreign keys
- Step 5: Review whether the data structure is normalized
- Step 6: Identify the indexes

Description

- A relational database system should model the real-world environment where it's used. The job of the designer is to analyze the real-world system and then map it onto a relational database system.
- A table in a relational database typically represents an object, or *entity*, in the real world. Each column of a table is used to store an *attribute* associated with the entity, and each row represents one *instance* of the entity.
- To model a database and the relationships between its tables after a real-world system, you can use a technique called *entity-relationship (ER) modeling*. Some of the diagrams you'll see in this chapter apply the basic elements of ER modeling.

Figure 10-1 The basic steps for designing a data structure

How to identify the data elements

The first step for designing a data structure is to identify the data elements required by the system. You can use several techniques to do that, including analyzing the existing system if there is one, evaluating comparable systems, and interviewing anyone who will be using the system. One particularly good source of information are the documents used by an existing system.

In figure 10-2, for example, you can see an invoice that's used by an accounts payable system. We'll use this document as the main source of information for the database design presented in this chapter. Keep in mind, though, that you'll want to use all available resources when you design your own database.

If you study this document, you'll notice that it contains information about three different entities: vendors, invoices, and line items. First, the form itself has preprinted information about the vendor who issued the invoice, such as the vendor's name and address. If this vendor were to issue another invoice, this information wouldn't change.

This document also contains specific information about the invoice. Some of this information, such as the invoice number, invoice date, and invoice total, is general in nature. Although the actual information will vary from one invoice to the next, each invoice will include this information. In addition to this general information, each invoice includes information about the items that were purchased. Although each line item contains similar information, each invoice can contain a different number of line items.

One of the things you need to consider as you review a document like this is how much information your system needs to track. For an accounts payable system, for example, you may not need to store detailed data such as the information about each line item. Instead, you may just need to store summary data like the invoice total. As you think about what data elements to include in the database, then, you should have an idea of what information you'll need to get back out of the system.

An invoice that can be used to identify data elements

Acme Fabrication, Inc.				
<i>Custom Contraptions, Contrivances and Confabulations</i>		Invoice Number: 101-1088		
1234 West Industrial Way East Los Angeles California 90022		Invoice Date: 04/05/20		
800.555.1212 fax 562.555.1213 www.acmefabrication.com		Terms: Net 30		
<hr/>				
Part No.	Qty.	Description	Unit Price	Extension
CUST345	12	Design service, hr	100.00	1200.00
457332	7	Baling wire, 25x3ft roll	79.90	559.30
50173	4375	Duct tape, black, yd	1.09	4768.75
328771	2	Rubber tubing, 100ft roll	4.79	9.58
CUST281	7	Assembly, hr	75.00	525.00
CUST917	2	Testing, hr	125.00	250.00
		Sales Tax		245.20
<hr/>				
Your salesperson:	Ruben Goldberg, ext 4512		\$7,557.83	
Accounts receivable:	Inigo Jones, ext 4901		PLEASE PAY THIS AMOUNT	
<i>Thanks for your business!</i>				

The data elements identified on the invoice document

Vendor name	Invoice date	Item extension
Vendor address	Invoice terms	Vendor sales contact name
Vendor phone number	Item part number	Vendor sales contact extension
Vendor fax number	Item quantity	Vendor AR contact name
Vendor web address	Item description	Vendor AR contact extension
Invoice number	Item unit price	Invoice total

Description

- Depending on the nature of the system, you can identify data elements in a variety of ways, including interviewing users, analyzing existing systems, and evaluating comparable systems.
- The documents used by a real-world system, such as the invoice shown above, can often help you identify the data elements of the system.
- As you identify the data elements of a system, you should begin thinking about the entities that those elements are associated with. That will help you identify the tables of the database later on.

Figure 10-2 How to identify the data elements

How to subdivide the data elements

Some of the data elements you identify in step 1 of the design procedure will consist of multiple components. The next step, then, is to divide these elements into their smallest useful values. Figure 10-3 shows how you can do that.

The first example in this figure shows how you can divide the name of the sales contact for a vendor. Here, the name is divided into two elements: a first name and a last name. When you divide a name like this, you can easily perform operations like sorting by last name and using the first name in a salutation, such as “Dear Ruben.” By contrast, if the full name is stored in a single column, you have to use the string functions to extract the component you need. And, as you learned in the last chapter, that can lead to inefficient and complicated code. In general, then, you should separate a name like this whenever you’ll need to use the name components separately. Later, when you need to use the full name, you can combine the first and last names using concatenation.

The second example shows how you typically divide an address. Notice in this example that the street number and street name are stored in a single column. Although you could store these components in separate columns, that usually doesn’t make sense since these values are typically used together. That’s what I mean when I say the data elements should be divided into their smallest *useful* values.

With that guideline in mind, you might even need to divide a single string into two or more components. A bulk mail system, for example, might require a separate column for the first three digits of the zip code. And a telephone number might require two columns: one for the area code and another for the rest of the number. Historically, the area code was a useful value as it provided information about the geographical location of the phone. However, now that so many phone numbers are mobile numbers, this value has become less useful.

As in the previous step, knowledge of the real-world system and of the information that will be extracted from the database is critical. In some circumstances, it may be okay to store data elements with multiple components in a single column. That can simplify your design and reduce the overall number of columns. In general, though, most designers divide data elements as much as possible. That way, it’s easy to accommodate almost any query, and you don’t have to change the database design later on when you realize that you need to use just part of a column value.

A name that's divided into first and last names



An address that's divided into street address, city, state, and zip code



Description

- If a data element contains two or more components, you should consider subdividing the element into those components. That way, you won't need to parse the element each time you use it.
- The extent to which you subdivide a data element depends on how it will be used. Because it's difficult to predict all future uses for the data, most designers subdivide data elements as much as possible.
- When you subdivide a data element, you can easily rebuild it when necessary by concatenating the individual components.

Figure 10-3 How to subdivide the data elements

How to identify the tables and assign columns

Figure 10-4 presents the three main entities for the accounts payable system and lists the possible data elements that can be associated with each one. In most cases, you'll recognize the main entities that need to be included in a data structure as you identify the data elements. As I reviewed the data elements represented on the invoice document in figure 10-2, for example, I identified the three entities shown in this figure: vendors, invoices, and invoice line items. Although you may identify additional entities later on in the design process, it's sufficient to identify the main entities at this point. These entities will become the tables of the database.

After you identify the main entities, you need to determine which data elements are associated with each entity. These elements will become the columns of the tables. In many cases, the associations are obvious. For example, it's easy to determine that the vendor name and address are associated with the vendors entity and the invoice date and invoice total are associated with the invoices entity. Some associations, however, aren't so obvious. In that case, you may need to list a data element under two or more entities. In this figure, for example, you can see that the invoice number is included in both the invoices and invoice line items entities and the account number is included in all three entities. Later, when you normalize the data structure, you may be able to remove these repeated elements. For now, though, it's okay to include them.

Before I go on, I want to point out the notation I used in this figure. To start, any data elements I included that weren't identified in previous steps are shown in *italics*. Although you should be able to identify most of the data elements in the first two steps of the design process, you'll occasionally think of additional elements during the third step. In this case, since the initial list of data elements was based on a single document, I added several data elements to this list.

Similarly, you may decide during this step that you don't need some of the data elements you've identified. For example, I decided that I didn't need the fax number or web address of each vendor. So I used the strikethrough feature of my word processor to indicate that these data elements should not be included.

Finally, I identified the data elements that are included in two or more tables by coding an asterisk after them. Although you can use any notation you like for this step of the design process, you'll want to be sure that you document your design decisions. For a complicated design, you may even want to use a *CASE (computer-aided software engineering)* tool.

By the way, a couple of the new data elements I added may not be clear to you if you haven't worked with a corporate accounts payable system before. "Terms" refers to the payment terms that the vendor offers. For example, the terms might be net 30 (the invoice must be paid in 30 days) or might include a discount for early payment. "Account number" refers to the general ledger accounts that a company uses to track its expenses. For example, one account number might be assigned for advertising expenses, while another might be for office supplies. Each invoice that's paid is assigned to an account, and in some cases, different line items on an invoice are assigned to different accounts.

Possible tables and columns for an accounts payable system

Vendors	Invoices	Invoice line items
Vendor name	Invoice number*	Invoice number*
Vendor address	Invoice date	Item part number
Vendor city	Terms*	Item quantity
Vendor state	Invoice total	Item description
Vendor zip code	<i>Payment date</i>	Item unit price
Vendor phone number	<i>Payment total</i>	Item extension
Vendor fax number	<i>Invoice due date</i>	<i>Account number*</i>
Vendor web address	<i>Credit total</i>	<i>Sequence number</i>
Vendor contact first name		<i>Account number*</i>
Vendor contact last name		
Vendor contact phone		
Vendor AR-first name		
Vendor AR-last name		
Vendor AR-phone		
<i>Terms*</i>		
<i>Account number*</i>		

Description

- After you identify and subdivide all of the data elements for a database, you should group them by the entities with which they're associated. These entities will later become the tables of the database, and the elements will become the columns.
- If a data element relates to more than one entity, you can include it under all of the entities it relates to. Then, when you normalize the database, you may be able to remove the duplicate elements.
- As you assign the elements to entities, you should omit elements that aren't needed, and you should add any additional elements that are needed.

The notation used in this figure

- Data elements that were previously identified but aren't needed are crossed out.
- Data elements that were added are displayed in italics.
- Data elements that are related to two or more entities are followed by an asterisk.
- You can use a similar notation or develop one of your own. You can also use a CASE (computer-aided software engineering) tool if one is available to you.

Figure 10-4 How to identify the tables and assign columns

How to identify the primary and foreign keys

Once you identify the entities and data elements of a system, the next step is to identify the relationships between the tables. To do that, you need to identify the primary and foreign keys as shown in figure 10-5.

As you know, a primary key is used to uniquely identify each row in a table. In some cases, you can use an existing column as the primary key. For example, you might consider using the VendorName column as the primary key of the Vendors table. Because the values for this column can be long, however, and because it would be easy to enter a value incorrectly, that's not a good candidate. Instead, an identity column is used as the primary key.

Similarly, you might consider using the InvoiceNumber column as the primary key of the Invoices table. However, it's possible for different vendors to use the same invoice number, so this value isn't necessarily unique. Because of that, an identity column is used as the primary key of this table as well.

To uniquely identify the rows in the InvoiceLineItems table, this design uses a *composite key*. This composite key uses two columns to uniquely identify each row. The first column is the InvoiceID column from the Invoices table, and the second column is the InvoiceSequence column. This is necessary because this table may contain more than one row (line item) for each invoice. And that means that the InvoiceID value by itself won't be unique.

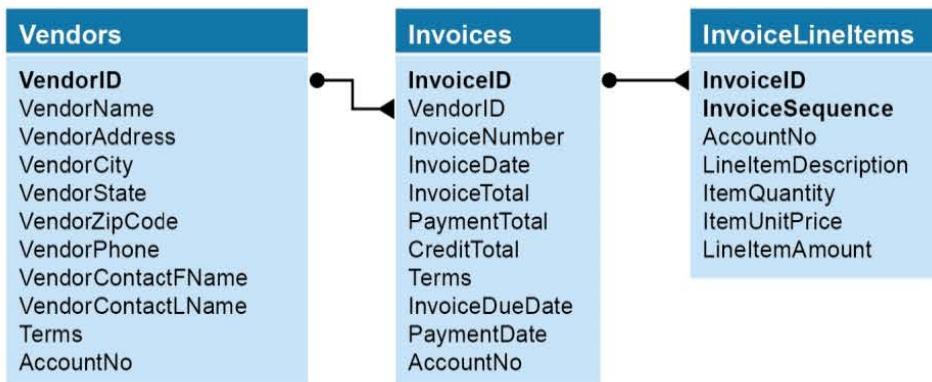
This book uses the composite key in the InvoiceLineItems table to show how to work with composite keys. However, it usually makes more sense to use a single column as the primary key. For example, the InvoiceLineItems table could start with an InvoiceLineItemID column that uniquely identifies each row in the table. Then, you could use that column as the primary key, and you could consider dropping the InvoiceSequence column.

After you identify the primary key of each table, you need to identify the relationships between the tables and add foreign key columns as necessary. In most cases, two tables will have a one-to-many relationship with each other. For example, each vendor can have many invoices, and each invoice can have many line items. To identify the vendor that each invoice is associated with, a VendorID column is included in the Invoices table. Because the InvoiceLineItems table already contains an InvoiceID column, it's not necessary to add another column to this table.

The diagram at the top of this figure illustrates the relationships I identified between the tables in the accounts payable system. As you can see, the primary keys are displayed in bold. Then, the lines between the tables indicate how the primary key in one table is related to the foreign key in another table. Here, a small, round connector indicates the "one" side of the relationship, and the triangular connector indicates the "many" side of the relationship.

In addition to the one-to-many relationships shown in this diagram, you can also use many-to-many relationships and one-to-one relationships. The second diagram in this figure, for example, shows a many-to-many relationship between an Employees table and a Committees table. As you can see, this type of relationship can be implemented by creating a *linking table*, also called a

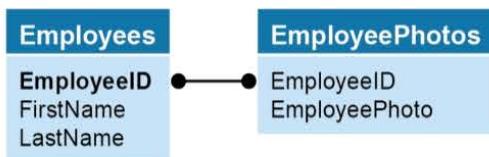
The relationships between the tables in the accounts payable system



Two tables with a many-to-many relationship



Two tables with a one-to-one relationship



Description

- Most tables should have a primary key that uniquely identifies each row. If necessary, you can use a *composite key* that uses two or more columns to uniquely identify each row.
- The values of the primary keys should seldom, if ever, change. The values should also be short and easy to enter correctly.
- If a suitable column doesn't exist, you can create an identity column that can be used as the primary key.
- If two tables have a one-to-many relationship, you may need to add a foreign key column to the table on the "many" side. The foreign key column must have the same data type as the primary key column it's related to.
- If two tables have a many-to-many relationship, you'll need to define a *linking table* to relate them. Then, each of the tables in the many-to-many relationship will have a one-to-many relationship with the linking table. The linking table doesn't usually have a primary key.
- If two tables have a one-to-one relationship, they should be related by their primary keys. This type of relationship is typically used to improve performance. Then, columns with large amounts of data can be stored in a separate table.

Figure 10-5 How to identify the primary and foreign keys

connecting table or an *associate table*. This table contains the primary key columns from the two tables. Then, each table has a one-to-many relationship with the linking table. Notice that the linking table doesn't have its own primary key. Because this table doesn't correspond to an entity and because it's used only in conjunction with the Employees and Committees tables, a primary key isn't needed.

The third example in figure 10-5 illustrates two tables that have a one-to-one relationship. With this type of relationship, both tables have the same primary key, which means that the information could be stored in a single table. This type of relationship is often used when a table contains one or more columns with large amounts of data. In this case, the EmployeePhotos table contains a large binary column with a photo of each employee. Because this column is used infrequently, storing it in a separate table will make operations on the Employees table more efficient. Then, when this column is needed, it can be combined with the columns in the Employees table using a join.

How to enforce the relationships between tables

Although the primary keys and foreign keys indicate how the tables in a database are related, SQL Server doesn't enforce those relationships automatically. Because of that, any of the operations shown in the table at the top of figure 10-6 would violate the *referential integrity* of the tables. If you deleted a row from a primary key table, for example, and the foreign key table included rows related to that primary key, the referential integrity of the two tables would be destroyed. In that case, the rows in the foreign key table that no longer have a related row in the primary key table would be *orphaned*. Similar problems can occur when you insert a row into the foreign key table or update a primary key or foreign key value.

To enforce those relationships and maintain the referential integrity of the tables, you can use one of two features provided by SQL Server: declarative referential integrity or triggers. To use *declarative referential integrity (DRI)*, you define *foreign key constraints* that indicate how the referential integrity between the tables is enforced. You'll learn more about defining foreign key constraints in the next two chapters. For now, just realize that these constraints can prevent all of the operations listed in this figure that violate referential integrity.

Operations that can violate referential integrity

This operation...	Violates referential integrity if...
Delete a row from the primary key table	The foreign key table contains one or more rows related to the deleted row
Insert a row in the foreign key table	The foreign key value doesn't have a matching primary key value in the related table
Update the value of a foreign key	The new foreign key value doesn't have a matching primary key value in the related table
Update the value of a primary key	The foreign key table contains one or more rows related to the row that's changed

Description

- *Referential integrity* means that the relationships between tables are maintained correctly. That means that a table with a foreign key doesn't have rows with foreign key values that don't have matching primary key values in the related table.
- In SQL Server, you can enforce referential integrity by using declarative referential integrity or by defining triggers.
- To use *declarative referential integrity (DRI)*, you define *foreign key constraints*. You'll learn how to do that in the next two chapters.
- When you define foreign key constraints, you can specify how referential integrity is enforced when a row is deleted from the primary key table. The options are to return an error or to delete the related rows in the foreign key table.
- You can also specify how referential integrity is enforced when the primary key of a row is changed and foreign key constraints are in effect. The options are to return an error or to change the foreign keys of all the related rows to the new value.
- If referential integrity isn't enforced and a row is deleted from the primary key table that has related rows in the foreign key table, the rows in the foreign key table are said to be *orphaned*.
- The three types of errors that can occur when referential integrity isn't enforced are called the *deletion anomaly*, the *insertion anomaly*, and the *update anomaly*.

Figure 10-6 How to enforce the relationships between tables

How normalization works

The next step in the design process is to review whether the data structure is *normalized*. To do that, you look at how the data is separated into related tables. If you follow the first four steps for designing a database that are presented in this chapter, your database will already be partially normalized when you get to this step. However, almost every design can be normalized further.

Figure 10-7 illustrates how *normalization* works. The first two tables in this figure show some of the problems caused by an *unnormalized* data structure. In the first table, you can see that each row represents an invoice. Because an invoice can have one or more line items, however, the ItemDescription column must be repeated to provide for the maximum number of line items. But since most invoices have fewer line items than the maximum, this can waste storage space.

In the second table, each line item is stored in a separate row. That eliminates the problem caused by repeating the ItemDescription column, but it introduces a new problem: the invoice number must be repeated in each row. This, too, can cause storage problems, particularly if the repeated column is large. In addition, it can cause maintenance problems if the column contains a value that's likely to change. Then, when the value changes, each row that contains the value must be updated. And if a repeated value must be reentered for each new row, it would be easy for the value to vary from one row to another.

To eliminate the problems caused by *data redundancy*, you can normalize the data structure. To do that, you apply the *normal forms* you'll learn about later in this chapter. As you'll see, there are a total of seven normal forms. However, it's common to apply only the first three.

The diagram in this figure, for example, shows the accounts payable system in third normal form. Here, the Terms table stores data that's needed by the Vendors and Invoices tables. Similarly, the GLAccounts table stores data that's needed by the Vendors and InvoiceLineItems tables. Storing terms and accounts data in one table instead of in multiple tables reduces data redundancy. At this point, it might not be clear to you how this works, but it should become clearer as you learn about the different normal forms.

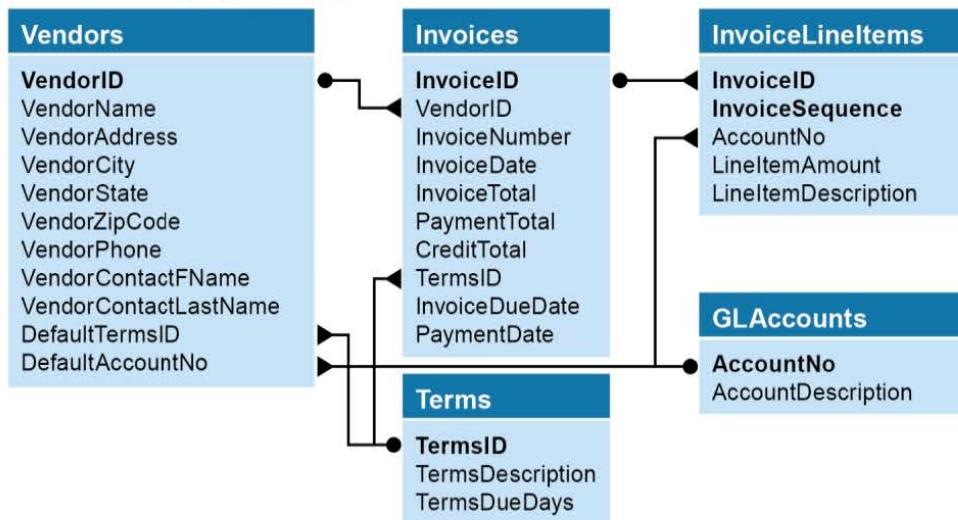
A table that contains repeating columns

	InvoiceNumber	ItemDescription1	ItemDescription2	ItemDescription3
1	112897	C# ad	SQL ad	Library directory
2	97/552	Catalogs	SQL flyer	NULL
3	97/533B	Card revision	NULL	NULL

A table that contains redundant data

	InvoiceNumber	ItemDescription
1	112897	C# ad
2	112897	SQL ad
3	112897	Library directory
4	97/522	Catalogs
5	97/522	SQL flyer
6	97/533B	Card revision

The accounts payable system in third normal form



Description

- *Normalization* is a formal process you can use to separate the data in a data structure into related tables. Normalization reduces *data redundancy*, which can cause storage and maintenance problems.
- In an *unnormalized data structure*, a table can contain information about two or more entities. It can also contain repeating columns, columns that contain repeating values, and data that's repeated in two or more rows.
- In a *normalized data structure*, each table contains information about a single entity, and each piece of information is stored in exactly one place.
- To normalize a data structure, you apply the *normal forms* in sequence. Although there are a total of seven normal forms, a data structure is typically considered normalized if the first three normal forms are applied.

Figure 10-7 How normalization works

How to identify the columns to be indexed

The last step in the design process is to identify the columns that should be indexed. An *index* is a structure that provides for locating one or more rows directly. Without an index, SQL Server has to perform a *table scan*, which involves searching through the entire table. Just as the index of a book has page numbers that direct you to a specific subject, a database index has pointers that direct the system to a specific row. This can speed performance not only when you're searching for rows based on a search condition, but when you're joining data from tables as well. If a join is done based on a primary key to foreign key relationship, for example, and an index is defined for the foreign key column, SQL Server can use that index to locate the rows for each primary key value.

In general, a column should meet the guidelines listed at the top of figure 10-8 before you consider creating an index for it. To start, you should index a column if it will be used frequently in search conditions or joins. Since you use foreign keys in most joins, you should typically index each foreign key column. The column should also contain mostly distinct values, and the values in the column should be updated infrequently. If these conditions aren't met, the overhead of maintaining the index will probably outweigh the advantages of using it.

SQL Server provides for two types of indexes. A *clustered index* defines the sequence in which the rows of the table are stored. Because of that, each table can contain a single clustered index. Although SQL Server creates a clustered index automatically for the primary key, you can change that if you need to. The second list in this figure presents some guidelines you can use to determine when to change the clustered index from the primary key column to another column. If you review these guidelines, you'll see that the primary key is usually the best column to use for the clustered index.

The other type of index is a *nonclustered index*. You can define up to 249 nonclustered indexes for each table. You should be aware, however, that the indexes must be updated each time you add, update, or delete a row. Because of that, you don't want to define more indexes than you need.

As you identify the indexes for a table, keep in mind that, like a key, an index can consist of two or more columns. This type of index is called a *composite index*. A special type of composite index that includes all of the columns used by a query is called a *covering index*. Although a covering index speeds retrieval, the overhead to maintain this type of index is significant, particularly if the table is updated frequently. Because of that, you won't usually define covering indexes.

Since you don't want to add unnecessary indexes, some database designers recommend adding indexes later when the database is in testing or production. That way, you can test the queries that are commonly run against the database and see how they perform. Then, if they don't perform well, you can add indexes.

Management Studio includes a feature that can help you identify and add indexes that would improve performance. To do that, enter a commonly run query into a Query Editor window. Then, select the *Query*→*Display Estimated Execution Plan* command. This should show any missing indexes that would improve performance. If you want to add the missing index, you can right-click it and select the *Missing Index Details* command to generate a script for creating that index.

When to create an index

- When the column is a foreign key
- When the column is used frequently in search conditions or joins
- When the column contains a large number of distinct values
- When the column is updated infrequently

When to reassign the clustered index

- When the column is used in almost every search condition
- When the column contains mostly distinct values
- When the column is small
- When the column values seldom, if ever, change
- When most queries against the column will return large result sets

Description

- An *index* provides a way for SQL Server to locate information more quickly. When it uses an index, SQL Server can go directly to a row rather than having to search through all the rows until it finds the ones you want.
- An index can be either *clustered* or *nonclustered*. Each table can have one clustered index and up to 249 nonclustered indexes.
- The rows of a table are stored in the sequence of the clustered index. By default, SQL Server creates a clustered index for the primary key. If you don't identify a primary key, the rows of the table are stored in the order in which they're entered.
- Indexes speed performance when searching and joining tables. However, they can't be used in search conditions that use the LIKE operator with a pattern that starts with a wildcard. And they can't be used in search conditions that include functions or expressions.
- You can create *composite indexes* that include two or more columns. You should use this type of index when the columns in the index are updated infrequently or when the index will cover almost every search condition on the table.
- Because indexes must be updated each time you add, update, or delete a row, you shouldn't create more indexes than you need.

Figure 10-8 How to identify the columns to be indexed

How to normalize a data structure

The topics that follow describe the seven normal forms and teach you how to apply the first three. As I said earlier, you apply these three forms to some extent in the first four database design steps, but these topics will give you more insight into the process. Then, the last topic explains when and how to denormalize a data structure. When you finish these topics, you'll have the basic skills for designing databases that are efficient and easy to use.

The seven normal forms

Figure 10-9 summarizes the seven normal forms. Each normal form assumes that the previous forms have already been applied. Before you can apply the third normal form, for example, the design must already be in the second normal form.

Strictly speaking, a data structure isn't normalized until it's in the fifth or sixth normal form. However, the normal forms past the third normal form are applied infrequently. Because of that, I won't present those forms in detail here. Instead, I'll just describe them briefly so you'll have an idea of how to apply them if you need to.

The *Boyce-Codd normal form* is a slightly stronger version of the third normal form that can be used to eliminate *transitive dependencies*. With this type of dependency, one column depends on another column, which depends on a third column. Most tables that are in the third normal form are also in the Boyce-Codd normal form.

The fourth normal form can be used to eliminate multiple *multivalued dependencies* from a table. A multivalued dependency is one where a primary key column has a one-to-many relationship with a non-key column. This normal form gets rid of misleading many-to-many relationships.

To apply the fifth normal form, you continue to divide the tables of the data structure into smaller tables until all redundancy has been removed. When further splitting would result in tables that couldn't be used to reconstruct the original table, the data structure is in fifth normal form. In this form, most tables consist of little more than key columns with one or two data elements.

The *domain-key normal form*, sometimes called the sixth normal form, is only of academic interest since no database system has implemented a way to apply it. For this reason, even normalization purists might consider a database to be normalized in fifth normal form.

This figure also lists the benefits of normalizing a data structure. To summarize, normalization produces smaller, more efficient tables. In addition, it reduces data redundancy, which makes the data easier to maintain and reduces the amount of storage needed for the database. Because of these benefits, you should always consider normalizing your data structures.

You should also be aware that the subject of normalization is a contentious one in the database community. In the academic study of computer science, normalization is considered a form of design perfection that should always be strived for. In practice, though, database designers and DBAs tend to use normalization as a flexible design guideline.

The seven normal forms

Normal form	Description
First (1NF)	The value stored at the intersection of each row and column must be a scalar value, and a table must not contain any repeating columns.
Second (2NF)	Every non-key column must depend on the primary key.
Third (3NF)	Every non-key column must depend <i>only</i> on the primary key.
Boyce-Codd (BCNF)	A non-key column can't be dependent on another non-key column. This prevents <i>transitive dependencies</i> , where column A depends on column C and column B depends on column C. Since both A and B depend on C, A and B should be moved into another table with C as the key.
Fourth (4NF)	A table must not have more than one <i>multivalued dependency</i> , where the primary key has a one-to-many relationship to non-key columns. This form gets rid of misleading many-to-many relationships.
Fifth (5NF)	The data structure is split into smaller and smaller tables until all redundancy has been eliminated. If further splitting would result in tables that couldn't be joined to recreate the original table, the structure is in fifth normal form.
Domain-key (DKNF) or Sixth (6NF)	Every constraint on the relationship is dependent only on key constraints and <i>domain</i> constraints, where a domain is the set of allowable values for a column. This form prevents the insertion of any unacceptable data by enforcing constraints at the level of a relationship, rather than at the table or column level. DKNF is less a design model than an abstract "ultimate" normal form. SQL Server has no way to implement the constraints required for DKNF.

The benefits of normalization

- Since a normalized database has more tables than an unnormalized database, and since each table can have a clustered index, the database has more clustered indexes. That makes data retrieval more efficient.
- Since each table contains information about a single entity, each index has fewer columns (usually one) and fewer rows. That makes data retrieval and insert, update, and delete operations more efficient.
- Each table has fewer indexes, which makes insert, update, and delete operations more efficient.
- Data redundancy is minimized, which simplifies maintenance and reduces storage.
- Queries against the database run faster.

Description

- Each normal form assumes that the design is already in the previous normal form.
- A database is typically considered to be normalized if it is in third normal form. The other four forms are not commonly used and are not covered in detail in this book.

Figure 10-9 The seven normal forms

How to apply the first normal form

Figure 10-10 illustrates how you apply the first normal form to an unnormalized invoice data structure consisting of the data elements that are shown in figure 10-2. The first two tables in this figure illustrate structures that aren't in first normal form. Both of these tables contain a single row for each invoice. Because each invoice can contain one or more line items, however, the first table allows for repeating values in the ItemDescription column. The second table is similar, except it includes a separate column for each line item description. Neither of these structures is acceptable in first normal form.

The third table in this figure has eliminated the repeating values and columns. To do that, it includes one row for each line item. Notice, however, that this has increased the data redundancy. Specifically, the vendor name and invoice number are now repeated for each line item. This problem can be solved by applying the second normal form.

Before I describe the second normal form, I want you to realize that I intentionally omitted many of the columns in the invoice data structure from the examples in this figure and the next figure. In addition to the columns shown here, for example, each of these tables would also contain the vendor address, invoice date, invoice total, etc. By eliminating these columns, it will be easier for you to focus on the columns that are affected by applying the normal forms.

The invoice data with a column that contains repeating values

	VendorName	InvoiceNumber	ItemDescription
1	Cahners Publishing	112897	C# ad, SQL ad, Library directory
2	Zylka Design	97/522	Catalogs, SQL flyer
3	Zylka Design	97/533B	Card revision

The invoice data with repeating columns

	VendorName	InvoiceNumber	ItemDescription1	ItemDescription2	ItemDescription3
1	Cahners Publishing	112897	C# ad	SQL ad	Library directory
2	Zylka Design	97/522	Catalogs	SQL flyer	NULL
3	Zylka Design	97/533B	Card revision	NULL	NULL

The invoice data in first normal form

	VendorName	InvoiceNumber	ItemDescription
1	Cahners Publishing	112897	C# ad
2	Cahners Publishing	112897	SQL ad
3	Cahners Publishing	112897	Library directory
4	Zylka Design	97/522	Catalogs
5	Zylka Design	97/522	SQL flyer
6	Zylka Design	97/533B	Card revision

Description

- For a table to be in first normal form, its columns must not contain repeating values. Instead, each column must contain a single, scalar value. In addition, the table must not contain repeating columns that represent a set of values.
- A table in first normal form often has repeating values in its rows. This can be resolved by applying the second normal form.

Figure 10-10 How to apply the first normal form

How to apply the second normal form

Figure 10-11 shows how to apply the second normal form. To be in second normal form, every column in a table that isn't a key column must depend on the entire primary key. This form only applies to tables that have composite primary keys, which is often the case when you start with data that is completely unnormaled. The table at the top of this figure, for example, shows the invoice data in first normal form after key columns have been added. In this case, the primary key consists of the InvoiceID and InvoiceSequence columns.

Now, consider the three non-key columns shown in this table. Of these three, only one, ItemDescription, depends on the entire primary key. The other two, VendorName and InvoiceNumber, depend only on the InvoiceID column. Because of that, these columns should be moved to another table. The result is a data structure like the second one shown in this figure. Here, all of the information related to an invoice is stored in the Invoices table, and all of the information related to an individual line item is stored in the InvoiceLineItems table.

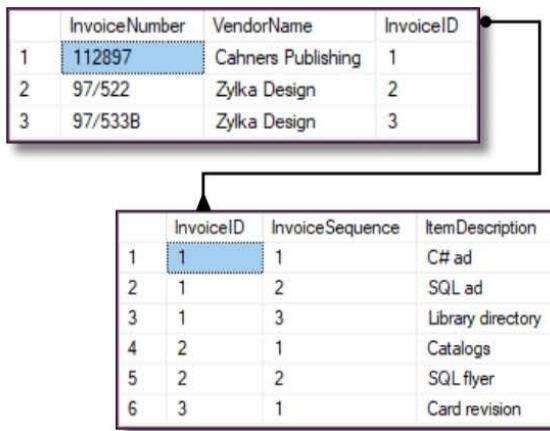
Notice that the relationship between these tables is based on the InvoiceID column. This column is the primary key of the Invoices table, and it's the foreign key in the InvoiceLineItems table that relates the rows in that table to the rows in the Invoices table. This column is also part of the primary key of the InvoiceLineItems table.

When you apply second normal form to a data structure, it eliminates some of the redundant row data in the tables. In this figure, for example, you can see that the invoice number and vendor name are now included only once for each invoice. In first normal form, this information was included for each line item.

The invoice data in first normal form with keys added

	InvoiceID	VendorName	InvoiceNumber	InvoiceSequence	ItemDescription
1	1	Cahners Publishing	112897	1	C# ad
2	1	Cahners Publishing	112897	2	SQL ad
3	1	Cahners Publishing	112897	3	Library directory
4	2	Zylka Design	97/522	1	Catalogs
5	2	Zylka Design	97/522	2	SQL flyer
6	3	Zylka Design	97/533B	1	Card revision

The invoice data in second normal form



Description

- For a table to be in second normal form, every non-key column must depend on the entire primary key. If a column doesn't depend on the entire key, it indicates that the table contains information for more than one entity. This is reflected by the table's composite key.
- To apply second normal form, you move columns that don't depend on the entire primary key to another table and then establish a relationship between the two tables.
- Second normal form helps remove redundant row data, which can save storage space, make maintenance easier, and reduce the chance of storing inconsistent data.

Figure 10-11 How to apply the second normal form

How to apply the third normal form

To apply the third normal form, you make sure that every non-key column depends *only* on the primary key. Figure 10-12 illustrates how you can apply this form to the data structure for the accounts payable system. At the top of this figure, you can see all of the columns in the Invoices and InvoiceLineItems tables in second normal form. Then, you can see a list of questions that you might ask about some of the columns in these tables when you apply third normal form.

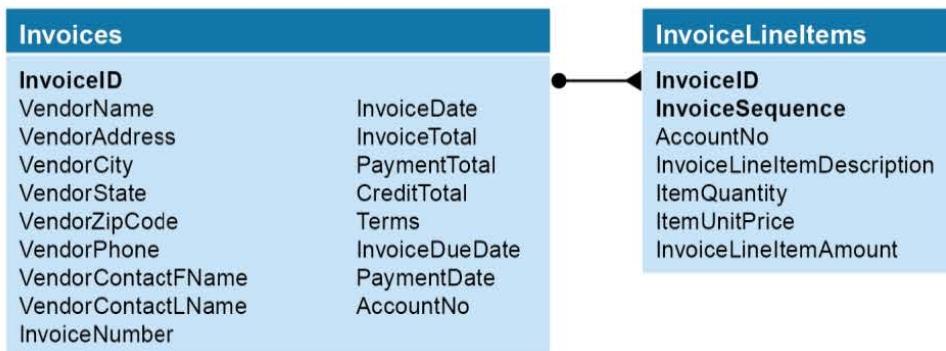
First, does the vendor information depend only on the InvoiceID column? Another way to phrase this question is, “Will the information for the same vendor change from one invoice to another?” If the answer is no, the vendor information should be stored in a separate table. That way, you can be sure that the vendor information for each invoice for a vendor will be the same. In addition, you will reduce the redundancy of the data in the Invoices table. This is illustrated by the diagram in this figure that shows the accounts payable system in third normal form. Here, a Vendors table has been added to store the information for each vendor. This table is related to the Invoices table by the VendorID column, which has been added as a foreign key to the Invoices table.

Second, does the Terms column depend only on the InvoiceID column? The answer to that question depends on how this column is used. In this case, it’s used not only to specify the terms for each invoice, but also to specify the default terms for a vendor. Because of that, the terms information could be stored in both the Vendors and the Invoices tables. To avoid redundancy, however, the information related to different terms can be stored in a separate table, as illustrated by the Terms table in this figure. As you can see, the primary key of this table is an identity column named TermsID. Then, a foreign key column named DefaultTermsID has been added to the Vendors table, and a foreign key column named TermsID has been added to the Invoices table.

Third, does the AccountNo column depend only on the InvoiceID column? Again, that depends on how this column is used. In this case, it’s used to specify the general ledger account number for each line item, so it depends on the InvoiceID and the InvoiceSequence columns. In other words, this column should be stored in the InvoiceLineItems table. In addition, each vendor has a default account number, which should be stored in the Vendors table. Because of that, another table named GLAccounts has been added to store the account numbers and account descriptions. Then, foreign key columns have been added to the Vendors and InvoiceLineItems tables to relate them to this table.

Fourth, can the InvoiceDueDate column in the Invoices table and the InvoiceLineItemAmount column in the InvoiceLineItems table be derived from other data in the database? If so, they depend on the columns that contain that data rather than on the primary key columns. In this case, the value of the InvoiceLineItemAmount column can always be calculated from the ItemQuantity and ItemUnitPrice columns. Because of that, this column could be omitted. Alternatively, you could omit the ItemQuantity and ItemUnitPrice columns and keep just the InvoiceLineItemAmount column. That’s what I did in the data structure shown in this figure. The solution you choose, however, depends on how the data will be used.

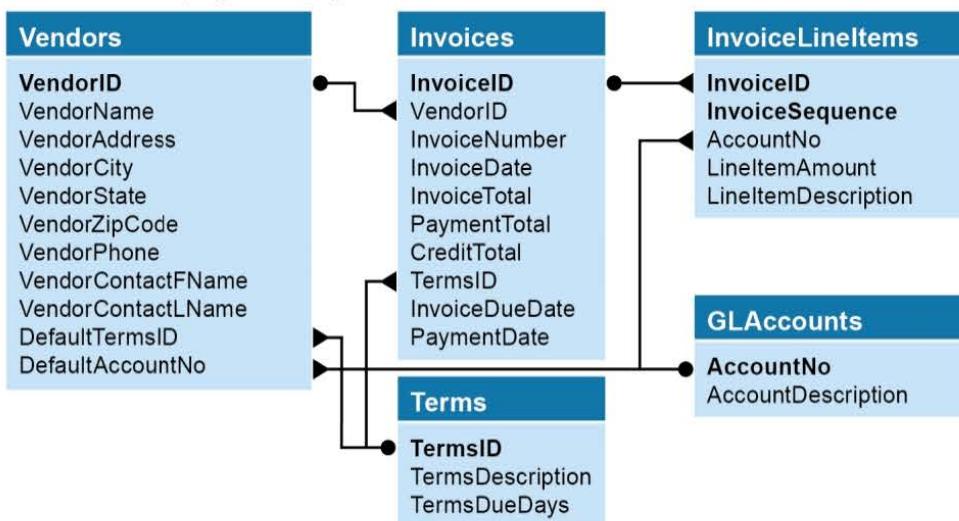
The accounts payable system in second normal form



Questions about the structure

1. Does the vendor information (VendorName, VendorAddress, etc.) depend only on the InvoiceID column?
2. Does the Terms column depend only on the InvoiceID column?
3. Does the AccountNo column depend only on the InvoiceID column?
4. Can the InvoiceDueDate and InvoiceLineItemAmount columns be derived from other data?

The accounts payable system in third normal form



Description

- For a table to be in third normal form, every non-key column must depend only on the primary key.
- If a column doesn't depend only on the primary key, it implies that the column is assigned to the wrong table or that it can be computed from other columns in the table. A column that can be computed from other columns contains *derived data*.

Figure 10-12 How to apply the third normal form

By contrast, although the InvoiceDueDate column could be calculated from the InvoiceDate column in the Invoices table and the TermsDueDays column in the related row of the Terms table, the system also allows this date to be overridden. Because of that, the InvoiceDueDate column should not be omitted. If the system didn't allow this value to be overridden, however, this column could be safely omitted.

When and how to denormalize a data structure

Denormalization is the deliberate deviation from the normal forms. Most denormalization occurs beyond the third normal form. By contrast, the first three normal forms are almost universally applied.

To illustrate when and how to denormalize a data structure, figure 10-13 presents the design of the accounts payable system in fifth normal form. Here, notice that the vendor addresses are stored in a separate table that contains the address, city, state, and zip code for each vendor. In addition, the vendor contacts are stored in a separate table that contains the first name, last name, and phone number for each vendor contact.

Since this allows you to use the same address or contact for multiple vendors, this reduces data redundancy if multiple vendors share the same address or contact person. However, since vendor address and contact information is now split across three tables, a query that retrieves vendor addresses and contact information requires two joins. By contrast, if you left the address and contact information in the Vendors table, no joins would be required, but the Vendors table would be larger.

In general, you should denormalize based on the way the data will be used. In this case, the system rarely needs to query vendors without address and contact information. In addition, it's rare that multiple vendors would have the same address or contact information. For these reasons, I've denormalized my design by eliminating the Addresses and Contacts tables.

You might also consider denormalizing a table if the data it contains is updated infrequently. In that case, redundant data isn't as likely to cause problems.

Finally, you should consider including derived data in a table if that data is used frequently in search conditions. For example, if you frequently query the Invoices table based on invoice balances, you might consider including a column that contains the balance due. That way, you won't have to calculate this value each time it's queried. Keep in mind, though, that if you store derived data, it's possible for it to deviate from the derived value. For this reason, you may need to protect the derived column so it can't be updated directly. Alternatively, you could update the table periodically to reset the value of the derived column.

Because normalization eliminates the possibility of data redundancy errors and optimizes the use of storage, you should carefully consider when and how to denormalize a data structure. In general, you should denormalize only when the increased efficiency outweighs the potential for redundancy errors and storage problems. Of course, your decision to denormalize should also be based on your knowledge of the real-world environment in which the system will be used. If you've carefully analyzed the real-world environment as outlined in this chapter, you'll have a good basis for making that decision.