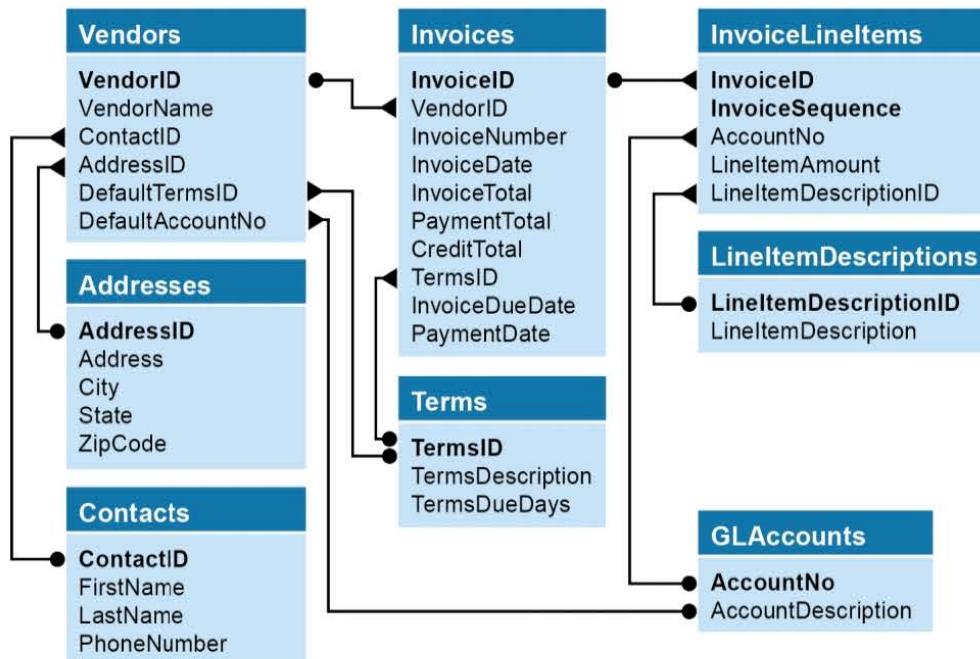


The accounts payable system in fifth normal form



When to denormalize

- When a column from a joined table is used repeatedly in search criteria, you should consider moving that column to the primary key table if it will eliminate the need for a join.
- If a table is updated infrequently, you should consider denormalizing it to improve efficiency. Because the data remains relatively constant, you don't have to worry about data redundancy errors once the initial data is entered and verified.
- Include columns with derived values when those values are used frequently in search conditions. If you do that, you need to be sure that the column value is always synchronized with the value of the columns it's derived from.

Description

- Data structures that are normalized to the fourth normal form and beyond typically require more joins than tables normalized to the third normal form.
- Most designers *denormalize* data structures to some extent, usually to the third normal form.
- *Denormalization* can result in larger tables, redundant data, and reduced performance.
- Only denormalize when necessary. It is better to adhere to the normal forms unless it is clear that performance will be improved by denormalizing.

Figure 10-13 When and how to denormalize a data structure

Perspective

Database design is a complicated subject. Because of that, it's impossible to teach you everything you need to know in a single chapter. With the skills you've learned in this chapter, however, you should now be able to design simple databases of your own. More important, you should now be able to evaluate the design of any database that you work with. That way, you can be sure that the queries you code will be as efficient and as effective as possible.

One aspect of database design that isn't covered in this chapter is designing the security of the database. Among other things, that involves creating login IDs and database users and assigning permissions. It may also involve organizing the tables and other objects in the database into two or more schemas. You'll learn more about how to implement database security in chapter 17.

Terms

data structure	deletion anomaly
entity	normalization
attribute	data redundancy
instance	unnormalized data structure
entity-relationship (ER) modeling	normalized data structure
CASE (computer-aided software engineering)	normal forms
composite key	index
linking table	table scan
connecting table	clustered index
associate table	nonclustered index
referential integrity	composite index
declarative referential integrity (DRI)	covering index
foreign key constraints	Boyce-Codd normal form
triggers	transitive dependency
orphaned row	multivalued dependency
update anomaly	domain-key normal form
insertion anomaly	derived data
	denormalized data structure
	denormalization

Exercises

1. Design a database diagram for a product orders database with four tables. Indicate the relationships between tables and identify the primary key and foreign keys in each table. Explain your design decisions.

Customers	Orders	OrderLineItems
CustomerID CustomerName CustomerAddress CustomerPhone ...	OrdersID CustomerID OrderDate ShipAddress ShipDate ...	OrderID OrderSequence ProductID Quantity UnitPrice ...

2. Add the two tables below into the design for exercise 1. Create additional tables and columns, if necessary. Explain your design decisions.

Shippers	Employees
ShipperID ShipperName ShipperAddress ShipperPhone ...	EmployeeID FirstName LastName SSN HireDate ...

3. Modify your design for exercise 2 to identify the columns that should be indexed, and explain your decision.
4. Design a database diagram that allows individuals to be assigned membership in one or more groups. Each group can have any number of individuals and each individual can belong to any number of groups. Create additional tables and columns, if necessary. Explain your design decisions.
5. Modify your design for exercise 4 to keep track of the *role* served by each individual in each group. Each individual can only serve one role in each group. Each group has a unique set of roles that members can fulfill. Create additional tables and columns, if necessary. Explain your design decisions.

How to create a database and its tables with SQL statements

Now that you've learned how to design a database, you're ready to learn how to implement your design. To do that, you use the set of SQL statements that are known as the data definition language (DDL). As an application programmer, you can use the DDL statements to create and modify the database objects such as tables and sequences that you need for testing. Beyond that, knowing what these statements do will give you a better appreciation for how a database works.

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
Perspective	372

An introduction to DDL

All of the SQL statements that you've seen so far have been part of the data manipulation language, or DML. But now, you'll learn how to use the SQL statements that are part of the data definition language. You use these statements to define the objects of a database.

The SQL statements for data definition

Figure 11-1 summarizes the *data definition language*, or *DDL*, statements that you use to create, delete, or change the *objects* of a database. In this chapter, you'll learn how to use the statements that work with databases, tables, indexes, and sequences. You'll learn how to use the statements that work with other objects in later chapters.

To work with the objects of a database, you often use the Management Studio that comes with SQL Server. This tool lets you create and change database objects using a graphical user interface. To do that, it generates and executes the DDL statements that implement the changes you've made. You'll learn how to use the Management Studio to work with database objects in chapter 12.

But first, this chapter teaches you how to code the DDL statements yourself. This is useful for two reasons. First, you sometimes need to examine and verify the DDL that's generated by the Management Studio. This is especially true for large database projects. Second, knowing the DDL statements helps you use the DML statements more effectively. Beyond that, if you ever use a DBMS that doesn't offer a graphical tool like the Management Studio, you have to code the DDL yourself.

Because the syntax of each of the DDL statements is complex, this chapter doesn't present complete syntax diagrams for the statements. Instead, the diagrams present only the most commonly used clauses. If you're interested in the complete syntax of any statement, of course, you can find it in the SQL Server documentation.

If you're working on a large database project, you probably won't have the option of coding DDL statements at all because that will be handled by a database administrator (DBA). This is a common practice because the DDL statements can destroy data if they're used incorrectly. In addition, many of the optional clauses for these statements are used for tuning the performance of the system, which is typically the role of a DBA.

For small projects, though, the SQL programmer may often have to serve as the DBA too. And even for large databases, the SQL programmer often uses the DDL to create and work with smaller databases that are needed for testing or for special projects.

DDL statements to create, modify, and delete objects

Statement	Description
CREATE DATABASE	Creates a new database.
CREATE TABLE	Creates a new table in the current database.
CREATE INDEX	Creates a new index for the specified table.
CREATE SEQUENCE	Creates a new sequence in the current database.
CREATE FUNCTION	Creates a new function in the current database.
CREATE PROCEDURE	Creates a new stored procedure in the current database.
CREATE TRIGGER	Creates a new trigger in the current database.
CREATE VIEW	Creates a new view in the current database.
ALTER TABLE	Modifies the structure of the specified table.
ALTER SEQUENCE	Modifies the attributes of a sequence.
ALTER FUNCTION	Modifies the specified function.
ALTER PROCEDURE	Modifies the specified stored procedure.
ALTER TRIGGER	Modifies the specified trigger.
ALTER VIEW	Modifies the specified view.
DROP DATABASE	Deletes the specified database.
DROP TABLE	Deletes the specified table.
DROP SEQUENCE	Deletes the specified sequence.
DROP INDEX	Deletes the specified index.
DROP FUNCTION	Deletes the specified function.
DROP PROCEDURE	Deletes the specified stored procedure.
DROP TRIGGER	Deletes the specified trigger.
DROP VIEW	Deletes the specified view.

Description

- You use the *data definition language (DDL)* statements to create, modify, and delete database objects such as the database itself, the tables contained in a database, and the indexes for those tables.
- Typically, a database administrator is responsible for using the DDL statements on production databases in a large database system. However, every SQL programmer should be comfortable using these statements so they can create and work with small databases for testing.
- In most cases, you'll use the graphical user interface of the Management Studio to create and maintain database objects as described in chapter 12. Although the Management Studio generates DDL statements for you, you may need to verify or correct these statements. To do that, you need to understand their syntax and use.
- If you use a SQL database other than SQL Server, it may not have a graphical tool for managing database objects. In that case, you must use the DDL statements.

Figure 11-1 The SQL statements for data definition

Rules for coding object names

When you create most database objects, you give them names. In SQL Server, the name of an object is its *identifier*. Each identifier can be up to 128 characters in length. To code an identifier, you typically follow the formatting rules presented in figure 11-2.

As you can see, the formatting rules limit the characters you can use in an identifier. For example, the first character of an identifier can be a letter, an underscore, an at sign, or a number sign. The characters that can be used in the remainder of the identifier include all of the characters allowed as the first character, plus numbers and dollar signs. Note that a regular identifier can't include spaces and can't be a Transact-SQL reserved keyword, which is a word that's reserved for use by SQL Server.

The first set of examples in this figure presents some valid regular identifiers. Notice that the identifier in the second example starts with a number sign. This type of identifier is used for a temporary table or procedure. Similarly, an identifier that starts with an at sign as in the fifth example is used for a local variable or parameter. You'll learn about these special types of identifiers in chapters 14 and 15.

In most cases, you'll create objects with identifiers that follow the formatting rules shown here. If you're working with an existing database, however, the identifiers may not follow these rules. In that case, you have to delimit the identifiers to use them in SQL statements. You can code a delimited identifier by enclosing it in either brackets or double quotes. The second set of examples shows how this works. Here, two of the identifiers are enclosed in brackets and one is enclosed in double quotes. The identifier in the first example must be delimited because it starts with a percent sign. The identifier in the second example must be delimited because it includes spaces. The third example illustrates that even when a name follows the formatting rules, you can delimit it. In most cases, though, there's no reason to do that.

Formatting rules for regular identifiers

- The first character of an identifier must be a letter as defined by the Unicode Standard 3.2, an underscore (_), an at sign (@), or a number sign (#).
- All characters after the first must be a letter as defined by the Unicode Standard 3.2, a number, an at sign, a dollar sign (\$), a number sign, or an underscore.
- An identifier can't be a Transact-SQL reserved keyword.
- An identifier can't contain spaces or special characters other than those already mentioned.

Valid regular identifiers

```
Employees  
#PaidInvoices  
ABC$123  
Invoice_Line_Items  
@TotalDue
```

Valid delimited identifiers

```
[%Increase]  
"Invoice Line Items"  
[@TotalDue]
```

Description

- The name of an object in SQL Server is called its *identifier*. Most objects are assigned an identifier when they're created. Then, the identifier can be used to refer to the object.
- SQL Server provides for two classes of identifiers. Regular identifiers follow the formatting rules for identifiers. Delimited identifiers are enclosed in brackets ([]) or double quotation marks ("") and may or may not follow the formatting rules. If an identifier doesn't follow the formatting rules, it must be delimited.
- An identifier can contain from 1 to 128 characters.
- An at sign (@) at the beginning of an identifier indicates that the identifier is a local variable or parameter, a number sign (#) indicates that the identifier is a temporary table or procedure, and two number signs (##) indicates that the identifier is a global temporary object. See chapters 14 and 15 for details.

Figure 11-2 Rules for coding object names

How to create databases, tables, and indexes

The primary role of the DDL statements is to define database objects on the server. So to start, the three topics that follow will teach you how to code the DDL statements that you use to create databases, tables, and indexes.

How to create a database

Figure 11-3 presents the basic syntax of the CREATE DATABASE statement. This statement creates a new database on the current server. In many cases, you'll code this statement with just a database name to create the database with the default options. This is illustrated by the first example in this figure.

The CREATE DATABASE statement in this example creates a database named New_AP. When you issue a statement like this one, SQL Server creates two files and allocates space for them. The first file, New_AP.mdf, will hold the data for the database. The second file, New_AP_log.ldf, will keep a log of any changes made to the database.

If you want to use a database that was created on another server, you can copy the mdf file to your server. At that point, though, it's simply a data file. To be able to use the database, you have to *attach* it to your server. To do that, you use two of the optional clauses in the CREATE DATABASE statement, ON PRIMARY and FOR ATTACH. As you can see in the second example, you specify the name of the file that contains the database in the ON PRIMARY clause. Then, instead of creating a new database, SQL Server simply makes the existing database available from the current server.

In this figure, the statement that attaches the database doesn't specify an existing transaction log file for the database. As a result, SQL Server attempts to use an ldf file with the same name as the database followed by "_log". For example, for the data stored in the file named Test_AP.mdf, SQL Server will attempt to use the log data stored in the file named Test_AP_log.ldf. If a log file with that name doesn't exist, SQL Server will create a new log file with that name. However, it will also return an error message that indicates that a new log file was created.

Most of the clauses that aren't included in the syntax shown here are used to tune the database by changing the locations of the database files. For small databases, though, this tuning usually isn't necessary. If you want to learn about these options, you can refer to the description of this statement in the SQL Server documentation.

The basic syntax of the CREATE DATABASE statement

```
CREATE DATABASE database_name  
[ON [PRIMARY] (FILENAME = 'file_name')]  
[FOR ATTACH]
```

A statement that creates a new database

```
CREATE DATABASE New_AP;
```

The response from the system

Commands completed successfully.

A statement that attaches an existing database file

```
CREATE DATABASE Test_AP  
ON PRIMARY (FILENAME =  
'C:\Murach\SQL Server 2019\Datasets\Test_AP.mdf')  
FOR ATTACH;
```

The response from the system

Commands completed successfully.

Description

- The CREATE DATABASE statement creates a new, empty database on the current server. Although the ANSI standards don't include this statement, it's supported by virtually all SQL database systems. The optional clauses shown here, however, are supported only by SQL Server.
- If you code this statement without any options, the new database is created using the default settings and the database files are stored in the default directory on the hard drive. For most small database projects, these settings are acceptable.
- One of the files SQL Server creates when it executes the CREATE DATABASE statement is a *transaction log file*. This file is used to record modifications to the database. SQL Server generates the name for this file by appending “_log” to the end of the database name. The database name is limited to 123 characters.
- If you have a copy of a database file that you'd like to work with on your server, you can use the FOR ATTACH clause in addition to the ON PRIMARY clause to *attach* the file as a database to the current server.
- Most of the optional clauses that have been omitted from this syntax are used to specify the underlying file structure of the database. These clauses are used by DBAs to tune the performance of the database. See the SQL Server documentation for details.

Warning

- On some systems, the CREATE DATABASE statement can overwrite an existing database. Because of that, you'll want to check with the DBA before using this statement.

How to create a table

Figure 11-4 presents the basic syntax of the CREATE TABLE statement. By default, this statement creates a new table in the default schema, dbo, within the current database. If that's not what you want, you can qualify the table name with the schema name or the database name.

All of the clauses and keywords for the CREATE TABLE statement can be divided into two categories: attributes that affect a single column and attributes that affect the entire table. This figure summarizes some of the common column attributes. You'll learn about the table attributes later in this chapter.

In its simplest form, the CREATE TABLE statement consists of the name of the new table followed by the names and data types of its columns. This is illustrated by the first example of this figure. Notice that the column definitions are enclosed in parentheses. In most cases, you'll also code one or more attributes for each column as illustrated by the second example in this figure.

To identify whether a column can accept null values, you code either the NULL or NOT NULL keyword. If you omit both keywords, the default value is NULL unless the column is also defined as the primary key, in which case the default is NOT NULL.

The PRIMARY KEY keywords identify the primary key for the table. To create a primary key based on a single column, you can code these keywords as an attribute of that column. To create a primary key based on two or more columns, however, you must code PRIMARY KEY as a table attribute. You'll see how to do that in a later figure.

When you identify a column as the primary key, two of the column's attributes are changed automatically. First, the column is forced to be NOT NULL. Second, the column is forced to contain a unique value for each row. In addition, a clustered index is automatically created based on the column.

In addition to a primary key, you can also define one or more unique keys using the UNIQUE keyword. Unlike a primary key column, a unique key column can contain null values. And instead of creating a clustered index for the key, SQL Server creates a nonclustered index. Like the PRIMARY keyword, you can code the UNIQUE keyword at either the column or the table level.

NOT NULL, PRIMARY KEY, and UNIQUE are examples of *constraints*. Constraints are special attributes that restrict the data that can be stored in the columns of a table. You'll learn how to code other constraints in a moment.

The IDENTITY keyword defines a column as an identity column. As you know, SQL Server assigns an identity column a unique integer value. This value is generated by incrementing the previous value for the column. SQL Server allows only one identity column per table, and that column is typically used as the primary key.

The DEFAULT attribute specifies a default value for a column. This value is used if another value isn't specified. The default value that's specified must correspond to the data type for the column.

The last attribute, SPARSE, optimizes the storage of null values for a column. Since this optimization requires more overhead to retrieve non-null

The basic syntax of the CREATE TABLE statement

```
CREATE TABLE table_name
  (column_name_1 data_type [column_attributes]
  [, column_name_2 data_type [column_attributes]]...
  [, table_attributes])
```

Common column attributes

Attribute	Description
NULL NOT NULL	Indicates whether or not the column can accept null values. If omitted, NULL is the default unless PRIMARY KEY is specified.
PRIMARY KEY UNIQUE	Identifies the primary key or a unique key for the table. If PRIMARY is specified, the NULL attribute isn't allowed.
IDENTITY	Identifies an identity column. Only one identity column can be created per table.
DEFAULT default_value	Specifies a default value for the column.
SPARSE	Optimizes storage of null values for the column. This attribute was introduced with SQL Server 2008.

A statement that creates a table without column attributes

```
CREATE TABLE Vendors
  (VendorID      INT,
  VendorName    VARCHAR(50));
```

A statement that creates a table with column attributes

```
CREATE TABLE Invoices
  (InvoiceID      INT          PRIMARY KEY IDENTITY,
  VendorID       INT          NOT NULL,
  InvoiceDate    DATE         NULL,
  InvoiceTotal   MONEY        NULL DEFAULT 0);
```

A column definition that uses the SPARSE attribute

```
VendorAddress2    VARCHAR(50)    SPARSE NULL
```

Description

- The CREATE TABLE statement creates a table based on the column definitions, column attributes, and table attributes you specify. A database can contain as many as two billion tables.
- A table can contain between one and 1,024 columns. Each column must have a unique name and must be assigned a data type. In addition, you can assign one or more of the column attributes shown above.
- You can also assign one or more constraints to a column or to the entire table. See figures 11-7, 11-8, and 11-9 for details.
- For the complete syntax of the CREATE TABLE statement, refer to the SQL Server documentation.

Figure 11-4 How to create a table

values, you should only use it when a column contains a high percentage of null values. As a general guideline, it usually makes sense to use the SPARSE attribute when at least 60% of the column's values are null.

How to create an index

Figure 11-5 presents the basic syntax of the CREATE INDEX statement, which creates an index based on one or more columns of a table. This syntax omits some of the optional clauses that you can use for tuning the indexes for better performance. This tuning is often done by DBAs working with large databases, but usually isn't necessary for small databases.

In the last chapter, you learned that a table can have one clustered index and up to 249 nonclustered indexes. By default, SQL Server creates a clustered index based on the primary key of a table, which is usually what you want. Because of that, you'll rarely create a clustered index.

To create an index, you name the table and columns that the index will be based on in the ON clause. For each column, you can specify the ASC or DESC keyword to indicate whether you want the index sorted in ascending or descending sequence. If you don't specify a sort order, ASC is the default.

The first example in this figure creates an index based on the VendorID column in the Invoices table. Because none of the optional keywords are specified, this creates a nonclustered index that is sorted in ascending sequence.

The second example creates a nonclustered index based on two columns in the Invoices table: InvoiceDate and InvoiceTotal. Notice here that the InvoiceDate column is sorted in descending sequence. That way, the most recent invoices will occur first.

For most databases, you can achieve adequate performance using indexes like the ones shown in the first two examples. Since these indexes index every row in the table, they are known as *full-table indexes*. However, when a database becomes very large, you may be able to improve performance by creating *filtered indexes*, which are indexes that use a WHERE clause to filter the rows in the index.

In general, it makes sense to create a filtered index when the number of rows in the index is small compared to the total number of rows in the table as shown in the third and fourth examples. That way, it's easier for the database engine to use and maintain the index, which results in better performance. Otherwise, a full-table index may yield better performance.

You should also notice the names that are assigned to the indexes in these examples. Although you can name an index anything you like, SQL Server's convention is to prefix index names with the characters *IX_*. So I recommend you do that too. Then, if the index is based on a single column, you can follow the prefix with the name of that column as shown in the first example. Or, if an index is based on two or more columns, you can use the table name instead of the column names as shown in the second example. If necessary, you can add additional information to identify the index as shown in the last two examples.

The basic syntax of the CREATE INDEX statement

```
CREATE [CLUSTERED | NONCLUSTERED] INDEX index_name  
    ON table_name (col_name_1 [ASC | DESC] [, col_name_2 [ASC | DESC]]...)  
    [WHERE filter-condition]
```

A statement that creates a nonclustered index based on a single column

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

A statement that creates a nonclustered index based on two columns

```
CREATE INDEX IX_Invoices  
    ON Invoices (InvoiceDate DESC, InvoiceTotal);
```

A statement that creates a filtered index for a subset of data in a column

```
CREATE INDEX IX_InvoicesPaymentFilter  
    ON Invoices (InvoiceDate DESC, InvoiceTotal)  
    WHERE PaymentDate IS NULL;
```

A statement that creates a filtered index for categories in a column

```
CREATE INDEX IX_InvoicesDateFilter  
    ON Invoices (InvoiceDate DESC, InvoiceTotal)  
    WHERE InvoiceDate > '2020-02-01';
```

Description

- You use the CREATE INDEX statement to create an index for a table. An index can improve performance when SQL Server searches for rows in the table.
- SQL Server automatically creates a clustered index for a table's primary key. If that's not what you want, you can drop the primary key constraint using the ALTER TABLE statement shown in figure 11-11 and then recreate the primary key with a nonclustered index.
- Each table can have a single clustered index and up to 999 nonclustered indexes. SQL Server automatically creates a nonclustered index for each unique key other than the primary key.
- By default, an index is sorted in ascending sequence. If that's not what you want, you can code the DESC keyword. The sequence you use should be the sequence in which the rows are retrieved most often when using that index.
- A *full-table index* is an index that applies to every row in the table.
- A *filtered index* is a type of nonclustered index that includes a WHERE clause that filters the rows that are included in the index. Filtered indexes were introduced with SQL Server 2008.
- A filtered index can improve performance when the number of rows in the index is small compared to the total number of rows in the table.
- For more details about working with filtered indexes, refer to the SQL Server documentation.

Figure 11-5 How to create an index

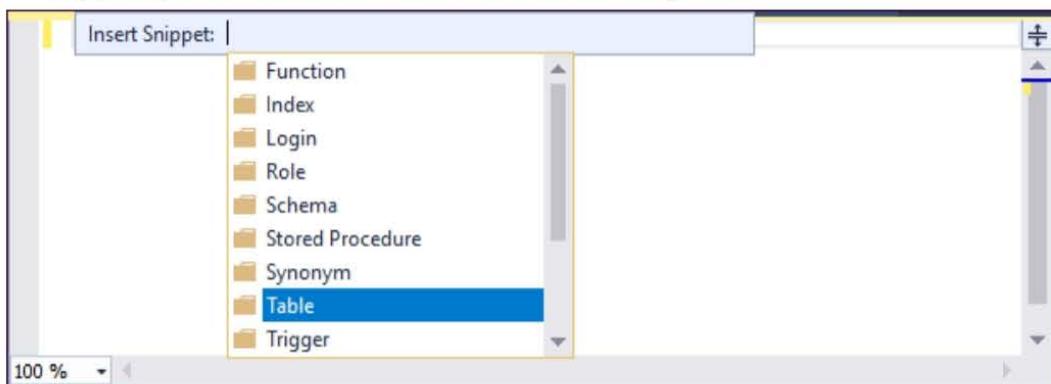
How to use snippets to create database objects

Now that you're familiar with the statements for creating tables and indexes, you should know that you don't have to start these statements from scratch when you use the SQL Server Management Studio. Instead, you can use a feature introduced with SQL Server 2012 called Transact-SQL *snippets* that provide the basic structure of the statements for creating these objects as well as many others. Figure 11-6 shows how snippets work.

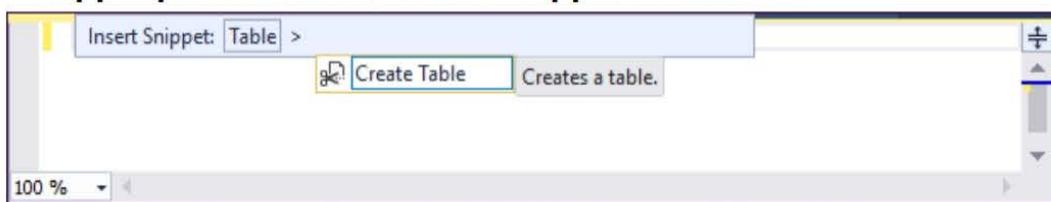
To insert a snippet, you use the *snippet picker*. The easiest way to display this picker is to right-click in the Query Editor window and then select the Insert Snippet command from the shortcut menu. When you do, a list of folders that correspond to different types of database objects is displayed. Then, you can double-click the folder for the type of object you want to create to display a list of the snippets for that object. Finally, you can double-click a snippet to insert it into your code. In this figure, I inserted the snippet for the CREATE TABLE statement.

After you insert a snippet, you need to replace the highlighted portions of code so the object is defined appropriately. For the CREATE TABLE statement shown here, for example, you'll want to change the name of the table, and you'll want to change the definitions for the two columns. You may also want to change the schema name. In addition, you'll typically need to add code to the snippet. To create a table, for example, you'll need to enter the definitions for any additional columns.

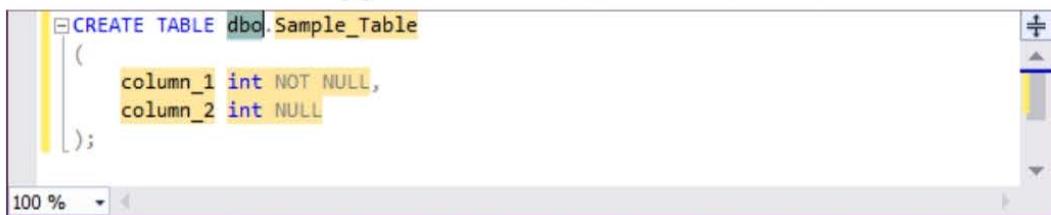
The snippet picker with a list of database object folders



The snippet picker with the list of snippets for a table



The CREATE TABLE snippet after it has been inserted



Description

- Transact-SQL *snippets* help you write statements for creating database objects.
- To insert a Transact-SQL snippet, right-click in the Query Editor window and select the Insert Snippet command from the resulting menu. Then, use the *snippet picker* to select a snippet.
- To select a snippet using the snippet picker, double-click on the folder for the object you want to create, then double-click on a snippet in the list that's displayed. Alternatively, you can select the folder or snippet and then press the Tab or Enter key.
- Once a snippet has been inserted, you can replace the highlighted portions with your own code and add any other required code. To move from one highlighted portion of code to the next, press the Tab key. To move to the previous highlighted portion, press the Shift+Tab keys.

Figure 11-6 How to use snippets to create database objects

How to use constraints

As you've already learned, you can code constraints to restrict the values that can be stored in a table. These constraints are tested before a new row is added to a table or an existing row is updated. Then, if one or more of the constraints aren't satisfied, the operation isn't performed.

The constraints you've seen so far identify a primary key or unique key column or prevent null values in a column. Now you'll learn how to code other types of constraints. In particular, you'll learn how to code constraints to validate data and to enforce referential integrity.

An introduction to constraints

Figure 11-7 summarizes the five types of constraints provided by SQL Server. Except for NOT NULL, each of these constraints can be coded at either the column level or the table level. You've already seen how to code a primary key constraint at the column level, and you can code a unique key constraint in the same way. Now, the first example in this figure shows how to code a primary key constraint at the table level.

In this example, the primary key consists of two columns. Because of that, it can't be defined at the column level. Notice that when you code a constraint at the table level, you must code a comma at the end of the preceding column definition. If you don't, SQL Server will try to associate the constraint with the preceding column, and an error will result.

Two types of constraints you haven't seen yet are *check constraints* and *foreign key constraints*. You'll learn more about these types of constraints in the next two topics. To illustrate the difference between *column-level constraints* and *table-level constraints*, however, the second and third examples in this figure show two ways you can code the same two check constraints. The first example uses column-level constraints to limit the values in the InvoiceTotal and PaymentTotal columns to numbers greater than or equal to zero. The second example uses a compound condition to specify both constraints at the table level. Although the first technique is preferred, the second example illustrates that a table-level constraint can refer to any of the columns in a table. By contrast, a column-level constraint can refer only to the column that contains the constraint.

Column and table constraints

Constraint	Used as a column-level constraint	Used as a table-level constraint
NOT NULL	Prevents null values from being stored in the column.	n/a
PRIMARY KEY	Requires that each row in the table have a unique value in the column. Null values are not allowed.	Requires that each row in the table have a unique set of values over one or more columns. Null values are not allowed.
UNIQUE	Requires that each row in the table have a unique value in the column.	Requires that each row in the table have a unique set of values over one or more columns.
CHECK	Limits the values for a column.	Limits the values for one or more columns.
[FOREIGN KEY] REFERENCES	Enforces referential integrity between a column in the new table and a column in a related table.	Enforces referential integrity between one or more columns in the new table and one or more columns in the related table.

A statement that creates a table with a two-column primary key constraint

```
CREATE TABLE InvoiceLineItems1
(InvoiceID           INT          NOT NULL,
InvoiceSequence      SMALLINT    NOT NULL,
InvoiceLineItemAmount MONEY      NOT NULL,
PRIMARY KEY (InvoiceID, InvoiceSequence));
```

A statement that creates a table with two column-level check constraints

```
CREATE TABLE Invoices1
(InvoiceID           INT      NOT NULL IDENTITY PRIMARY KEY,
InvoiceTotal         MONEY   NOT NULL CHECK (InvoiceTotal >= 0),
PaymentTotal         MONEY   NOT NULL DEFAULT 0 CHECK (PaymentTotal >= 0));
```

The same statement with the check constraints coded at the table level

```
CREATE TABLE Invoices2
(InvoiceID           INT      NOT NULL IDENTITY PRIMARY KEY,
InvoiceTotal         MONEY   NOT NULL,
PaymentTotal         MONEY   NOT NULL DEFAULT 0,
CHECK ((InvoiceTotal >= 0) AND (PaymentTotal >= 0)));
```

Description

- *Constraints* are used to enforce the integrity of the data in a table by defining rules about the values that can be stored in the columns of the table. Constraints can be used at the column level to restrict the value of a single column or at the table level to restrict the value of one or more columns.
- You code a *column-level constraint* as part of the definition of the column it constrains. You code a *table-level constraint* as if it were a separate column definition, and you name the columns it constrains within that definition.
- Constraints are tested before a new row is added to a table or an existing row is updated. If the new or modified row meets all of the constraints, the operation succeeds. Otherwise, an error occurs and the operation fails.

Figure 11-7 An introduction to constraints

How to use check constraints

To code a check constraint, you use the syntax presented in figure 11-8. As you can see, you code the CHECK keyword followed by the condition that the data must satisfy. This condition is evaluated as a Boolean expression. The insert or update operation that's being performed is allowed only if this expression evaluates to a True value.

The first example in this figure uses a column-level check constraint to limit the values in the InvoiceTotal column to numbers greater than zero. This is similar to the constraints you saw in the previous figure. Notice that if you try to store a negative value in this column as illustrated by the INSERT statement in this example, the system responds with an error and the insert operation is terminated.

The second example shows how you can use a check constraint to limit a column to values that have a specific format. Note that although this constraint limits the values in a single column, it's coded at the table level because it refers to a column other than the one being constrained. The first part of the condition in this check constraint uses a LIKE expression to restrict the VendorCode column to six characters, consisting of two alphabetic characters followed by four numeric characters. Then, the second part of the condition restricts the first two characters of the VendorCode column to the first two characters of the VendorName column.

In general, you should use check constraints to restrict the values in a column whenever possible. In some situations, however, check constraints can be too restrictive. As an example, consider a telephone number that's constrained to the typical “(000) 000-0000” format used for US phone numbers. The problem with this constraint is that it wouldn't let you store phone numbers with extensions (although you could store extensions in a separate column) or phone numbers with an international format.

For this reason, check constraints aren't used by all database designers. That way, the database can store values with formats that weren't predicted when the database was designed. However, this flexibility comes at the cost of allowing some invalid data. For some systems, this tradeoff is acceptable.

Keep in mind, too, that application programs that add and update data can also include data validation. In that case, check constraints may not be necessary. Because you can't always assume that an application program will check for valid data, though, you should include check constraints whenever that makes sense.

The syntax of a check constraint

`CHECK (condition)`

A column-level check constraint that limits invoices to positive amounts

A statement that defines the check constraint

```
CREATE TABLE Invoices3
(InvoiceID      INT      NOT NULL IDENTITY PRIMARY KEY,
InvoiceTotal    MONEY    NOT NULL CHECK (InvoiceTotal > 0));
```

An INSERT statement that fails due to the check constraint

```
INSERT Invoices3
VALUES (-100);
```

The response from the system

The INSERT statement conflicted with the CHECK constraint "CK_Invoices3_Invoi_0BC6C43E". The conflict occurred in database "New_AP", table "dbo.Invoices3", column 'InvoiceTotal'.
The statement has been terminated.

A table-level check constraint that limits vendor codes to a specific format

A statement that defines the check constraint

```
CREATE TABLE Vendors1
(VendorCode      CHAR(6)      NOT NULL PRIMARY KEY,
VendorName     VARCHAR(50)   NOT NULL,
CHECK          ((VendorCode LIKE '[A-Z][A-Z][0-9][0-9][0-9][0-9]') AND
                (LEFT(VendorCode,2) = LEFT(VendorName,2))));
```

An INSERT statement that fails due to the check constraint

```
INSERT Vendors1
VALUES ('Mc4559','Castle Printers, Inc.');
```

The response from the system

The INSERT statement conflicted with the CHECK constraint "CK_Vendors1_164452B1". The conflict occurred in database "New_AP", table "dbo.Vendors1".
The statement has been terminated.

Description

- *Check constraints* limit the values that can be stored in the columns of a table.
- The condition you specify for a check constraint is evaluated as a Boolean expression. If the expression is true, the insert or update operation proceeds. Otherwise, it fails.
- A check constraint that's coded at the column level can refer only to that column. A check constraint that's coded at the table level can refer to any column in the table.

How to use foreign key constraints

Figure 11-9 presents the syntax of a foreign key constraint, also known as a *reference constraint*. This type of constraint is used to define the relationships between tables and to enforce referential integrity.

To create a foreign key constraint at the column level, you code the REFERENCES keyword followed by the name of the related table and the name of the related column in parentheses. Although you can also code the FOREIGN KEY keywords, these keywords are optional and are usually omitted. After the REFERENCES clause, you can code the ON DELETE and ON UPDATE clauses. I'll have more to say about these clauses in a moment.

The first two statements in this figure show how to create two related tables. The first statement creates the primary key table, a table named Vendors9. Then, the second statement creates the foreign key table, named Invoices9. Notice that the VendorID column in this table includes a REFERENCES clause that identifies the VendorID column in the Vendors9 table as the related column.

The next statement in this figure is an INSERT statement that attempts to insert a row into the Invoices9 table. Because the Vendors9 table doesn't contain a row with the specified VendorID value, however, the insert operation fails.

Before I go on, you should realize that although the foreign key of one table is typically related to the primary key of another table, that doesn't have to be the case. Instead, a foreign key can be related to any unique key. For the purposes of this topic, though, I'll assume that the related column is a primary key column.

By default, you can't delete a row from the primary key table if related rows exist in a foreign key table. Instead, you have to delete the related rows from the foreign key table first. If that's not what you want, you can code the ON DELETE clause with the CASCADE option. Then, when you delete a row from the primary key table, the delete is *cascaded* to the related rows in the foreign key table. Because a *cascading delete* can destroy valuable data if it's used improperly, you should use it with caution.

The ON UPDATE clause is similar. If you code the CASCADE keyword in this clause, a change to the value of a primary key is automatically cascaded to the related rows in the foreign key table. Otherwise, the change isn't allowed. Since most tables are designed so their primary key values don't change, you won't usually code the ON UPDATE clause.

When you code a foreign key constraint at the column level, you relate a single column in the foreign key table to a single column in the primary key table. If the keys consist of two or more columns, however, you have to code the constraint at the table level. For example, suppose that a foreign key consists of two columns named CustomerID2 and CustomerID4 and that the foreign key is related to two columns with the same name in a table named Customers. Then, you would define the foreign key constraint like this:

```
FOREIGN KEY (CustomerID2, CustomerID4)
    REFERENCES Customers (CustomerID2, CustomerID4)
```

In this case, you must include the FOREIGN KEY keywords.

The syntax of a column-level foreign key constraint

```
[FOREIGN KEY] REFERENCES ref_table_name (ref_column_name)
    [ON DELETE {CASCADE|NO ACTION}]
    [ON UPDATE {CASCADE|NO ACTION}]
```

The syntax of a table-level foreign key constraint

```
FOREIGN KEY (column_name_1 [, column_name_2]...)
    REFERENCES ref_table_name (ref_column_name_1 [, ref_column_name_2]...)
    [ON DELETE {CASCADE|NO ACTION}]
    [ON UPDATE {CASCADE|NO ACTION}]
```

A foreign key constraint defined at the column level

A statement that creates the primary key table

```
CREATE TABLE Vendors9
(VendorID      INT NOT NULL PRIMARY KEY,
VendorName     VARCHAR(50) NOT NULL);
```

A statement that creates the foreign key table

```
CREATE TABLE Invoices9
(InvoiceID      INT NOT NULL PRIMARY KEY,
VendorID        INT NOT NULL REFERENCES Vendors9 (VendorID),
InvoiceTotal    MONEY NULL);
```

An INSERT statement that fails because a related row doesn't exist

```
INSERT Invoices9
VALUES (1, 99, 100);
```

The response from the system

```
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Invoices9_Vendo_1367E606". The conflict occurred in database "New_AP",
table "dbo.Vendors9", column 'VendorID'.
The statement has been terminated.
```

Description

- You use the FOREIGN KEY clause to define a *foreign key constraint*, also called a *reference constraint*. A foreign key constraint defines the relationship between two tables and enforces referential integrity.
- A foreign key constraint that's coded at the column level can only relate a single column in the new table to a single column in the related table. A constraint that's coded at the table level can relate two tables by two or more columns.
- Typically, a foreign key constraint refers to the primary key of the related table. However, it can also refer to a unique key.
- The ON DELETE clause specifies what happens to rows in the table if the row in the related table with the same key value is deleted. The ON UPDATE clause specifies what happens to rows in the table if the key of the related row is updated.
- The CASCADE keyword causes the rows in this table to be deleted or updated to match the row in the related table. This is known as a *cascading delete* or a *cascading update*.
- The NO ACTION keyword prevents the row in the related table from being deleted or updated and causes an error to be raised. This is usually the preferred option.

How to change databases and tables

After you create a database, you may need to change it. For example, you may need to add a new table or index. To do that, you can use the CREATE statements that you've already learned. If you need to modify an existing table, however, or if you need to delete an existing index, table, or database, you'll need to use the statements that follow.

How to delete an index, table, or database

Figure 11-10 presents the syntax of the three DROP statements you use to delete an index, a table, or a database. You can use these statements to delete one or more indexes, tables, or databases on the current server.

If other objects depend on the object you're trying to delete, SQL Server won't allow the deletion. For example, you can't delete a table if a foreign key constraint in another table refers to that table, and you can't delete an index if it's based on a primary key or a unique key. In addition, you can't drop a database that's currently in use.

You should also know that when you delete a table, many of the objects related to that table are deleted as well. That includes any indexes, triggers, or constraints defined for the table. By contrast, any views or stored procedures that are associated with a deleted table are not deleted. Instead, you have to delete these objects explicitly using the statements you'll learn in chapter 15.

Because the DROP statements delete objects permanently, you'll want to use them cautiously. In fact, you may want to create a backup copy of the database before using any of these statements. That way, you can restore the database if necessary.

The syntax of the DROP INDEX statement

```
DROP INDEX index_name_1 ON table_name_1 [, index_name_2 ON table_name_2]...
```

The syntax of the DROP TABLE statement

```
DROP TABLE table_name_1 [, table_name_2]...
```

The syntax of the DROP DATABASE statement

```
DROP DATABASE database_name_1 [, database_name_2]...
```

Statements that delete database objects

A statement that deletes an index from the Invoices table

```
DROP INDEX IX_Invoices ON Invoices;
```

A statement that deletes a table from the current database

```
DROP TABLE Vendors1;
```

A statement that qualifies the table to be deleted

```
DROP TABLE New_AP.dbo.Vendors1;
```

A statement that deletes a database

```
DROP DATABASE New_AP;
```

Description

- You can use the DROP INDEX statement to delete one or more indexes from one or more tables in any database on the current server.
- You can use the DROP TABLE statement to delete one or more tables from any database on the current server. To delete a table from a database other than the current database, you must qualify the table name with the database name.
- You can use the DROP DATABASE statement to delete one or more databases from the current server.
- You can't delete a table if a foreign key constraint in another table refers to that table.
- When you delete a table, all of the data, indexes, triggers, and constraints are deleted. Any views or stored procedures associated with the table must be deleted explicitly.
- You can't delete an index that's based on a primary key or unique key constraint. To do that, you have to use the ALTER TABLE statement. See figure 11-11 for details.

Warnings

- You can't undo a delete operation. For this reason, you may want to back up the database before you use any of these statements so you can restore it if necessary.
- You should never use these statements on a production database without first consulting the DBA.

How to alter a table

Figure 11-11 presents the basic syntax of the ALTER TABLE statement. You can use this statement to modify an existing table in one of several ways. The clauses shown here are the ones you're most likely to use.

The first example in this figure shows how to add a new column to a table. As you can see, you code the column definition the same way you do when you create a new table: You specify the column name, followed by its data type and its attributes.

The second example shows how to drop an existing column. Note that SQL Server prevents you from dropping some columns. For example, you can't drop a column if it's the primary key column, if it's used in a check constraint or a foreign key constraint, or if an index is based on it.

The third and fourth examples show how to add constraints to a table. The third example adds a check constraint, and the fourth example adds a foreign key constraint. You can use the same technique to add a primary key or unique constraint. Note that you use this technique regardless of whether the constraint refers to a single column or to two or more columns. That's because the ALTER COLUMN clause only lets you change the data type or the NULL or NOT NULL attribute of an existing column. You can't use it to add column constraints.

When you add a table constraint, SQL Server automatically checks that existing data meets the constraint. If that's not what you want, you can include the WITH NOCHECK keywords in the ALTER statement. This is illustrated in the third example.

In addition to adding constraints, you can use the ALTER TABLE statement to delete constraints. To do that, you have to know the name of the constraint. Although you can name a constraint when you create it, you don't usually do that. That's why I didn't include that information in the syntax for creating constraints. Instead, you usually let SQL Server generate a constraint name for you. Then, if you need to delete the constraint, you can use the Management Studio as described in the next chapter to find out what name SQL Server assigned to it.

The last example shows how to modify the data type of an existing column. In this case, a column that was defined as VARCHAR(100) is changed to VARCHAR(200). Because the new data type is wider than the old data type, you can be sure that the existing data will still fit. However, that's not always the case. Because of that, SQL Server checks to be sure that no data will be lost before it changes the data type. If the change will result in a loss of data, it's not allowed.

The basic syntax of the ALTER TABLE statement

```
ALTER TABLE table_name [WITH CHECK|WITH NOCHECK]
{ADD new_column_name data_type [column_attributes] |
 DROP COLUMN column_name |
 ALTER COLUMN column_name new_data_type [NULL|NOT NULL] |
 ADD [CONSTRAINT] new_constraint_definition |
 DROP [CONSTRAINT] constraint_name}
```

Examples of the ALTER TABLE statement

A statement that adds a new column

```
ALTER TABLE Vendors
ADD LastTranDate DATE NULL;
```

A statement that drops a column

```
ALTER TABLE Vendors
DROP COLUMN LastTranDate;
```

A statement that adds a new check constraint

```
ALTER TABLE Invoices WITH NOCHECK
ADD CHECK (InvoiceTotal >= 1);
```

A statement that adds a foreign key constraint

```
ALTER TABLE InvoiceLineItems WITH CHECK
ADD FOREIGN KEY (AccountNo) REFERENCES GLAccounts(AccountNo);
```

A statement that changes the data type of a column

```
ALTER TABLE InvoiceLineItems
ALTER COLUMN InvoiceLineItemDescription VARCHAR(200);
```

Description

- You use the ALTER TABLE statement to modify an existing table. You can use this statement to add columns or constraints, drop columns or constraints, or change the definition of an existing column, including changing the column's data type.
- Before SQL Server changes the data type of a column, it checks to be sure that no data will be lost. If it will, the operation isn't performed.
- You can modify a column to allow null values as long as the column isn't defined as the primary key. You can modify a column so it doesn't allow null values as long as none of the existing rows contain null values in that column.
- You can add a column that doesn't allow null values only if you specify a default value for that column.
- To delete a constraint, you must know its name. If you let SQL Server generate the name for you, you can use the Management Studio as shown in the next chapter to look up the name.
- By default, SQL Server verifies that existing data satisfies a new check or foreign key constraint. If that's not what you want, you can code the WITH NOCHECK keywords.

Warning

- You should never alter a table in a production database without first consulting the DBA.

How to work with sequences

A *sequence* is a type of database object introduced with SQL Server 2012 that automatically generates a sequence of integer values. Because you can use the IDENTITY attribute for the primary key of a table to generate a simple sequence of numbers that starts with 1 and is incremented by 1, you won't typically use a sequence for that purpose. Instead, you'll use a sequence only if you want to generate a more complex sequence of numbers or if you want to share the sequence between multiple tables.

How to create a sequence

Figure 11-12 shows how to create a sequence. Most of the time, you can create a sequence by coding the CREATE SEQUENCE statement followed by the name of the sequence and the starting value. In the first example, for instance, the CREATE SEQUENCE statement creates a sequence named TestSequence1. This sequence starts with a value of 1, is incremented by a value of 1, has minimum and maximum values that are determined by the data type (bigint by default). In addition, this sequence doesn't cycle back to the minimum value when it reaches the last number in the sequence, and it doesn't cache any sequence numbers. (Although CACHE is the default, a cache size must be specified to use it.) In most cases, these settings are adequate.

If you need to create a sequence that works differently, you can use any of the other clauses of the CREATE SEQUENCE statement to modify the sequence. For example, you can use the INCREMENT BY clause as shown in the second example to increment the sequence numbers by a value other than 1. Although you'll typically code a positive increment value to create an ascending sequence, you can also code a negative value to create a descending sequence.

The third example shows how to create a sequence using all of the optional clauses. This example generates a sequence of int values that begins with 100, is incremented by a value of 10, has a minimum value of 0, has a maximum value of 1,000,000, stores 10 values in the cache at a time, and cycles back to the beginning of the sequence when it reaches the end. Note that a sequence cycles back to the minimum value for the sequence (the maximum value for a descending sequence) even if a starting value is specified. Because of that, you'll want to be sure to specify a minimum value when you use the CYCLE keyword.

How to use a sequence

Once you've created a sequence, you can use it in a variety of ways. For example, you can use it to specify the default value for a column in a table. You can also use it in an INSERT statement as the value for a column. This is illustrated in the second set of examples in figure 11-12. Here, the first example creates a table that contains an int column named SequenceNo. Then, the second example inserts two rows into the table. Here, the NEXT VALUE FOR function

How to create a sequence

The syntax of the CREATE SEQUENCE statement

```
CREATE SEQUENCE sequence_name
    [AS integer_type]
    [START WITH starting_integer]
    [INCREMENT BY increment_integer]
    [{MINVALUE minimum_integer | NO MINVALUE}]
    [{MAXVALUE maximum_integer | NO MAXVALUE}]
    [{CYCLE|NOCYCLE}]
    [{CACHE cache_size|NOCACHE}]
```

A statement that creates a sequence that starts with 1

```
CREATE SEQUENCE TestSequence1
    START WITH 1;
```

A statement that specifies a starting value and an increment for a sequence

```
CREATE SEQUENCE TestSequence2
    START WITH 10
    INCREMENT BY 10;
```

A statement that specifies all optional parameters for a sequence

```
CREATE SEQUENCE TestSequence3
    AS int
    START WITH 100 INCREMENT BY 10
    MINVALUE 0 MAXVALUE 1000000
    CYCLE CACHE 10;
```

How to use a sequence

A statement that creates a test table

```
CREATE TABLE SequenceTable(
    SequenceNo      INT,
    Description     VARCHAR(50));
```

Statements that get the next value for a sequence

```
INSERT INTO SequenceTable
VALUES (NEXT VALUE FOR TestSequence3, 'First inserted row')
INSERT INTO SequenceTable
VALUES (NEXT VALUE FOR TestSequence3, 'Second inserted row');
```

A statement that gets the current value of the sequence

```
SELECT current_value FROM sys.sequences WHERE name = 'TestSequence3';
```

current_value
110

Description

- You use the CREATE SEQUENCE statement to generate integer values for a column in one or more tables.
- By default, the CREATE SEQUENCE statement creates a sequence with the bigint data type that starts with the minimum value for the data type, is incremented by a value of 1, has minimum and maximum values based on the data type, doesn't specify the cache size, and doesn't restart the sequence when the end is reached.
- You can use the NEXT VALUE FOR function to get the next value in the sequence.
- You can query the sys.sequences table to get information about a sequence.

Figure 11-12 How to create and use a sequence

gets the next value from the sequence named TestSequence3 so it can be inserted into the SequenceNo column.

The last example in figure 11-12 shows how to use the sys.sequences catalog view to get information about a sequence. You'll learn about catalog views in chapter 13. For now, just realize that the SELECT statement shown here retrieves the current value for the sequence from this view. If you review the starting value and the increment for the sequence, you'll see how this works.

How to delete a sequence

When you delete a table, the sequences that are used by the table aren't deleted since they're independent of any table. As a result, if you want to delete a sequence, you must use the DROP SEQUENCE statement shown in figure 11-13. Here, the first example drops the sequence named TestSequence2 that was created in figure 11-12.

How to alter a sequence

Once you've created a sequence, you can use the ALTER SEQUENCE statement to alter the attributes of the sequence as shown in figure 11-13. This statement is similar to the CREATE SEQUENCE statement. The two differences are that you can't change the data type for a sequence, and you use the RESTART clause to set a new starting number for the sequence. In addition, you can't set the minimum and maximum values so they don't make sense. For example, if the starting value of the sequence is 1, you can't set the minimum value to 2 without resetting the starting value so it's greater than or equal to 2. Similarly, if the current value of the sequence is 99, you can't set the maximum value to 98 without resetting the starting value so it's less than or equal to 98.

The syntax of the DROP SEQUENCE statement

```
DROP SEQUENCE sequence_name1[, sequence_name2]...
```

A statement that drops a sequence

```
DROP SEQUENCE TestSequence2;
```

The syntax of the ALTER SEQUENCE statement

```
ALTER SEQUENCE sequence_name
  [RESTART [WITH starting_integer]]
  [INCREMENT BY increment_integer]
  [{MINVALUE minimum_integer | NO MINVALUE}]
  [{MAXVALUE maximum_integer | NO MAXVALUE}]
  [{CYCLE|NOCYCLE}]
  [{CACHE cache_size|NOCACHE}]
```

A statement that alters a sequence

```
ALTER SEQUENCE TestSequence1
  INCREMENT BY 9
  MINVALUE 1 MAXVALUE 999999
  CACHE 9
  CYCLE;
```

Description

- You can use the DROP SEQUENCE statement to delete a sequence. A sequence can't be deleted if it's used as the default value for a column.
- You can use the ALTER SEQUENCE statement to alter the attributes of a sequence. However, you can't change the data type, and you can't set the minimum and maximum values so they don't make sense based on existing values.
- If you omit the WITH value of the RESTART clause, the sequence is restarted based on the current definition of the sequence.

Figure 11-13 How to delete and alter a sequence

How to work with collations

So far, this book has assumed that you’re using the default collation for SQL Server. Now, you’ll learn more about collations and why you might want to use a collation that’s different from the default. But first, you need to learn a little about encodings.

An introduction to encodings

A *character set* refers to a set of characters and their numeric codes. An *encoding* refers to the representation of these numeric codes. For example, *Unicode* is a character set that defines numeric codes for characters from most of the world’s writing systems, and *UTF-8* and *UTF-16* are encodings of Unicode that provide two different ways to represent these characters.

Figure 11-14 begins by presenting four encodings that are commonly used by SQL Server. The *Latin1* encoding, technically known as ISO/IEC 8859-1, is the default encoding used for the *char* and *varchar* types. It uses 1 byte per character to represent 256 of the most commonly used characters in Western European languages. That includes the 128 characters in the older *ASCII* character set.

The *UCS-2* encoding is the default encoding used for the *nchar* and *nvarchar* types. It uses 2 bytes per character and can be used to represent the first 65,536 Unicode characters.

SQL Server 2012 introduced support for the *supplementary characters* provided by Unicode that go beyond the original 65,536 characters. To provide for these characters in *nchar* and *nvarchar* types, SQL Server uses the *UTF-16* encoding. This encoding uses 4 bytes per character to store supplementary characters such as emojis.

SQL Server 2019 introduced support for the *UTF-8* encoding, which can be used to store Unicode characters, including the supplementary characters, in *char* and *varchar* data types. This encoding requires from 1 to 4 bytes per character and is widely used when working with the Internet.

The second table in this figure should help you understand the differences in the bytes required to store characters using the three Unicode encodings. To store an ASCII character, for example, *UTF-8* requires 1 byte, but *UCS-2* and *UTF-16* require 2 bytes. On the other hand, to store Asian ideographs, *UTF-8* requires 3 bytes but *UCS-2* and *UTF-16* require only two. Both *UTF-8* and *UTF-16* require 4 bytes to store supplementary characters, and these characters aren’t supported by *UCS-2*. This is important to keep in mind when you decide what collation to use.

Common character encodings

Name	Bytes	Supported characters
Latin1	1	The letters, digits, and punctuation for most Western European languages.
UCS-2	2	The first 65,536 Unicode characters, with 2 bytes for each character.
UTF-8	1-4	All Unicode characters, with 1 to 3 bytes for the first 65,536 characters and 4 bytes for the supplementary characters.
UTF-16	2 or 4	All Unicode characters, with 2 bytes for the first 65,536 characters and 4 bytes for the supplementary characters.

Bytes required for Unicode characters

Numeric code range	Characters	UCS-2	UTF-8	UTF-16
0-127	ASCII	2	1	2
128-2047	European letters and Middle Eastern script	2	2	2
2048-65,535	Korean, Chinese, and Japanese ideographs	2	3	2
65,536-1,114,111	Supplementary characters	N/A	4	4

Description

- *Character sets* and *encodings* map a numeric code to each character. In practice, these terms are used interchangeably.
- One byte consists of eight *bits*, which can be combined in 256 different ways. Two bytes, or 16 bits, can be combined in 65,536 different ways.
- Most systems use the same numeric codes for the first 128 characters. These are the codes defined by the *ASCII (American Standard Code for Information Interchange)* character set.
- The *Latin1* (ISO/IEC 8859-1) encoding starts with the 128 ASCII characters and adds another 128 characters commonly used by Western Europe languages.
- *Unicode* begins by defining 65,536 Unicode characters from most of the world's writing systems. The first 256 of these characters are the ones in the Latin1 character set.
- To work with Unicode characters, you use an encoding such as *UCS-2*, *UTF-8*, or *UTF-16*. All three of these encodings allow you to access the first 65,536 Unicode characters.
- Unicode also defines *supplementary characters* that go beyond the first 65,536 characters. SQL Server 2012 and later provide support for these characters, and you can work with them using either the UTF-8 encoding with the char or varchar data type (SQL Server 2019 or later) or the UTF-16 encoding with the nchar or nvarchar data type.
- The Latin1 and UCS-2 encodings are known as *fixed-length encodings*, since they always use the same number of bytes per character.
- The UTF-8 and UTF-16 encodings are known as *variable-length encodings*, since the number of bytes per character varies depending on the character.

Figure 11-14 An introduction to character sets and encodings

An introduction to collations

A *collation* determines how character data is stored in a server, database, or column and how that data can be sorted and compared. You specify a collation by coding the collation name, followed by one or more collation options.

SQL Server's default collation, for example, uses _Latin1 to specify that the Latin1 encoding should be used for the char and varchar types and the UCS-2 encoding should be used for the nchar and nvarchar types. Then, the _CI option specifies that sorting should be case-insensitive. This means that SQL Server sorts uppercase letters such as A and lowercase letters such as a at the same level, which is usually what you want. Finally, the _AS option specifies that character accents are used when sorting. This means that Latin letters that have accents such as Á are not sorted at the same level as characters such as A that don't have accents, which is usually what you want.

Figure 11-15 begins by describing three collation sets supported by SQL Server. Then, it describes some of the collation options. If SQL Server isn't storing or sorting characters the way you want, you can change the collation as described in figure 11-17.

If you're creating a new database, for example, you probably want to use a collation from the Windows set, not the SQL Server set. That's because the collations in the Windows set are newer and provide sorting that's compatible with Windows sorting. In most cases, you only want to use a collation from the SQL Server set for backwards compatibility with existing SQL Server databases.

When sorting, you may want to include one or more of the sorting options. To perform a case-sensitive sort, you can use a collation that includes the _CS option. Similarly, to perform an accent-sensitive sort, you can use a collation that includes the _AS option. Or, to perform a sort as quickly as possible, you can use a collation that includes the _BIN or _BIN2 option. Here, *bin* stands for *binary*, and it means that SQL Server sorts characters by their numeric codes. This results in a case-sensitive and accent-sensitive sort.

When specifying the collation, the _SC and _UTF8 options change how SQL Server stores data. To start, the _SC option allows supplementary characters to be stored in the nchar and nvarchar types using UTF-16 encoding. Similarly, using both the _SC and _UTF8 options allows Unicode characters, including supplementary characters, to be stored in the char and varchar types using UTF-8 encoding.

Note that if you use the _SC or the _SC and _UTF8 collation options, your string data types may not be able to store as many characters as they would without these options. That's because, as you learned in the last figure, many of the Unicode characters require more than one byte with UTF-8 encoding, and all of the supplementary characters require four bytes with both UTF-8 and UTF-16 encoding. When you use these collation options, then, you need to be sure to declare string columns with the appropriate size. You'll learn more about that in a minute.

Collation sets supported by SQL Server

Set	Description
Windows	Collations in this set store data based on an associated Windows system locale and provide sorting that's completely compatible with Windows.
SQL Server	The collations in this set have a prefix of SQL_. They provide sorting that's not compatible with Windows for non-Unicode data. This set is older than the Windows set and is provided for backwards compatibility with existing SQL Server databases.
Binary	The collations in this set have a suffix of _BIN or _BIN2. They can only be used for sorting. Because they sort by numeric code, they always perform a case-sensitive and accent-sensitive sort. They are also the fastest collations for sorting.

Some collation options

Option	Name	Description
_CS	Case Sensitive	Case is used when sorting.
_CI	Case Insensitive	Case is not used when sorting. This is the default.
_AS	Accent Sensitive	Character accents are used when sorting.
_AI	Accent Insensitive	Character accents are not used when sorting. This is the default.
_BIN	Binary (legacy)	Sorts based on the numeric codes of the characters.
_BIN2	Binary (new)	Sorts based on the Unicode codes of the characters.
_SC	Supplementary Characters	Allows supplementary characters to be stored. Can be used only with version 90 and 100 Windows collations. Version 140 collations automatically provide for supplementary characters.
_UTF8	UTF-8	Uses UTF-8 for the char and varchar data types.

Collation examples

The default server collation for the English (United States) locale

`SQL_Latin1_General_CI_AS`

The closest Windows equivalent to this collation

`Latin1_General_100_CI_AS`

A collation that provides for supplementary characters using UTF-8

`Latin1_General_100_CI_AS_SC_UTF8`

Description

- A *collation* determines how character data is stored in a server, database, or column and how that data can be sorted and compared. You specify a collation by coding the collation name, followed by one or more collation options.
- For the Latin1_General_100_CI_AS collation, the char and nchar types use the 1-byte Latin1 character set, and the nchar and nvarchar types use the 2-byte UCS-2 encoding to access most of the Unicode characters.
- With SQL Server 2012 and later, you can use a collation that supports supplementary characters. Then, the nchar and nvarchar types use UTF-16 encoding, not UCS-2.
- With SQL Server 2019 and later, you can use a collation that supports UTF-8 encoding. Then, the char and varchar types use UTF-8 encoding, not Latin1.

Figure 11-15 An introduction to collations

How to view collations

Figure 11-16 starts by showing how to view the default collation for a server. Here, the first example shows the collation that's the default for many locales, including the United States.

The second example shows how to view all collations that are available on the current server. To do that, you can use a SELECT statement to view all rows and columns returned by a system function named FN_HELPCOLLATIONS. As the result set shows, this function returns the name and a description for each collation.

The third example shows how to view collations with a specific name. To do that, you can add a LIKE clause that uses the % wildcard character to limit the number of rows that are returned. Here, the SELECT statement uses the % wildcard character to return all rows that have a name that starts with Latin1_General_100.

The fourth example shows how to view the collation for a specific database. To do that, you can use a SELECT statement that queries the data that's returned by the sys.databases catalog view. As you'll learn in chapter 13, you can use catalog views to retrieve information about database objects. In this example, the SELECT statement displays the name of the database and its collation where the database name is AP. Here, the result set shows that the collation for the AP database has been changed from the collation used by the server. You'll learn how to specify a different collation in the next figure.

The fifth example shows how to view the collations for each column in a table. To do that, you can query the tables that are returned by the sys.tables and sys.columns catalog views. Here, the SELECT statement returns the table name, column name, and collation for each column in the current database. In this figure, the current database happens to be the AP database, but this SELECT statement works for any database. This result set shows that a collation is assigned to each string column, but not to columns that store other data types such as numbers and dates.

Now that you have a basic understanding of collations, you may wonder which collation you should use. For new development, it's generally considered a good practice to use a collation from the Windows set, not one from the SQL Server set, which is provided mainly for backwards compatibility.

Beyond that, the answer depends on the nature of the string data that you need to store in your database. If your string data consists entirely of characters from Western European languages, using the Latin1 encoding allows you to minimize storage by using the char and varchar types to store these characters using 1 byte per character. If your string data consists mostly of characters from Western European languages but you need to support Unicode characters too, using UTF-8 encoding allows you to minimize storage. However, if your string data contains a high percentage of characters that are Asian ideographs, using the UCS-2 or UTF-16 encodings and the nchar and nvarchar types minimizes storage requirements by using 2 bytes for Asian ideographs instead of the 3 bytes required by UTF-8. Of course, if you use UTF-8 or UTF-16, you need to make sure your columns specify enough bytes to avoid data truncation errors.

How to view the default collation for a server

```
SELECT CONVERT(varchar, SERVERPROPERTY('collation'));
```

(No column name)
1 SQL_Latin1_General_CI_AS

How to view all available collations for a server

```
SELECT * FROM sys.fn_helpcollations();
```

name	description
3137 Latin1_General_100_CI_AS_SC	Latin1-General-100, case-insensitive, accent-sensitive,...
3138 Latin1_General_100_CI_AS_WS_SC	Latin1-General-100, case-insensitive, accent-sensitive,...
3139 Latin1_General_100_CI_AS_KS_SC	Latin1-General-100, case-insensitive, accent-sensitive,...
3140 Latin1_General_100_CI_AS_KS_WS_SC	Latin1-General-100, case-insensitive, accent-sensitive,...
3141 Latin1_General_100_CS_AI_SC	Latin1-General-100, case-sensitive, accent-insensitive,...
3142 Latin1_General_100_CS_AI_WS_SC	Latin1-General-100, case-sensitive, accent-insensitive,...
3143 Latin1_General_100_CS_AI_KS_SC	Latin1-General-100, case-sensitive, accent-insensitive,...
3144 Latin1_General_100_CS_AI_KS_WS_SC	Latin1-General-100, case-sensitive, accent-insensitive,...
3145 Latin1_General_100_CS_AS_SC	Latin1-General-100, case-sensitive, accent-sensitive, ...
3146 Latin1_General_100_CS_AS_WS_SC	Latin1-General-100, case-sensitive, accent-sensitive, ...

How to view all collations with a specific name

```
SELECT * FROM sys.fn_helpcollations()
WHERE name LIKE 'Latin1_General_100%';
```

How to view the collation for a database

```
SELECT name, collation_name
FROM sys.databases
WHERE name = 'AP';
```

name	collation_name
1 AP	Latin1_General_100_CI_AS

How to view the collations for the columns in a database

```
SELECT sys.tables.name AS TableName, sys.columns.name AS ColumnName,
       collation_name
  FROM sys.columns inner join sys.tables
    ON sys.columns.object_id = sys.tables.object_id;
```

TableName	ColumnName	collation_name
31 Terms	TermID	NULL
32 Terms	TermDescription	Latin1_General_100_CI_AS
33 Terms	TermsDueDays	NULL
34 Vendors	VendorID	NULL
35 Vendors	VendorName	Latin1_General_100_CI_AS
36 Vendors	VendorAddress1	Latin1_General_100_CI_AS
37 Vendors	VendorAddress2	Latin1_General_100_CI_AS
38 Vendors	VendorCity	Latin1_General_100_CI_AS

Note

- When you install SQL Server, the default collation is based on the operating system locale.

Figure 11-16 How to view collations

How to specify a collation

Figure 11-17 shows how to specify a collation at three levels: database, column, and expression. In most cases, you want to specify the collation at the database level as shown in the first example. Then, all the columns that store string data are defined with that collation. If necessary, though, you can override collation that's set at the database level by setting it at the column or expression level as shown by the second and third examples.

To specify a collation, you use the COLLATE clause. For a new database or table, you add this clause to the CREATE statement for the database or columns in the table. For an existing database or table, you add these clauses to the ALTER statement for the database or columns in the table.

When you specify a collation for a database, it sets the encoding that's used for the columns of the database that store string data. In the first example, the CREATE DATABASE statement specifies a collation that uses Latin1 encoding for char and varchar types and UCS-2 encoding for nchar and nvarchar types. Then, the ALTER DATABASE statement specifies a collation that uses UTF-8 encoding for char and varchar types and UTF-16 encoding for nchar and nvarchar types.

The second example works similarly, except that the collation only applies to the EmployeeName column, not the entire AR database. In other words, the column-level collation overrides the database-level collation.

The third example specifies an expression-level collation for an ORDER BY clause of a SELECT statement. More specifically, it uses the _BIN2 option to specify a binary collation. Binary collations provide the fastest sorting, but you can only use them in expressions such as this ORDER BY clause. In other words, you can't use them at the database or column level.

If you switch from Latin1 to UTF-8, you may need to quadruple the maximum number of bytes for any char and varchar types impacted by the new collation. Otherwise, these data types may not be able to support as many characters as they did previously. For example, if you attempt to store characters that use more than 1 byte per character, you may get a truncation error like the one shown in the fourth example. Here, the Message column was defined with the varchar(10) type, so it can store 10 bytes. Then, the user attempted to insert 7 characters, but the last three characters are emojis. Because emojis require 4 bytes per character, the string requires a total of 16 bytes ($4 \times 1 + 3 \times 4$), which causes an error.

In the second example, the CREATE TABLE statement uses the Latin1 encoding for the EmployeeName column that's defined with the varchar(50) type. Then, when the ALTER TABLE statement changes the collation for this column to one that uses UTF-8, it also changes the data type to varchar(200). That way, this column can store at least 50 characters, even if all characters use 4 bytes.

The same principles apply if you switch from UCS-2 to UTF-16. In this case, however, you only need to double the maximum number of bytes for any nchar and nvarchar types impacted by the new collation. That's because a column with the nchar(10) type can store 20 bytes. As a result, to make sure you don't get a truncation error, you only need to double it to nchar(20) so it can store 40 bytes.

The clause used to specify a collation

[COLLATE collation]

How to specify a collation at the database level

For a new database

```
CREATE DATABASE AR COLLATE Latin1_General_100_CI_AS;
```

For an existing database

```
ALTER DATABASE AR COLLATE Latin1_General_100_CI_AS_SC_UTF8;
```

How to specify a collation at the column level

For a column in a new table

```
CREATE TABLE Employees
(
    EmployeeID      INT          PRIMARY KEY,
    EmployeeName    VARCHAR(50)   COLLATE Latin1_General_100_CI_AS
);
```

For a column in an existing table

```
ALTER TABLE Employees
ALTER COLUMN EmployeeName VARCHAR(200)
                           COLLATE Latin1_General_100_CI_AS_SC_UTF8;
```

How to specify a collation for an expression

```
SELECT * FROM Employees
ORDER BY EmployeeName COLLATE Latin1_General_100_BIN2;
```

A possible error message after switching to UTF-8 or UTF-16

```
String or binary data would be truncated in table 'Test', column 'Message'.
Truncated value: 'Hi! 😊😊😊'.
```

Description

- You can use the COLLATE clause to set the collation at the database level, at the column level, or for an expression in an ORDER BY clause.
- The default collation for a database is the collation for the server, and the default collation for a column in a table is the collation for the database.
- You can use a binary collation only to specify that a binary sort is to be performed on an expression.
- To use UTF-8 encoding, a collation must also provide for supplementary characters.
- If you switch from Latin1 to UTF-8, you may need to quadruple the maximum number of bytes for any char and varchar types impacted by the new collation. Otherwise, these data types may not be able to support as many characters as they did previously.
- If you switch from UCS-2 to UTF-16, you may need to double the maximum number of bytes for any nchar and nvarchar types impacted by the new collation. Otherwise, these data types may not be able to support as many characters as they did previously.

The script used to create the AP database

To complete this chapter, figure 11-18 presents the DDL statements that I used to create the AP database that's used in the examples throughout this book. By studying these DDL statements, you'll get a better idea of how a database is actually implemented. Note, however, that these statements are coded as part of a script. So before I describe the DDL statements, I'll introduce you to scripts.

How the script works

In this figure, all of the DDL statements are coded as part of a *script*, which consists of one or more SQL statements that are stored in a file. This is typically the way that all of the objects for a database are created. In chapter 14, you'll learn the details of coding scripts, but here are some basic concepts.

A script consists of one or more *batches*. The script shown in the two parts of this figure, for example, consists of two batches. Each batch consists of one or more SQL statements that are executed as a unit. To signal the end of a batch and execute the statements it contains, you use the GO command. As you can see, then, the first batch shown in this figure consists of a single CREATE DATABASE statement, and the second batch consists of several CREATE TABLE and CREATE INDEX statements. Notice that a GO command isn't required at the end of the second batch, which is the last batch in this script.

To create and execute a script, you can use the Management Studio. Although you may not be aware of it, you're creating a script each time you enter a SQL statement into the Management Studio. So far, though, the scripts you've created have consisted of a single batch.

The reason for breaking a script like the one shown here into batches is that some of the statements must be executed before others can execute successfully. Before any tables can be created in the AP database, for example, the database itself must be created.

The only other statement used in this script that you're not familiar with is the USE statement. You use this statement to change the current database. That way, after the script creates the AP database, the statements that follow will operate on that database rather than on the one that's selected in the Management Studio toolbar.

How the DDL statements work

Notice that each CREATE TABLE statement in this script lists the primary key column (or columns) first. Although this isn't required, it's a conventional coding practice. Also note that the order in which you declare the columns defines the default order for the columns. That means that when you use a SELECT * statement to retrieve all of the columns, they're returned in this order. For that reason, you'll want to define the columns in a logical sequence.

The SQL script that creates the AP database**Page 1**

```
CREATE DATABASE AP;
GO

USE AP;
CREATE TABLE Terms
(TermsID           INT          NOT NULL PRIMARY KEY,
TermsDescription  VARCHAR(50)   NOT NULL,
TermsDueDays      SMALLINT    NOT NULL);

CREATE TABLE GLAccounts
(AccountNo         INT          NOT NULL PRIMARY KEY,
AccountDescription VARCHAR(50) NOT NULL);

CREATE TABLE Vendors
(VendorID          INT          NOT NULL IDENTITY PRIMARY KEY,
VendorName        VARCHAR(50)  NOT NULL,
VendorAddress1    VARCHAR(50)  NULL,
VendorAddress2    VARCHAR(50)  SPARSE NULL,
VendorCity        VARCHAR(50)  NOT NULL,
VendorState       CHAR(2)      NOT NULL,
VendorZipCode     VARCHAR(20)  NOT NULL,
VendorPhone       VARCHAR(50)  NULL,
VendorContactLName VARCHAR(50) NULL,
VendorContactFName VARCHAR(50) NULL,
DefaultTermsID    INT          NOT NULL
                           REFERENCES Terms(TermsID),
DefaultAccountNo  INT          NOT NULL
                           REFERENCES GLAccounts(AccountNo));
```

Basic script concepts

- Instead of creating database objects one at a time, you can write a *script* that contains all of the statements needed to create the database and its tables and indexes.
- A script is a set of one or more *batches* that can be stored in a file. A batch is a sequence of SQL statements that are executed as a unit. You can use the Management Studio to create and execute script files.
- The GO command signals the end of the batch and causes all of the statements in the batch to be executed. You should issue a GO command when the execution of the next statement depends on the successful completion of the previous statements.
- SQL Server executes the last batch in a script automatically, so a final GO command isn't required.
- To change the current database within a script, you use the USE statement.

Note

- The Terms and GLAccounts tables are created first so the other tables can define foreign keys that refer to them. Similarly, the Vendors table is created before the Invoices table, and the Invoices table is created before the InvoiceLineItems table (see part 2).

Figure 11-18 The script used to create the AP database (part 1 of 2)

Also notice that most of the columns in this database are assigned the NOT NULL constraint. The exceptions are the VendorAddress1, VendorAddress2, VendorPhone, VendorContactLName, and VendorContactFName columns in the Vendors table and the PaymentDate column in the Invoices table. Because not all vendor addresses will require two lines and because some vendors won't provide a street address at all, a null value can be assigned to both address columns to indicate that they're not applicable. Similarly, you may not have a phone number and contact information for each vendor. For this reason, you could assign a null value to one of these columns to indicate an unknown value. Finally, an invoice wouldn't be assigned a payment date until it was paid. Until that time, you could assign a null value to the PaymentDate column to indicate that it hasn't been paid.

I could also have used a default date to indicate an unpaid invoice. To do that, I could have defined the PaymentDate column like this:

```
PaymentDate DATE NOT NULL DEFAULT '1900-01-01'
```

In this case, the date January 1, 1900 would be stored in the PaymentDate column unless another value was assigned to that column. Usually, a null value is a more intuitive representation of an unknown value than a default such as this, but either representation is acceptable. Keep in mind, though, that the technique you use will affect how you query the table.

Since at least 60% of the values for the VendorAddress2 column are likely to contain a null value, this column uses the SPARSE attribute to optimize storage of null values. Although this attribute isn't critical, it can significantly reduce the amount of storage required for this column.

The Vendors, Invoices, and InvoiceLineItems tables also define the appropriate foreign key constraints. Notice that the foreign key constraint in the InvoiceLineItems table that defines the relationship to the Invoices table includes the ON DELETE CASCADE clause. That way, if an invoice is deleted, its line items are deleted too.

Because each of the five tables in this database has a primary key, SQL Server creates a clustered index for each table based on that key. In addition, this script creates seven additional indexes to improve the performance of the database. The first five of these indexes are based on the foreign keys that each referring table uses to relate to another table. For example, since the VendorID column in the Invoices table references the VendorID column in the Vendors table, I created a nonclustered index on VendorID in the Invoices table. Similarly, I created indexes for TermsID in the Invoices table, DefaultTermsID and DefaultAccountNo in the Vendors table, and AccountNo in the InvoiceLineItems table. Finally, I created indexes for the VendorName column in the Vendors table and the InvoiceDate column in the Invoices table because these columns are frequently used to search for rows in these tables.

As you may have noticed, I created an index for each column that appears in a foreign key constraint except one: the InvoiceID column in the InvoiceLineItems table. Since this column is part of the composite primary key for this table, it's already included in the clustered index. For this reason, the addition of a nonclustered index on InvoiceID by itself won't improve performance.

The SQL script that creates the AP database**Page 2**

```

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);

CREATE TABLE InvoiceLineItems
(InvoiceID           INT          NOT NULL
                      REFERENCES Invoices(InvoiceID)
                      ON DELETE CASCADE,
InvoiceSequence      SMALLINT    NOT NULL,
AccountNo            INT          NOT NULL
                      REFERENCES GLAccounts(AccountNo),
InvoiceLineItemAmount MONEY      NOT NULL,
InvoiceLineItemDescription VARCHAR(100) NOT NULL,
PRIMARY KEY (InvoiceID, InvoiceSequence));

CREATE INDEX IX_Invoices_VendorID
    ON Invoices (VendorID);
CREATE INDEX IX_Invoices_TermsID
    ON Invoices (TermsID);
CREATE INDEX IX_Vendors_TermsID
    ON Vendors (DefaultTermsID);
CREATE INDEX IX_Vendors_AccountNo
    ON Vendors (DefaultAccountNo);
CREATE INDEX IX_InvoiceLineItems_AccountNo
    ON InvoiceLineItems (AccountNo);
CREATE INDEX IX_VendorName
    ON Vendors (VendorName);
CREATE INDEX IX_InvoiceDate
    ON Invoices (InvoiceDate DESC);

```

Notes

- The InvoiceLineItems table has a composite primary key that consists of the InvoiceID and InvoiceSequence columns. For this reason, the PRIMARY KEY constraint must be defined as a table-level constraint.
- In addition to the five indexes that SQL Server automatically creates for the primary key of each table, this script creates seven additional indexes. The first five are indexes for the foreign keys that are used in the REFERENCES constraints. The last two create indexes on the VendorName column and the InvoiceDate column since these columns are used frequently in search conditions.

Figure 11-18 The script used to create the AP database (part 2 of 2)

Perspective

Now that you've completed this chapter, you should be able to create and modify databases, tables, and indexes by coding DDL statements. This provides a valuable background for working with any database. In practice, though, you may sometimes want to use the Management Studio to perform the functions that are done by DDL statements, so that's what you'll learn to do in the next chapter.

Terms

data definition language (DDL)	sequence
database objects	character set
identifier	encoding
transaction log file	Unicode
attach a database	UTF-8
full-table index	UTF-16
filtered index	Latin1
snippet	ASCII (American Standard Code for Information Interchange)
snippet picker	UCS-2
constraint	supplementary characters
column-level constraint	fixed-length encoding
table-level constraint	variable-length encoding
check constraint	collation
foreign key constraint	script
reference constraint	batch
cascading delete	
cascading update	

Exercises

1. Create a new database named Membership.
2. Write the CREATE TABLE statements needed to implement the following design in the Membership database. Include foreign key constraints. Define IndividualID and GroupID as identity columns. Decide which columns should allow null values, if any, and explain your decision. Define the Dues column with a default of zero and a check constraint to allow only positive values.



3. Write the CREATE INDEX statements to create a clustered index on the GroupID column and a nonclustered index on the IndividualID column of the GroupMembership table.
4. Write an ALTER TABLE statement that adds a new column, DuesPaid, to the Individuals table. Use the bit data type, disallow null values, and assign a default Boolean value of False.
5. Write an ALTER TABLE statement that adds two new check constraints to the Invoices table of the AP database. The first should allow (1) PaymentDate to be null only if PaymentTotal is zero and (2) PaymentDate to be not null only if PaymentTotal is greater than zero. The second constraint should prevent the sum of PaymentTotal and CreditTotal from being greater than InvoiceTotal.
6. Delete the GroupMembership table from the Membership database. Then, write a CREATE TABLE statement that recreates the table, this time with a unique constraint that prevents an individual from being a member in the same group twice.
7. Write a SELECT statement that displays the collation name for the collation that's used by the Membership database. If NULL is displayed for the collation name, it means that there isn't an active connection to the Membership database. To fix that, select the Membership database from the Available Databases combo box in the SQL Editor toolbar.
8. Write an ALTER TABLE statement that changes the collation for the GroupName column in the Groups table to Latin1_General_100_CI_AS_SC_UTF8. Be sure to account for extra bytes that may be required to store Unicode characters with the UTF-8 encoding.

How to create a database and its tables with the Management Studio

Now that you've learned how to code all of the essential SQL statements for data definition, you're ready to learn how to use the Management Studio to generate this code for you. The Management Studio makes it easy to perform common tasks, and you'll use it frequently to create, modify, and delete the objects of a database. Once you learn how to use the Management Studio to create or modify the design of a database, you can decide when it makes sense to use it and when it makes sense to code the DDL statements yourself.

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.....	388
How to generate a change script when you modify a table.....	390
Perspective	392

How to work with a database

In chapter 2, you learned how to use the Management Studio to attach the files for an existing database to the SQL Server engine. Now, you'll learn how to use the Management Studio to create a new database from scratch.

How to create a database

Figure 12-1 shows the New Database dialog box that's used to create a new database. To display this dialog box, you can start the Management Studio, right-click on the Databases folder, and select the New Database command. Then, you can use the New Database dialog box to enter a name for the database. When you do, the names for the data and log files are automatically updated in the Database Files pane. In this figure, for example, I entered New_AP for the database name. As a result, the New Database dialog box automatically changed the name for the data file to New_AP, and it changed the name of the log file to New_AP_log. Although it's possible to alter these names, I recommend using this naming convention.

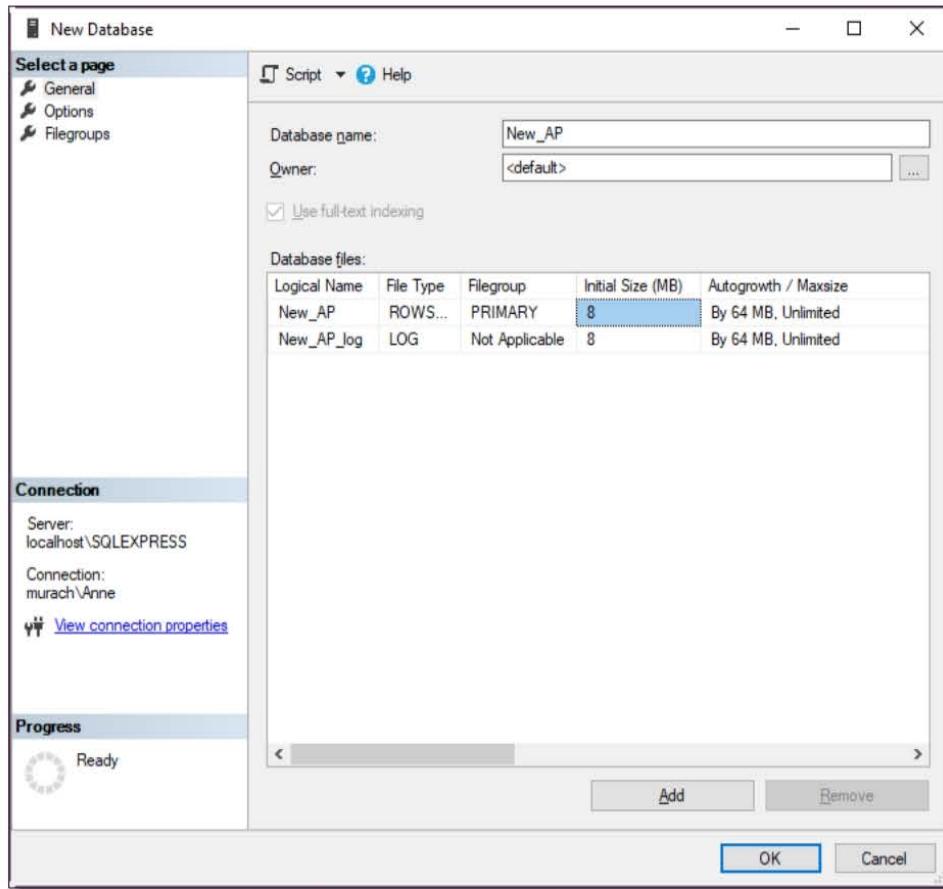
Since the default properties for a database are set correctly for most databases, that's usually all you need to do to create a new database. However, if necessary, you can use the New Database dialog box to change any default properties that are used by SQL Server. If you want to change the initial size for the data file for a database, you can click in the Initial Size column and enter a new initial size. If you want to change the owner of the database from the default owner, you can click on the button to the right of the Owner text box. If you want to change other properties, you can display the Options page and change the properties that are available from this page. And so on.

By default, the data and log files for a new database are stored in the directory shown in this figure, which is usually what you want. If it isn't, you can detach, move, and reattach the database files as described in chapter 2.

How to delete a database

If you want to delete a database, you can right-click on the database in the Management Studio, select the Delete command, and click OK in the resulting dialog box. Keep in mind, however, that this permanently deletes the data and log files for the database. Because of that, you may want to create a backup copy of the database before you delete it as described in chapter 2. That way, you can restore the database later if necessary. Alternatively, you may want to detach the database from the server instead of deleting it. That way, the database files aren't deleted and can be attached again later if necessary.

The New Database dialog box



The default directory for SQL Server 2019 databases

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

How to create a new database

- To create a new database, right-click on the Databases folder in the Management Studio and select the New Database command to display the New Database dialog box shown above. Then, enter a name for the database. This updates the names for the data and log files that are displayed in the Database Files pane. Finally, click OK to create the database and its files.

How to delete a database

- To delete a database, expand the Databases folder, right-click on the database, and select the Delete command to display the Delete Object dialog box. Then, click OK to delete the database and its files.
- If you get an error message indicating that the database can't be deleted because it's in use, select the Close Existing Connections option and click OK again.

Figure 12-1 How to create or delete a database

How to work with tables

In the last chapter, you learned how to work with tables, keys, indexes, and constraints by coding DDL statements. Now you'll learn how to work with these objects by using the Management Studio.

How to create, modify, or delete a table

Figure 12-2 shows how to use the Table Designer to create, modify, or delete a table. To start, this figure shows the Table Designer for a simple version of the Invoices table that only has four columns. Here, each column in the table is listed in the column grid that's at the top of the Table Designer. This grid includes the column name, the data type for the column, and whether the column allows null values. In addition to these properties, you can use the Column Properties pane that appears at the bottom of this window to set the other column properties. In this figure, for example, the InvoiceID column has been defined as an identity column. As you modify column properties, you'll find that the properties that are available for each column change depending on the properties that are specified in the column grid. For example, since the InvoiceDate column isn't of the int type, it can't be specified as an identity column.

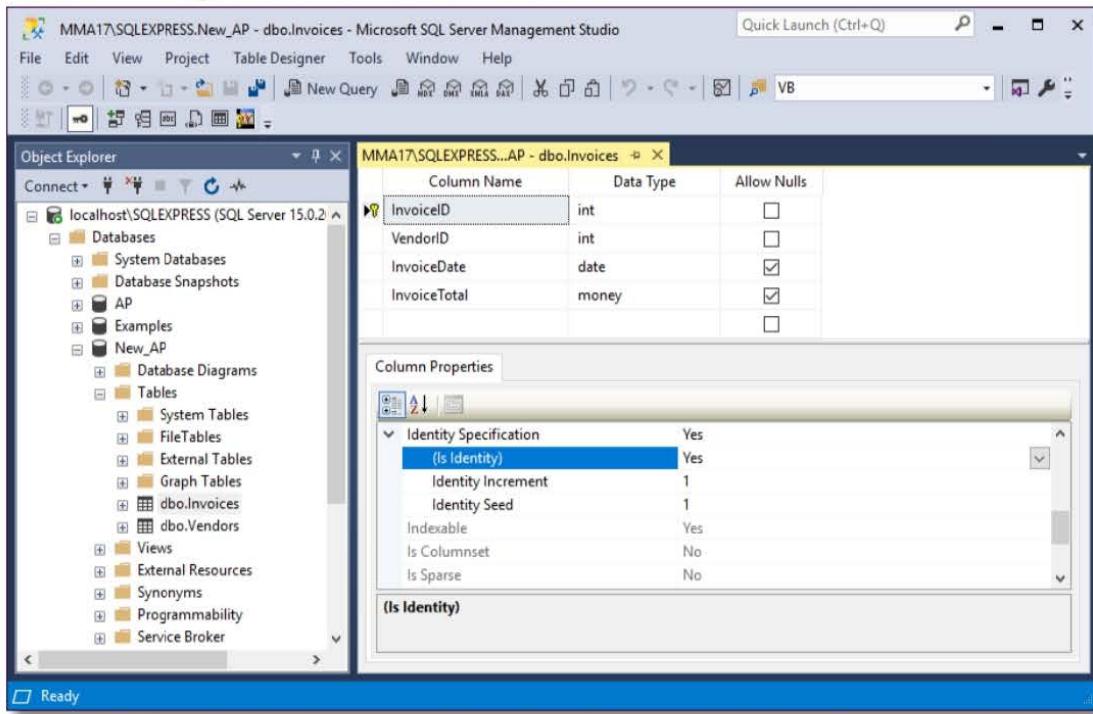
To create a new table, you can right-click on the Tables folder and select the Table command to display a blank table in the Table Designer. Then, you can enter the column names and data types for one or more columns in the table. When you save the table for the first time, the Management Studio will display a dialog box that allows you to enter a name for the table.

To edit the design of an existing table, you can expand the Tables folder, right-click on the table, and select the Design command. This displays the table in the Table Designer. In addition, the Management Studio automatically displays the Table Designer toolbar. You can use this toolbar to work with the keys, relationships, indexes, and constraints defined by a table.

To set the primary key for a table, for example, you can click on the box to the left of the key column to select it and then click on the Set Primary Key button that's available from the Table Designer toolbar. If the key consists of two or more columns, you can hold down the Ctrl key and click multiple columns to select them. When you set the primary key, a key icon appears to the left of the key column or columns. In this figure, for example, a key icon appears to the left of the InvoiceID column.

By default, you can't use the Table Designer to modify a table in such a way that requires dropping and recreating the table. For example, you can't modify the identity column for an existing table. Since dropping the table deletes all data in the table, this isn't usually what you want. However, in some cases, you may want to allow this type of change. To do that, you can pull down the Tools menu, select the Options command, expand the Designers group, select the Table and Database Designers group, and deselect the "Prevent saving changes that require table re-creation" option.

The Table Designer for the Invoices table



How to create or modify the design of a table

- To create a new table, right-click on the Tables folder and select the Table command to display a new table in the Table Designer. Then, when you click on the Save button in the toolbar, you can supply a name for the table.
- To edit the design of an existing table, expand the Tables folder, right-click on the table, and select the Design command to display the table in the Table Designer.
- To set the basic properties for each column, use the grid at the top of the Table Designer to specify the column name, data type, and whether or not the column allows nulls.
- To set other column properties, such as the identity column or a default value, select the column in the grid and use the Column Properties pane.
- To set the primary key, select the column or columns and click the Set Primary Key button in the Table Designer toolbar. Then, a key icon appears to the left of key columns.

How to delete a table

- To delete a table, expand the Tables folder, right-click on the table, select the Delete command, and click the OK button in the Delete Object dialog box that's displayed.

Note

- When you create a table using the Management Studio, the table is automatically stored in the default schema. If you want to transfer the table to a different schema, you can use the ALTER SCHEMA statement. See chapter 17 for details.

Figure 12-2 How to create, modify, or delete a table

How to work with foreign key relationships

Figure 12-3 shows how to specify foreign key relationships between tables. To start, you display the table that you want to contain the foreign key in the Table Designer as shown in figure 12-2. Then, you click on the Relationships button in the Table Designer toolbar to display the Foreign Key Relationships dialog box. To add a new foreign key relationship, you click on the Add button. This causes a relationship with a default name such as FK_Invoices_Invoices to be added to the list box on the left side of the dialog box.

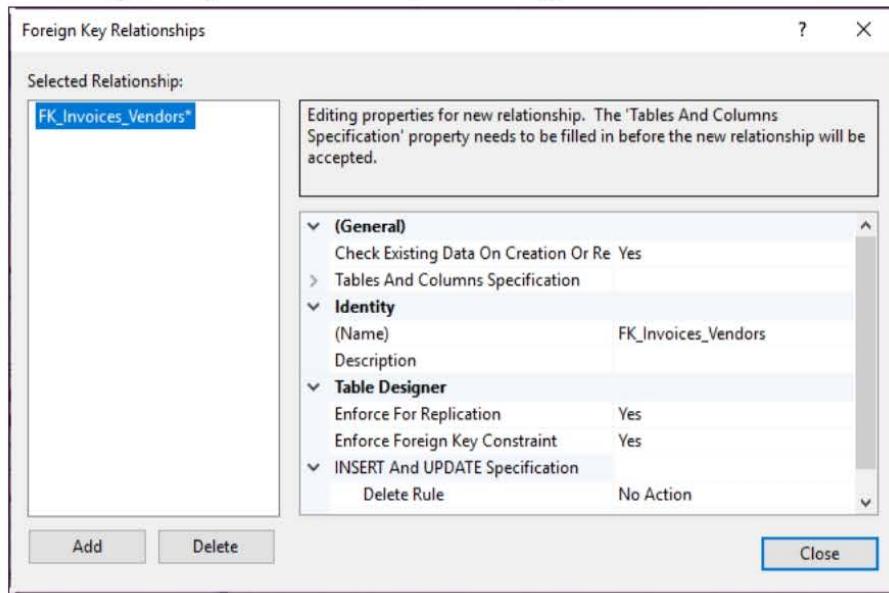
To specify the primary key table and the columns for a relationship, you use the Tables and Columns dialog box shown in this figure. Here, I specified that the VendorID column should be used as the foreign key relationship between the Invoices and Vendors tables. When I specified this relationship, the Tables and Columns dialog box automatically changed the name of the relationship to the more meaningful name of FK_Invoices_Vendors.

You can also use the Foreign Key Relationships dialog box to control how the foreign key constraint is enforced. In this figure, for example, the Enforce Foreign Key Constraint property is set to Yes so the referential integrity between these two tables will be maintained. If this property was set to No, SQL Server would recognize but not enforce the relationship. In most cases, then, you'll want to be sure this property is set to Yes.

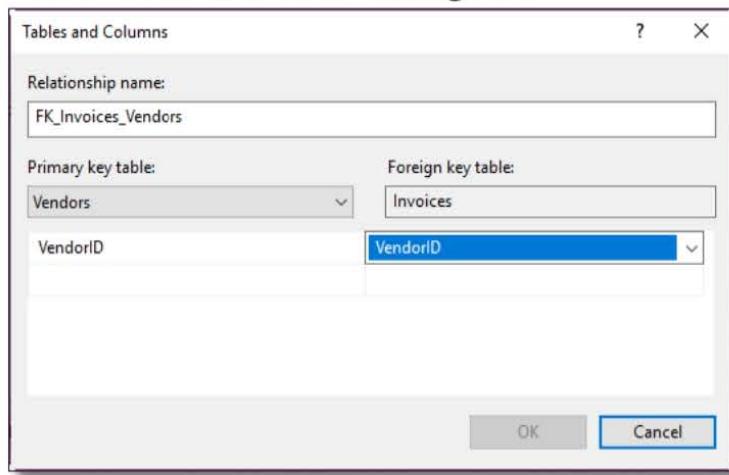
You'll also want to be sure that the Delete Rule and Update Rule properties are set the way you want them. In this figure, these properties, which appear in the INSERT and UPDATE Specification group, are set to No Action. That means that primary keys in the Vendors table can't be changed if related records exist in the Invoices table, and a row can't be deleted from the Vendors table if related rows exist in the Invoices table. In most cases, that's what you want. In other cases, though, you'll want to change these properties to Cascade so update and delete operations are cascaded to the foreign key table.

Although these properties are the ones you're most likely to change, you can also use the Check Existing Data On Creation Or Re-Enabling property to control whether SQL Server checks existing data to be sure that it satisfies the constraint. By default, this property is set to Yes, which is usually what you want. In addition, by default, the Enforce For Replication property is set to Yes, which causes the relationship to be enforced when the database is replicated. *Replication* is a technique that's used to create multiple copies of the same database in different locations. By using replication, SQL Server can keep the various copies of a database synchronized. Because this feature is only used by DBAs for enterprise systems, a complete presentation is beyond the scope of this book.

The Foreign Key Relationships dialog box for the Invoices table



The Tables and Columns dialog box



Description

- To display the Foreign Key Relationships dialog box for a table, display the table in the Table Designer and click on the Relationships button in the toolbar.
- To add a new foreign key relationship, click on the Add button. To delete an existing relationship, select the relationship and click on the Delete button.
- To specify the tables and columns that define the relationship, select the relationship, select the Tables And Columns Specification property, and click the button that appears to display the Tables and Columns dialog box. Then, use this dialog box to specify the primary key table and the appropriate columns in both tables.
- To set other properties for a relationship, select the relationship and use the properties grid to change the properties.

Figure 12-3 How to work with foreign key relationships

How to work with indexes and keys

Figure 12-4 shows how to work with the indexes and keys of a table. To start, you display the table that contains the foreign key in the Table Designer. Then, you click on the Manage Indexes and Keys button in the Table Designer toolbar to display the Indexes/Keys dialog box. In this figure, for example, the Indexes/Keys dialog box is shown for the Vendors table.

If you have defined a primary key as described in figure 12-2, the primary key for the table is displayed in this dialog box. In this figure, for example, the primary key is named PK_Vendors. If you click on this key to select it and view its properties, you'll see that it defines a unique primary key with a clustered index.

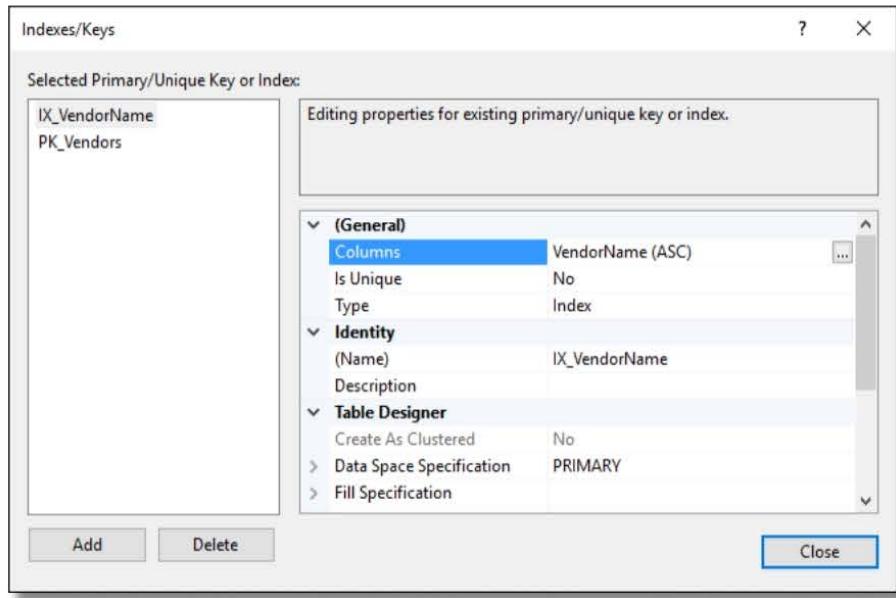
To add a new index, you can click on the Add button. This causes an index with a default name such as IX_Vendors to be added to the list box on the left side of the dialog box. Then, you can click on this index to select it, and you can set its properties. Here, you can use the Columns property to display a dialog box that allows you to specify the column or columns to index along with a sort order for each column. You can set the Type property to Index or Unique Key. If you set this property to Unique Key, the Is Unique property will automatically be set to Yes and grayed out. And finally, you can use the Name property to provide a more meaningful name for the index.

The Index/Keys dialog box in this figure shows most of the properties for an index named IX_VendorName. This index uses a nonclustered index to index the VendorName column in ascending order, and it does not require each vendor name to be unique. However, if you change the Type property to Unique Key, this index will define a unique key constraint. Then, SQL Server requires each vendor to have a unique name.

You can also create an index that enforces the uniqueness of its values without using a unique key constraint. To do that, you set the Is Unique property to Yes, and you set the Type property to Index. In most cases, though, you'll want to enforce uniqueness by setting the Type property to Unique Key.

By default, the Create As Clustered property is set to Yes for the primary key of the table, which is usually what you want. This causes the primary key to use a clustered index. Since a table can only have one clustered index, SQL Server grays out this property for all other indexes. If a table doesn't have a primary key, however, you can set the Create As Clustered property to Yes to create a clustered index for one index in the table.

The Indexes/Keys dialog box for the Vendors table



Description

- To display the Indexes/Keys dialog box for a table, display the table in the Table Designer and click on the Manage Indexes and Keys button in the Table Designer toolbar.
- To add a new index, click the Add button, use the Columns property to specify the column name and sort order for each column in the index, and use the Name property to enter the name you want to use for the index.
- To view or edit an existing index, select the index from the list box on the left side of the dialog box. Then, you can view its properties on the right side of the dialog box.
- To create a unique key and an index that's based on that key, use the Type property to select the Unique Index option. When you do, the Is Unique property will automatically be set to Yes.
- To create an index without creating a unique key, use the Type property to select the Index option and set the Is Unique property to No.
- To create a clustered index, set the Create As Clustered property to Yes. If a table already contains a clustered index, this property is grayed out to show that it isn't available.
- The other options in this dialog box are used for performance tuning. In most cases, the default values for these options are acceptable. For more information, see the SQL Server documentation.

Figure 12-4 How to work with indexes and keys

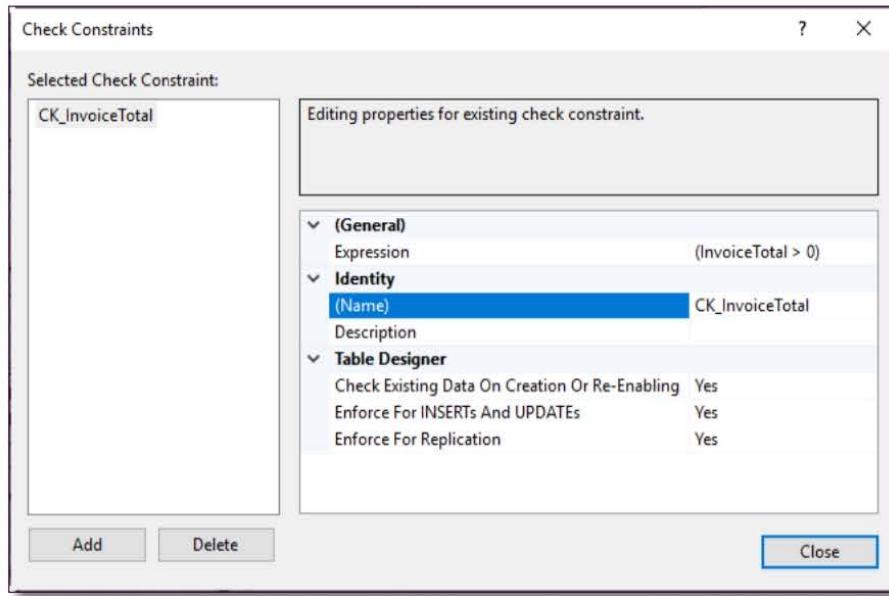
How to work with check constraints

Figure 12-5 shows how to work with check constraints. To start, it shows the Check Constraints dialog box that you can display by clicking on the Manage Check Constraints button in the Table Designer toolbar. You can use this dialog box to modify or delete existing check constraints for a table or to add new constraints. In this figure, for example, you can see a check constraint for the Invoices table. This constraint specifies that the InvoiceTotal column must be greater than zero.

As you learned in chapter 11, when you create check constraints using DDL, you can define them at either the column level or the table level. By contrast, the check constraints you create using the Check Constraints dialog box are always defined at the table level. Because of that, a constraint can refer to any column in the table where it's defined.

The properties that are available from the Table Designer group are similar to the properties you saw in the Foreign Key Relationships dialog box. The first one determines if existing data is checked when a new constraint is created. The second one determines if constraints are enforced when rows are inserted or updated. The third one determines if constraints are enforced when the database is replicated. In most cases, you'll set all three of these properties to Yes. If you want to temporarily disable a constraint during testing, however, you can do that by setting one or more of these properties to No.

The Check Constraints dialog box for the Invoices table



Description

- To display the Check Constraints dialog box for a table, display the table in the Table Designer and click on the Manage Check Constraints button in the Table Designer toolbar.
- To add a new constraint to the table, click the Add button. Then, you must enter the expression that defines the constraint in the Expression property, and you usually want to use the Name property to provide a meaningful name for the constraint.
- To delete a constraint, select the constraint and click the Delete button.
- To view or edit the properties for an existing constraint, select the constraint from the list that's displayed on the left side of the dialog box.
- By default, SQL Server checks existing data when you add a new check constraint to be sure it satisfies the constraint. If that's not what you want, you can set the Check Existing Data On Creation Or Re-Enabling property to No.
- By default, SQL Server enforces the constraint for insert and update operations. If that's not what you want, you can set the Enforce For INSERTs And UPDATEs property to No.
- By default, SQL Server enforces the constraint when the database is replicated. If that's not what you want, you can set the Enforce For Replication property to No.

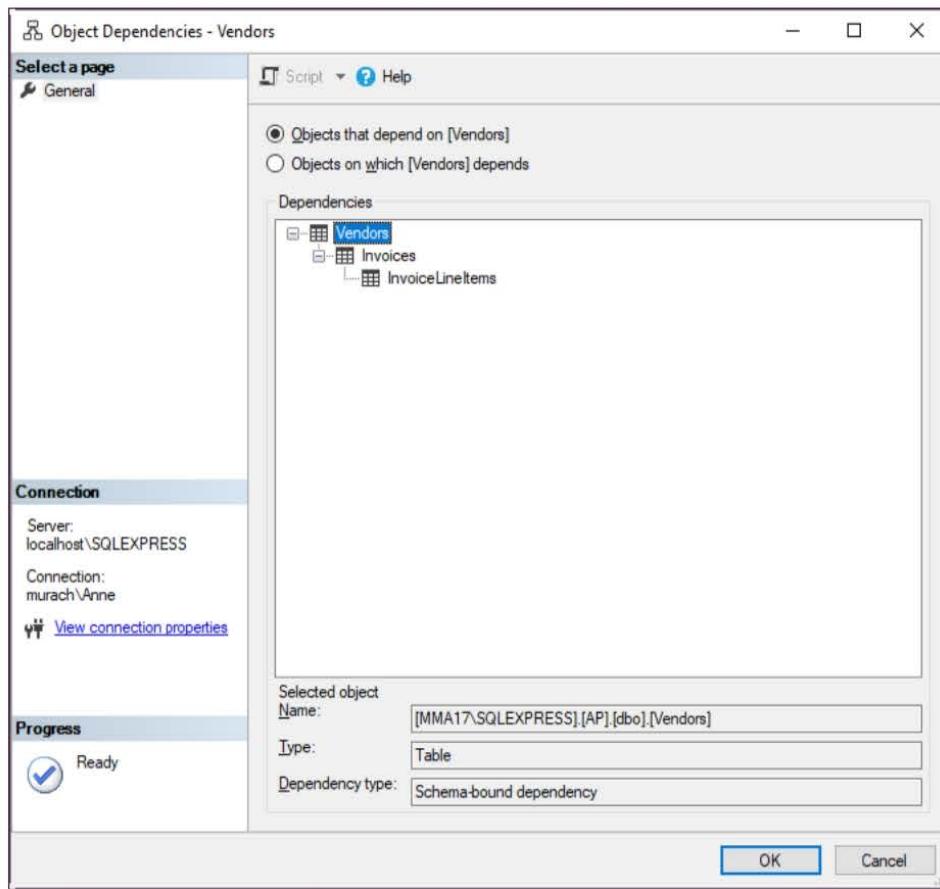
Figure 12-5 How to work with check constraints

How to examine table dependencies

Figure 12-6 shows how you can view the *dependencies* for a table. To do that, right-click on the table and select View Dependencies. Then, the Object Dependencies dialog box lists the objects that depend on that table. In this case, you can see that the Invoices table depends on the Vendors table, and the InvoiceLineItems table depends on the Invoices table.

Besides the dependencies that are based on the relationships between the tables, there may be views and stored procedures that depend on the Invoices table. Before you make a change to the Invoices table, then, you'll want to consider how that change will affect these objects.

The Object Dependencies dialog box for the Vendors table



Description

- To view the *dependencies* for a table, right-click the table and select View Dependencies to display the Object Dependencies dialog box.
- To expand or collapse a dependency, click on the plus (+) or minus sign (-) that's displayed to the left of the dependency.
- You should check table dependencies before you delete or modify a column or an entire table.

Figure 12-6 How to examine table dependencies

How to generate scripts

If you use the Management Studio to design a database, you may eventually need or want to generate a script that contains the DDL statements that define the database or that record the changes that you've made to a database. That way, you can save these scripts and run them later if necessary.

Fortunately, the Management Studio makes it easy to generate these types of scripts. Unfortunately, these scripts are often formatted in a way that's not easy to read. Worse, these scripts often contain DDL statements that are more complex than if you had coded them yourself. Still, by studying these DDL statements, you can learn a lot, and you can see the DDL statements that the Management Studio uses to create or alter a database.

How to generate scripts for databases and tables

Figure 12-7 describes how you can use the Management Studio to generate a script that creates, drops, or alters most database objects including the database itself. To do that, you right-click on the appropriate database object and select the appropriate commands from the resulting menus. For example, to create a script that creates the Invoices table, right-click on the Invoices table, select the Script Table as submenu, select the CREATE To submenu, and select the New Query Editor Window command. This generates a script like the one in this figure that creates the Invoices table, and it places this script in a new Query Editor window.

If you study this script, you'll see that it uses nearly a full page of DDL statements to create a simple Invoices table that only contains four columns. In addition, this script encloses all names in square brackets ([]) even though that's not required, it qualifies table names with the default schema even though that's not required, and it uses separate ALTER TABLE statements to define the foreign key relationship constraint, check constraints, and default constraint even though that isn't necessary. Compared with the script presented at the end of chapter 11, I think you'll agree that this script is unwieldy and difficult to read even though it's for a simple table that contains only four columns. But that's one of the downsides of using a graphical tool like the Management Studio instead of coding the DDL statements yourself.

Although the CREATE scripts are the ones that you'll want to generate most often, you can also generate other scripts such as ALTER and DROP scripts for most objects. In addition, you can generate SELECT, INSERT, UPDATE, and DELETE scripts for some objects such as tables, and you can generate EXECUTE scripts for other objects such as stored procedures. Some of these scripts are essentially templates that you'll need to modify before they become functional. Still, they can give you a good start for creating certain types of statements. With a little experimentation, you should be able to figure out how this works.

A generated script that creates a simple Invoices table

```
USE [New_AP]
GO

SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Invoices]
(
    [InvoiceID] [int] IDENTITY(1,1) NOT NULL,
    [VendorID] [int] NOT NULL,
    [InvoiceDate] [date] NULL,
    [InvoiceTotal] [money] NULL,
    CONSTRAINT [PK_Invoices] PRIMARY KEY CLUSTERED ([InvoiceID] ASC)
        WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
        ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
        OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
)
ON [PRIMARY]
GO

ALTER TABLE [dbo].[Invoices] ADD DEFAULT ((0)) FOR [InvoiceTotal]
GO

ALTER TABLE [dbo].[Invoices] WITH CHECK ADD CONSTRAINT [FK_Invoices_Vendors]
    FOREIGN KEY([VendorID]) REFERENCES [dbo].[Vendors] ([VendorID])
GO

ALTER TABLE [dbo].[Invoices]
    CHECK CONSTRAINT [FK_Invoices_Vendors]
GO

ALTER TABLE [dbo].[Invoices] WITH CHECK ADD CONSTRAINT [CK_InvoiceTotal]
    CHECK (([InvoiceTotal]>(0)))
GO

ALTER TABLE [dbo].[Invoices]
    CHECK CONSTRAINT [CK_InvoiceTotal]
GO
```

Description

- You can use the Management Studio to generate scripts to create, drop, or alter most objects that are contained in the database. You can send each script to Management Studio's Query Editor, a file, or to the clipboard.
- To generate a script that creates a database, right-click on the database and select Script Database as→CREATE To. Then, select New Query Editor Window, File, or Clipboard.
- To generate a script that creates a table, right-click on the table and select Script Table as→CREATE To. Then, select New Query Editor Window, File, or Clipboard.

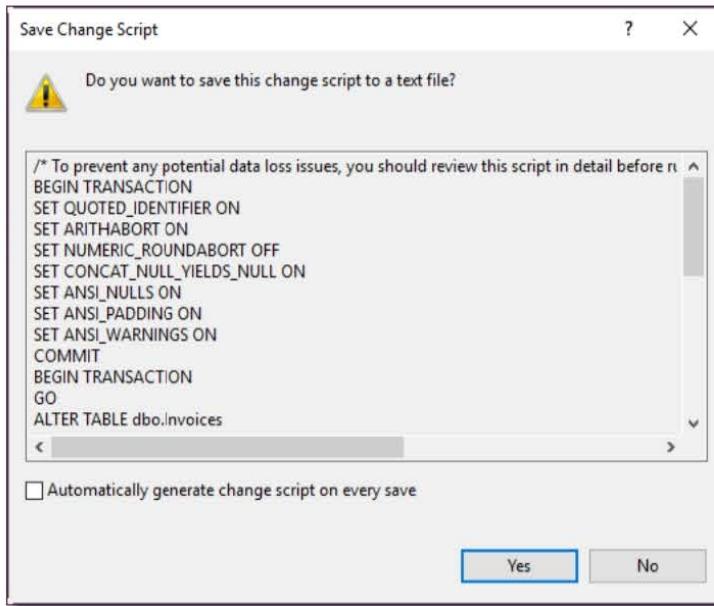
If you experiment with scripts, you'll find that you can send them to a new Query Editor window, to a file, or to the clipboard. Most of the time, it's easiest to send the generated script to a Query Editor window. Then, you can review the script before you run it, save it, or copy it to the clipboard. In some cases, though, you may want to send a script directly to a file without viewing it first. Or, you may want to send the script to the clipboard without viewing it first so you can paste it into another tool and use that tool to view, save, or run the script.

How to generate a change script when you modify a table

When you use the Table Designer to modify the design of a table, a Generate Change Script button becomes available in the Table Designer toolbar. If you want, you can click on this button to display the Save Change Script dialog box shown in figure 12-8. Then, you can use this dialog box to examine or save the SQL script that the Management Studio uses to alter the table. Most of the time, this isn't necessary. However, if you want to keep a permanent record of the change, or if you want to apply the change to other databases, you can save this script. Then, if necessary, you can run it against other databases.

If you want to generate and save a change script every time you modify a table, you can check the Automatically Generate Change Script On Every Save option. Then, every time you attempt to save changes to the design of a table, the Management Studio will prompt you with a Save Change Script dialog box like the one in this figure. To save the script, you can click Yes and respond to the resulting Save dialog box.

The Save Change Script dialog box



Description

- If you use the Table Designer to modify the design of a table, you can click the Generate Change Script button in the toolbar to display the Save Change Script dialog box shown above. You can use this dialog box to examine or save the SQL script that the Management Studio uses to alter the table.
- If you check the Automatically Generate Change Script On Every Save box, the Management Studio will prompt you with a Save Change Script dialog box each time you attempt to save changes to the design of a table.

Figure 12-8 How to generate a change script when you modify a table

Perspective

In this chapter, you learned how to use the Management Studio to create and work with database objects, such as tables, keys, indexes, and constraints as well as the database itself. Now that you know how to use the Management Studio to work with the design of a database, you may be wondering when you should use it for database design and when you should code the DDL statements yourself.

Although it's often a matter of preference, most SQL programmers find it easier to use the Management Studio to create and work with database objects. That way, they don't have to worry about the exact syntax of the DDL statements, which they may use infrequently.

However, some experienced database programmers prefer to enter the DDL statements themselves. That way, they have more control over how these scripts are coded. In addition, they can save a copy of the script that creates the database for future reference, which may be helpful if the database design ever needs to be ported to another type of database management system such as Oracle, DB2, or MySQL.

Terms

Table Designer
replication
dependencies

Exercises

1. Use the Management Studio to create a new database called Membership2 using the default settings. (If the database already exists, use the Management Studio to delete it and then recreate it.)
2. Use the Management Studio to create the following tables and relationships in the Membership database. Define IndividualID and GroupID as IDENTITY columns. Allow Address and Phone to accept null values; none of the other columns should allow null values. Define the Dues column with a default of zero and a check constraint to allow only positive values. Define the DuesPaid column with a default Boolean value of False.



3. Use the Management Studio to index the GroupMembership table. Create a clustered index on the GroupID column, a nonclustered index on the IndividualID column, and a unique index and constraint on both columns.

Section 4

Advanced SQL skills

This section teaches SQL skills that go beyond the essentials. After you read all of the chapters in this section, you'll have the skills of a professional SQL programmer. To make these chapters as easy to use as possible, they're designed as independent modules. That means that you can read them in any order you prefer.

In chapter 13, you can learn how to work with views, which let you simplify and restrict access to the data in a database. In chapter 14, you can learn how to use scripts to control the processing of SQL statements that you execute from a client tool like the Management Studio. Scripts are just one type of procedural program you can create in SQL Server. In chapter 15, you can learn how to use the other types of procedural programs: stored procedures, triggers, and functions.

In chapter 16, you can learn how to use transactions and locking to prevent data errors in a multi-user environment. In chapter 17, you can learn how to secure a database to restrict who has access to it. In chapter 18, you can learn how to work with XML. Finally in chapter 19, you can learn how to work with large binary values such as images, sound, and video.

How to work with views

As you've seen throughout this book, SELECT queries can be complicated, particularly if they use multiple joins, subqueries, or complex functions. Because of that, you may want to save the queries you use regularly. One way to do that is to store the statement in a file using the Management Studio. Another way is to create a view.

Unlike a file you create with the Management Studio, a view is stored as part of the database. That means it can be used not only by SQL programmers, but by users and application programs that have access to the database. This provides some distinct advantages over using tables directly, as you'll see in this chapter.

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
Perspective	416

An introduction to views

In chapter 1, you learned the basics of how views work. In the next topic, then, I'll just review this information. Then, I'll present some of the benefits of views so you'll know when and why you should use them.

How views work

A view is a SELECT statement that's stored with the database. To create a view, you use a CREATE VIEW statement like the one shown in figure 13-1. This statement creates a view named VendorsMin that retrieves the VendorName, VendorState, and VendorPhone columns from the Vendors table.

You can think of a view as a virtual table that consists only of the rows and columns specified in its CREATE VIEW statement. The table or tables that are listed in the FROM clause are called the base tables for the view. Since the view refers back to the base tables, it doesn't store any data itself, and it always reflects the most current data in the base tables.

To use a view, you refer to it from another SQL statement. The SELECT statement in this figure, for example, uses the VendorsMin view in the FROM clause instead of a table. As a result, this SELECT statement extracts its result set from the virtual table that the view represents. In this case, all the rows for vendors in California are retrieved from the view.

Because a view is stored as an object in a database, it can be used by anyone who has access to the database. That includes users who have access to the database through end-user programs such as programs that provide for ad hoc queries and report generation, and application programs that are written specifically to work with the data in the database. In fact, views are often designed to be used with these types of programs. In the next topic, you'll learn why.

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

(122 rows)

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	Attek 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

(75 rows)

Description

- A view consists of a SELECT statement that's stored as an object in the database. The tables referenced in the SELECT statement are called the *base tables* for the view.
- When you create a view, the query on which it's based is optimized by SQL Server before it's saved in the database. Then, you can refer to the view anywhere you would normally use a table in any of the data manipulation statements: SELECT, INSERT, UPDATE, and DELETE.
- Although a view behaves like a virtual table, it doesn't store any data. Since the view refers back to its base tables, it always returns current data.
- A view can also be referred to as a *viewed table* because it provides a view to the underlying base tables.

Figure 13-1 How views work

Benefits of using views

Figure 13-2 describes some of the advantages of using views. To start, the data that you access through a view isn't dependent on the structure of the database. To illustrate, suppose a view refers to a table that you've decided to divide into two tables. To accommodate this change, you simply modify the view; you don't have to modify any statements that refer to the view. That means that users who query the database using the view don't have to be aware of the change in the database structure, and application programs that use the view don't have to be modified.

You can also use views to restrict access to a database. To do that, you include just the columns and rows you want a user or application program to have access to in the views. Then, you let the user or program access the data only through the views. The view shown in this figure, for example, restricts access to a table that contains information on investors. In this case, the view provides access to name and address information that might be needed by the support staff that maintains the table. By contrast, another view that includes investment information could be used by the consultants who manage the investments.

Views are also flexible. Because views can be based on almost any SELECT statement, they can be used to provide just the data that's needed for specific situations. In addition, views can hide the complexity of a SELECT statement. That makes it easier for end users and application programs to retrieve the data they need. Finally, views can be used not only to retrieve data, but to modify data as well. You'll see how that works later in this chapter.

Some of the benefits provided by views

Benefit	Description
Design independence	Data that's accessed through a view is independent of the underlying database structure. That means that you can change the design of a database and then modify the view as necessary so the queries that use it don't need to be changed.
Data security	You can create views that provide access only to the data that specific users are allowed to see.
Flexibility	You can create custom views to accommodate different needs.
Simplified queries	You can create views that hide the complexity of retrieval operations. Then, the data can be retrieved using simple SELECT statements.
Updatability	With certain restrictions, a view can be used to update, insert, and delete data from a base table.

The data in a table of investors

	InvestorID	LastName	FirstName	Address	City	State	ZipCode	Phone	Investments	NetGain
1	1	Anders	Maria	345 Winchell Pl	Anderson	IN	46014	(765) 555-7878	15000.00	1242.57
2	2	Trujillo	Ana	1298 E Smathers St.	Benton	AR	72018	(510) 555-7733	43500.00	8497.44
3	3	Moreno	Antonio	6925 N Parkland Ave.	Puyallup	WA	98373	(253) 555-8332	22900.00	2338.87
4	4	Hardy	Thomas	83 d'Urerville Ln.	Casterbridge	GA	31209	(478) 555-1139	5000.00	-245.69
5	5	Berglund	Christina	22717 E 73rd Ave.	Dubuque	IA	52004	(319) 555-1139	11750.00	865.77

(5 rows)

A view that restricts access to certain columns

```
CREATE VIEW InvestorsGeneral
AS
SELECT InvestorID, LastName, FirstName, Address,
       City, State, ZipCode, Phone
FROM Investors;
```

The data retrieved by the view

	InvestorID	LastName	FirstName	Address	City	State	ZipCode	Phone		
1	1	Anders	Maria	345 Winchell Pl	Anderson	IN	46014	(765) 555-7878		
2	2	Trujillo	Ana	1298 E Smathers St.	Benton	AR	72018	(510) 555-7733		
3	3	Moreno	Antonio	6925 N Parkland Ave.	Puyallup	WA	98373	(253) 555-8332		
4	4	Hardy	Thomas	83 d'Urerville Ln.	Casterbridge	GA	31209	(478) 555-1139		
5	5	Berglund	Christina	22717 E 73rd Ave.	Dubuque	IA	52004	(319) 555-1139		

(5 rows)

Description

- You can create a view based on almost any SELECT statement. That means that you can code views that join tables, summarize data, and use subqueries and functions.
- You can restrict access to the data in a table by including selected columns in the SELECT clause for a view, or by including a WHERE clause in the SELECT statement so that only selected rows are retrieved by the view.

Figure 13-2 Benefits of using views

How to create and manage views

Now that you understand how views work and what benefits they provide, you're ready to learn how to create and manage them. That's what you'll learn in the topics that follow.

How to create a view

Figure 13-3 presents the CREATE VIEW statement you use to create a view. In its simplest form, you code the name of the view in the CREATE VIEW clause followed by the AS keyword and the SELECT statement that defines the view. The statement shown in this figure, for example, creates a view named VendorShortList. This view includes selected columns from the Vendors table for all vendors with invoices. When this statement is executed, the view is added to the current database and a message like the one shown in this figure is displayed to indicate that the statement was successful.

Because a SELECT statement can refer to a view, the SELECT statement you code within the definition of a view can also refer to another view. In other words, views can be nested. I recommend you avoid using nested views, however, because the dependencies between tables and views can become confusing, which can make problems difficult to locate.

The SELECT statement for a view can use any of the features of a normal SELECT statement with two exceptions. First, it can't include an ORDER BY clause unless it also uses a TOP clause or the OFFSET and FETCH clauses. That means that if you want to sort the result set that's extracted from a view, you have to include an ORDER BY clause in the SELECT statement that refers to the view. Second, it can't include the INTO keyword. That's because a view can't be used to create a permanent table.

By default, the columns in a view are given the same names as the columns in the base tables. If a view contains a calculated column, however, you'll want to name that column just as you do in other SELECT statements. In addition, you'll need to rename columns from different tables that have the same name. To do that, you can use the AS clause in the SELECT statement or you can code the column names in the CREATE VIEW clause. You'll see examples of both of these techniques in the next figure.

The CREATE VIEW statement also provides three optional clauses: WITH ENCRYPTION, WITH SCHEMABINDING, and WITH CHECK OPTION. The WITH ENCRYPTION clause prevents other users from examining the SELECT statement on which the view is based. In general, though, you don't need to use this option unless your system requires enhanced security.

The WITH SCHEMABINDING clause protects a view by binding it to the database structure, or schema. This prevents the underlying base tables from being deleted or modified in any way that affects the view. You'll typically use this option for production databases, but not for databases you're using for testing.

The WITH CHECK OPTION clause prevents a row in a view from being updated if that would cause the row to be excluded from the view. I'll have more to say about this clause in the topic on updating rows using a view.

The syntax of the CREATE VIEW statement

```
CREATE VIEW view_name [(column_name_1 [, column_name_2]...)]
[WITH {ENCRYPTION|SCHEMABINDING|ENCRYPTION,SCHEMABINDING}]
AS
select_statement
[WITH CHECK OPTION]
```

A CREATE VIEW statement that creates a view of vendors that have invoices

```
CREATE VIEW VendorShortList
AS
SELECT VendorName, VendorContactLName, VendorContactFName, VendorPhone
FROM Vendors
WHERE VendorID IN (SELECT VendorID FROM Invoices);
```

The response from the system

Commands completed successfully.

Description

- You use the CREATE VIEW statement to create a view. The name you give the view must not be the same as the name of any existing table or view.
- The SELECT statement within the view can refer to as many as 256 tables, and it can use any valid combination of joins, unions, or subqueries.
- You can create a view that's based on another view rather than on a table, called a *nested view*. SQL Server views can be nested up to 32 levels deep.
- The SELECT statement for a view can't include an INTO clause, and it can't include an ORDER BY clause unless the TOP clause or the OFFSET and FETCH clauses are also used. To sort the rows in a view, you have to include the ORDER BY clause in the SELECT statement that uses the view.
- You can name the columns in a view by coding a list of names in parentheses following the view name or by coding the new names in the SELECT clause. A column must be named if it's calculated from other columns or if a column with the same name already exists. Otherwise, the name from the base table can be used.
- You can use the WITH ENCRYPTION clause to keep users from examining the SQL code that defines the view.
- You can use the WITH SCHEMABINDING clause to bind a view to the *database schema*. Then, you can't drop the tables on which the view is based or modify the tables in a way that would affect the view.
- If you include the WITH SCHEMABINDING clause, you can't use the all columns operator (*) in the SELECT statement. In addition, you must qualify the names of tables and views in the FROM clause with the name of the schema that contains them.
- You can use the WITH CHECK OPTION clause to prevent a row from being updated through a view if it would no longer be included in the view. See figure 13-7 for details.

Examples that create views

To help you understand the flexibility that views provide, figure 13-4 presents several CREATE VIEW statements. The first statement creates a view that joins data from the Vendors and Invoices tables. The second statement creates a view that retrieves the top five percent of invoices in the Invoices table. Notice that this SELECT statement includes an ORDER BY clause that sorts the rows in descending sequence so the invoices with the largest amounts are retrieved. This is one of only two cases in which you can use the ORDER BY clause. The other is if you include the OFFSET and FETCH clauses on the ORDER BY clause.

The third and fourth statements illustrate the two ways that you can name the columns in a view. In both cases, the SELECT statement retrieves a calculated column, so a name must be assigned to this column. The third statement shows how you would do this using the CREATE VIEW clause. Notice that even if you only want to name one column, you have to include the names for all the columns even if they're the same as the names in the base tables. By contrast, if you name this column in the SELECT clause as shown in the fourth example, you can let the other column names default to the column names in the base table. Since this syntax is easier to use, you'll use it most of the time. Keep in mind, though, that the ANSI standards don't support this syntax for naming view columns.

The fifth statement creates a view that summarizes the rows in the Invoices table by vendor. This illustrates the use of the aggregate functions and the GROUP BY clause in a view. In this case, the rows are grouped by vendor name, and a count of the invoices and the invoice total are calculated for each vendor.

Like the first statement, the last statement joins data from the Vendors and Invoices table. Unlike the first statement, though, this statement includes the WITH SCHEMABINDING clause. That means that neither the Vendors nor the Invoices table can be deleted without first deleting the view. In addition, no changes can be made to these tables that would affect the view. Notice that the table names in the FROM clause are qualified with the name of the table's schema, in this case, dbo. If you include the WITH SCHEMABINDING clause, you must qualify the table names in this way.

A CREATE VIEW statement that uses a join

```
CREATE VIEW VendorInvoices
AS
SELECT VendorName, InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Vendors JOIN Invoices ON Vendors.VendorID = Invoices.VendorID;
```

A CREATE VIEW statement that uses TOP and ORDER BY clauses

```
CREATE VIEW TopVendors
AS
SELECT TOP 5 PERCENT VendorID, InvoiceTotal
FROM Invoices
ORDER BY InvoiceTotal DESC;
```

Two CREATE VIEW statements that name the columns in a view

A statement that names all the view columns in its CREATE VIEW clause

```
CREATE VIEW OutstandingInvoices
(InvoiceNumber, InvoiceDate, InvoiceTotal, BalanceDue)
AS
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
       InvoiceTotal - PaymentTotal - CreditTotal
  FROM Invoices
 WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
```

A statement that names just the calculated column in its SELECT clause

```
CREATE VIEW OutstandingInvoices
AS
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
       InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue
  FROM Invoices
 WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
```

A CREATE VIEW statement that summarizes invoices by vendor

```
CREATE VIEW InvoiceSummary
AS
SELECT VendorName, COUNT(*) AS InvoiceQty, SUM(InvoiceTotal) AS InvoiceSum
  FROM Vendors JOIN Invoices ON Vendors.VendorID = Invoices.VendorID
 GROUP BY VendorName;
```

A CREATE VIEW statement that uses the WITH SCHEMABINDING option

```
CREATE VIEW VendorsDue
WITH SCHEMABINDING
AS
SELECT InvoiceDate AS Date, VendorName AS Name,
       VendorContactFName + ' ' + VendorContactLName AS Contact,
       InvoiceNumber AS Invoice,
       InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue
  FROM dbo.Vendors JOIN dbo.Invoices
    ON Vendors.VendorID = Invoices.VendorID
 WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
```

Note

- If you name the columns of a view in the CREATE VIEW clause, you have to name all of the columns. By contrast, if you name the columns in the SELECT clause, you can name just the columns you need to rename.

How to create an updatable view

Once you create a view, you can refer to it in a SELECT statement as you saw in figure 13-1. In addition, you can refer to it in INSERT, UPDATE, and DELETE statements to modify an underlying table. To do that, the view must be updatable. Figure 13-5 lists the requirements for creating updatable views.

The first three requirements have to do with what you can code in the select list of the SELECT statement that defines the view. As you can see, the select list can't include the DISTINCT or TOP clause, it can't include aggregate functions, and it can't include calculated columns. In addition, the SELECT statement can't include a GROUP BY or HAVING clause, and two SELECT statements can't be joined by a union operation.

The first CREATE VIEW statement in this figure creates a view that's updatable. This view adheres to all of the requirements for updatable views. That means that you can refer to it in an INSERT, UPDATE, or DELETE statement. For example, you could use the UPDATE statement shown in this figure to update the CreditTotal column in the Invoices base table.

By contrast, the second CREATE VIEW statement in this figure creates a read-only view. This view is read-only because the select list contains a calculated value.

In general, using INSERT, UPDATE, and DELETE statements to update data through a view is inflexible and prone to errors. Because of that, you should avoid this technique whenever possible. Instead, you should consider using INSTEAD OF triggers to update data through a view. You'll learn about this type of trigger in chapter 15.

Requirements for creating updatable views

- The select list can't include a DISTINCT or TOP clause.
- The select list can't include an aggregate function.
- The select list can't include a calculated value.
- The SELECT statement can't include a GROUP BY or HAVING clause.
- The view can't include the UNION operator.

A CREATE VIEW statement that creates an updatable view

```
CREATE VIEW InvoiceCredit
AS
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal, PaymentTotal, CreditTotal
FROM Invoices
WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
```

An UPDATE statement that updates the view

```
UPDATE InvoiceCredit
SET CreditTotal = CreditTotal + 200
WHERE InvoiceTotal - PaymentTotal - CreditTotal >= 200;
```

A CREATE VIEW statement that creates a read-only view

```
CREATE VIEW OutstandingInvoices
AS
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
       InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue
FROM Invoices
WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
```

Description

- An *updatable view* is one that can be used in an INSERT, UPDATE, or DELETE statement to modify the contents of a base table that the view refers to. If a view is not updatable, it's called a *read-only view*.
- The requirements for coding updatable views are more restrictive than for coding read-only views. That's because SQL Server must be able to unambiguously determine which base tables and which columns are affected.
- You can also insert, update, and delete data through a view using an INSTEAD OF trigger. See chapter 15 for details.

How to delete or modify a view

Figure 13-6 presents the statements you use to delete or modify a view. To delete a view, you use the DROP VIEW statement. In this statement, you simply name the view you want to delete. Like the other statements for deleting database objects, this statement deletes the view permanently. So, you may want to make a backup copy of the database first if there's any chance that you may want to restore the view later.

To modify a view, you can use the ALTER VIEW statement. Notice that the syntax of this statement is the same as the syntax of the CREATE VIEW statement. If you understand the CREATE VIEW statement, then, you won't have any trouble using the ALTER VIEW statement.

Instead of using the ALTER VIEW statement to modify a view, you can delete the view and then recreate it. If you've defined permissions for the view, you should know that those permissions are deleted when the view is deleted. If that's not what you want, you should use the ALTER VIEW statement instead.

The examples in this figure show how you can use the DROP VIEW and ALTER VIEW statements. The first example is a CREATE VIEW statement that creates a view named Vendors_SW. This view retrieves rows from the Vendors table for vendors located in four states. Then, the second example is an ALTER VIEW statement that modifies this view so it includes vendors in two additional states. Finally, the third example is a DROP VIEW statement that deletes this view.

In the last chapter, you learned how to display the dependencies for a table. Before you delete a table, you should display its dependencies to determine if any views are dependent on the table. If so, you should delete the views along with the tables. If you don't, a query that refers to the view will cause an error. To prevent this problem, you can bind the view to the database schema by specifying the WITH SCHEMABINDING option in the CREATE VIEW or ALTER VIEW statement. Then, you won't be able to delete the base table without deleting the views that depend on it first.

The syntax of the DROP VIEW statement

```
DROP VIEW view_name
```

The syntax of the ALTER VIEW statement

```
ALTER VIEW view_name [(column_name_1 [, column_name_2]...)]
[WITH {ENCRYPTION|SCHEMABINDING|ENCRYPTION,SCHEMABINDING}]
AS
select_statement
[WITH CHECK OPTION]
```

A statement that creates a view

```
CREATE VIEW Vendors_SW
AS
SELECT *
FROM Vendors
WHERE VendorState IN ('CA', 'AZ', 'NV', 'NM');
```

A statement that modifies the view

```
ALTER VIEW Vendors_SW
AS
SELECT *
FROM Vendors
WHERE VendorState IN ('CA', 'AZ', 'NV', 'NM', 'UT', 'CO');
```

A statement that deletes the view

```
DROP VIEW Vendors_SW;
```

Description

- To delete a view from the database, use the DROP VIEW statement.
- To modify the definition of a view, you can delete the view and then create it again, or you can use the ALTER VIEW statement to specify the new definition.
- When you delete a view, any permissions that are assigned to the view are also deleted.
- If you delete a table, you should also delete any views that are based on that table. Otherwise, an error will occur when you run a query that refers to one of those views. To find out what views are dependent on a table, display the table's dependencies as described in chapter 12.
- If you specify the WITH SCHEMABINDING option when you create or modify a view, you won't be able to delete the base tables without first deleting the view.

Note

- ALTER VIEW isn't an ANSI-standard statement. Although it's supported on other SQL-based systems, its behavior on each system is different.

How to use views

So far, you've seen how to use views in SELECT statements to retrieve data from one or more base tables. But you can also use views in INSERT, UPDATE, and DELETE statements to modify the data in a base table. You'll learn how to do that in the topics that follow. In addition, you'll learn how to use some views provided by SQL Server to get information about the database schema.

How to update rows through a view

Figure 13-7 shows how you can update rows in a table through a view. To do that, you simply name the view that refers to the table in the UPDATE statement. Note that for this to work, the view must be updatable as described in figure 13-5. In addition, the UPDATE statement can only update the data in a single base table, even if the view refers to two or more tables.

The examples in this figure illustrate how this works. First, the CREATE VIEW statement creates an updatable view named VendorPayment that joins data from the Vendors and Invoices tables. The data that's retrieved by this view is shown in this figure. Then, the UPDATE statement uses this view to modify the PaymentDate and PaymentTotal columns for a specific vendor and invoice. As you can see, the Invoices table reflects this update.

Notice, however, that the row that was updated is no longer included in the view. That's because the row no longer meets the criteria in the WHERE clause of the SELECT statement that defines the view. If that's not what you want, you can include the WITH CHECK OPTION clause in the CREATE VIEW statement. Then, an update through the view isn't allowed if it causes the row to be excluded from the view. If the WITH CHECK OPTION clause had been included in the definition of the VendorPayment view, for example, the UPDATE statement in this figure would have resulted in an error message like the one shown.

A statement that creates an updatable view

```
CREATE VIEW VendorPayment
AS
SELECT VendorName, InvoiceNumber, InvoiceDate, PaymentDate,
       InvoiceTotal, CreditTotal, PaymentTotal
FROM Invoices JOIN Vendors ON Invoices.VendorID = Vendors.VendorID
WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
```

The data retrieved by the view before the update

	VendorName	InvoiceNumber	InvoiceDate	PaymentDate	InvoiceTotal	CreditTotal	PaymentTotal
6	Federal Express Corporation	263253273	2020-01-22	NULL	30.75	0.00	0.00
7	Malloy Lithographing Inc	P-0608	2020-01-23	NULL	20551.18	1200.00	0.00
8	Ford Motor Credit Company	9982771	2020-01-24	NULL	503.20	0.00	0.00
9	Cardinal Business Media, Inc.	134116	2020-01-28	NULL	90.36	0.00	0.00
10	Malloy Lithographing Inc	0-2436	2020-01-31	NULL	10976.06	0.00	0.00

A statement that updates the Invoices table through the view

```
UPDATE VendorPayment
SET PaymentTotal = 19351.18, PaymentDate = '2020-02-02'
WHERE VendorName = 'Malloy Lithographing Inc' AND InvoiceNumber = 'P-0608';
```

The updated Invoices table

	InvoiceID	VendorID	InvoiceNumber	InvoiceDate	InvoiceTotal	PaymentTotal	CreditTotal	TermsID	InvoiceDueDate
102	102	110	P-0608	2020-01-23	20551.18	19351.18	1200.00	3	2020-02-22
103	103	122	989319-417	2020-01-23	2051.59	2051.59	0.00	3	2020-02-22
104	104	123	263253243	2020-01-23	44.44	44.44	0.00	3	2020-02-22
105	105	106	9982771	2020-01-24	503.20	0.00	0.00	3	2020-02-23
106	106	110	0-2060	2020-01-24	23517.58	21221.63	2295.95	3	2020-02-23

The data retrieved by the view after the update

	VendorName	InvoiceNumber	InvoiceDate	PaymentDate	InvoiceTotal	CreditTotal	PaymentTotal
6	Federal Express Corporation	263253273	2020-01-22	NULL	30.75	0.00	0.00
7	Ford Motor Credit Company	9982771	2020-01-24	NULL	503.20	0.00	0.00
8	Cardinal Business Media, I...	134116	2020-01-28	NULL	90.36	0.00	0.00
9	Malloy Lithographing Inc	0-2436	2020-01-31	NULL	10976.06	0.00	0.00

The response if WITH CHECK OPTION is specified for the view

The attempted insert or update failed because the target view either specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows resulting from the operation did not qualify under the CHECK OPTION constraint.
The statement has been terminated.

Description

- You can use the UPDATE statement to update a table through a view. To do that, you name the view in the UPDATE clause.
- The view you name in the UPDATE statement must be updatable. In addition, the UPDATE statement can't update data in more than one base table.
- If you don't specify WITH CHECK OPTION when you create a view, a change you make through the view can cause the modified rows to no longer be included in the view.
- If you specify WITH CHECK OPTION when you create a view, an error will occur if you try to modify a row in such a way that it would no longer be included in the view.

Figure 13-7 How to update rows through a view

How to insert rows through a view

To insert rows through a view, you use the INSERT statement as shown in figure 13-8. At the top of this figure, you can see a CREATE VIEW statement for a view named IBM_Invoices. This view retrieves columns and rows from the Invoices table for the vendor named IBM. Then, the INSERT statement attempts to insert a row into the Invoices table through this view.

This insert operation fails, though, because the view and the INSERT statement don't include all of the required columns for the Invoices table. In this case, a value is required for the VendorID, InvoiceNumber, InvoiceDate, InvoiceTotal, TermsID, and InvoiceDueDate columns. By contrast, the InvoiceID column can be omitted because it's an identity column; the PaymentTotal and CreditTotal columns can be omitted because they have default values; and the PaymentDate column can be omitted because it allows null values.

In addition to providing values for all the required columns in a table, you should know that the INSERT statement can insert rows into only one table. That's true even if the view is based on two or more tables and all of the required columns for those tables are included in the view. In that case, you could use a separate INSERT statement to insert rows into each table through the view.

How to delete rows through a view

Figure 13-8 also shows how to delete rows through a view. To do that, you use a DELETE statement like the one shown here. This statement deletes an invoice from the Invoices table through the IBM_Invoices view. As you can see, the response from the system shows that one row was affected. Remember, though, that any delete operations on the Invoices table are cascaded to the InvoiceLineItems table. As a result, any line items for this invoice are also deleted.

A statement that creates an updatable view

```
CREATE VIEW IBM_Invoices
AS
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
WHERE VendorID = (SELECT VendorID FROM Vendors WHERE VendorName = 'IBM');
```

The contents of the view

	InvoiceNumber	InvoiceDate	InvoiceTotal
1	QP58872	2019-11-07	116.54
2	Q545443	2019-12-09	1083.58

An INSERT statement that fails due to columns with null values

```
INSERT INTO IBM_Invoices
(InvoiceNumber, InvoiceDate, InvoiceTotal)
VALUES ('RA23988', '2020-03-04', 417.34);
```

The response from the system

Cannot insert the value NULL into column 'VendorID', table 'AP.dbo.Invoices';
 column does not allow nulls. INSERT fails.
 The statement has been terminated.

A DELETE statement that succeeds

```
DELETE FROM IBM_Invoices
WHERE InvoiceNumber = 'Q545443';
```

The response from the system

(1 row affected)

Description

- You can use the INSERT statement to insert rows into a base table through a view. To do that, you name the view in the INSERT clause. Both the view and the INSERT statement must include all of the columns from the base table that require a value.
- If the view names more than one base table, an INSERT statement can insert data into only one of those tables.
- You can use the DELETE statement to delete rows from a base table through a view. To do that, you name the table in the DELETE clause. For this to work, the view must be based on a single table.

How to use the catalog views

The ANSI standards specify that a SQL database must maintain an online *system catalog* that lists all of the objects in a database. Although SQL Server lets you query the system catalogs directly, I don't recommend you do that. That's because if you do, you have to code queries that are dependent on the structure of the system tables that make up the system catalog. So if the system tables change in a future release of SQL Server, you have to change your queries.

Instead of querying the system tables directly, you can use the *catalog views* provided by SQL Server. Because these views are independent of the structure of the system tables, you don't have to worry about changing the queries that refer to them if the structure changes. Figure 13-9 lists some of these views and shows you how to use them.

To display the data defined by a catalog view, you use a SELECT statement just as you would for any other view. The SELECT statement shown in this figure, for example, displays the name and schema of every table in the current database. To do that, it joins the sys.tables and sys.schemas views on the schema_id column in each view.

If you look up the sys.tables view in the SQL Server documentation, you'll notice that this table doesn't include the name column that's retrieved by the SELECT statement in this figure. Instead, this column is inherited from the sys.objects view. A view like this that contains columns that can be inherited by other views is called a *base view*, and the view that inherits the columns is called the *derived view*. Because the columns of the base view are inherited automatically, you can think of these columns as part of the derived view.

Before catalog views were introduced with SQL Server 2005, you used *information schema views* to query the system catalog. Although these views are still available, we recommend you use the catalog views instead. That's because, unlike the information schema views, the catalog views provide access to all of the data in the system catalogs. In addition, the catalog views are more efficient than the information schema views.

At this point, you may be wondering why you would want to use the catalog views. After all, you can get the same information using the Management Studio. The answer is that you may occasionally need to get information about the objects in a database from a script. You'll learn how to do that in the next chapter.

Some of the SQL Server catalog views

View name	Contents
<code>sys.schemas</code>	One row for each schema in the current database.
<code>sys.sequences</code>	One row for each sequence in the current database.
<code>sys.tables</code>	One row for each table in the current database.
<code>sys.views</code>	One row for each view in the current database.
<code>sys.columns</code>	One row for each column in each table, view, or table-valued function in the current database.
<code>sys.key_constraints</code>	One row for each primary or unique key in each table in the current database.
<code>sys.foreign_keys</code>	One row for each foreign key.
<code>sys.foreign_key_columns</code>	One row for each column or set of columns that make up a foreign key.
<code>sys.objects</code>	One row for each user-defined object in the current database, except for triggers.

A SELECT statement that retrieves the name and schema of each table

```
SELECT sys.tables.name AS TableName, sys.schemas.name AS SchemaName
FROM sys.tables INNER JOIN sys.schemas
ON sys.tables.schema_id = sys.schemas.schema_id;
```

The result set

	TableName	SchemaName
1	ContactUpdates	dbo
2	GLAccounts	dbo
3	InvoiceArchive	dbo
4	InvoiceLineItem	dbo
5	Invoices	dbo
6	Terms	dbo
7	Vendors	dbo
8	sysdiagrams	dbo
9	VendorCopy	dbo

Description

- You can use the *catalog views* to examine the *system catalog*, which lists all of the system objects that define a database, including tables, views, columns, keys, and constraints.
- Some catalog views inherit columns from other catalog views. In that case, the catalog view from which the columns are inherited is called the *base view*, and the catalog view that inherits the columns is called the *derived view*.
- For a complete listing of the catalog views, refer to SQL Server documentation.

Figure 13-9 How to use the catalog views

How to use the View Designer

The Management Studio provides a graphical tool called the View Designer that you can use to work with views. However, many programmers prefer to use the Query Editor to manually code the SQL for views as described earlier. As a result, this topic only provides a brief description of the View Designer.

How to create or modify a view

You can use the View Designer to create or modify a view as described in figure 13-10. This figure shows a view named VendorPayment in the View Designer. This tool is similar to the Query Designer that was briefly introduced in chapter 2.

To create a new view, right-click on the Views folder and select the New View command. When you do, the View Designer prompts you to select the tables that the view will be based on. Then, it displays the tables that you select in the Diagram pane of the View Designer. When appropriate, the Diagram pane includes a link icon that shows the relationships between these tables. When you save the view for the first time, the Management Studio displays a dialog box that allows you to enter a name for the view.

To edit the design of an existing view, you can expand the Views folder, right-click on the view, and select the Design command to display the view in the View Designer. If necessary, you can use the Add Table button in the View Designer toolbar to add new tables to the Diagram pane.

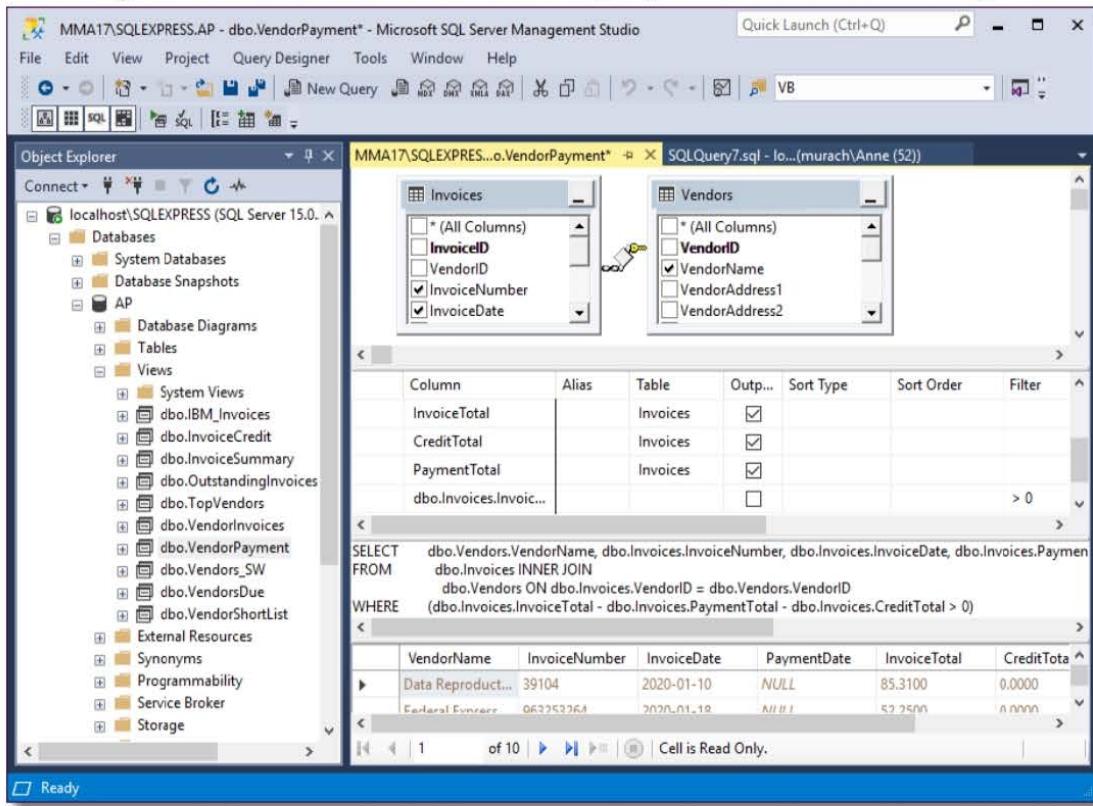
Once you display the tables for the view in the Diagram pane, you can use that pane to select the columns that are displayed in the Criteria pane. Then, you can use the Criteria pane to set the criteria and sort sequence for the query. In this figure, for example, the Criteria pane shows the columns that will be included in the view, and it shows that the last column, which is a calculated column and is not included in the result set, must be greater than zero.

As you work in the Diagram and Criteria panes, the View Designer generates a SQL statement and displays it in the SQL pane. When you have the statement the way you want it, you can test the view by clicking on the Execute SQL button in the View Designer toolbar. Then, the data that's returned by the view is displayed in the Results pane. In this figure, for example, the Results pane displays the results as read-only because the view that's defined by the SQL statement uses a calculated column.

How to delete a view

You can also use the Management Studio to delete a view. You do that using the same technique you use to delete any other type of database object. To start, right-click on the view and select Delete. Then, select OK to confirm the delete.

The Management Studio with a view displayed in the View Designer



How to create or modify the design of a view

- To create a new view, right-click on the Views folder and select the New View command to display a new view in the View Designer. Then, when you click on the Save button in the toolbar, you can supply a name for the view.
- To edit the design of an existing view, expand the Views folder, right-click on the view, and select the Design command to display the view in the View Designer.
- To add tables to the Diagram pane, click on the Add Table button in the View Designer toolbar.
- To select the columns for a view, use the Diagram pane.
- To specify the selection criteria and sort order for the view, use the Criteria pane.
- To view the code that's generated for the view or to modify the generated code, use the SQL pane.
- To display the results of the view in the Results pane, click on the Execute SQL button in the View Designer toolbar.

How to delete a view

- To delete a view, expand the Views folder, right-click on the view, select the Delete command, and click OK in the resulting Delete Object dialog box.

Figure 13-10 How to use the Management Studio to work with views

Perspective

In this chapter, you learned how to create and use views. As you've seen, views provide a powerful and flexible way to predefine the data that can be retrieved from a database. By using them, you can restrict the access to a database while providing a consistent and simplified way for end users and application programs to access that data.

Terms

view	read-only view
viewed table	catalog views
base table	base view
nested view	derived view
database schema	system catalog
updatable view	information schema view

Exercises

1. Write a CREATE VIEW statement that defines a view named InvoiceBasic that returns three columns: VendorName, InvoiceNumber, and InvoiceTotal. Then, write a SELECT statement that returns all of the columns in the view, sorted by VendorName, where the first letter of the vendor name is N, O, or P.
2. Create a view named Top10PaidInvoices that returns three columns for each vendor: VendorName, LastInvoice (the most recent invoice date), and SumOfInvoices (the sum of the InvoiceTotal column). Return only the 10 vendors with the largest SumOfInvoices and include only paid invoices.
3. Create an updatable view named VendorAddress that returns the VendorID, both address columns, and the city, state, and zip code columns for each vendor. Then, write a SELECT query to examine the result set where VendorID=4. Next, write an UPDATE statement that changes the address so that the suite number (Ste 260) is stored in VendorAddress2 rather than in VendorAddress1. To verify the change, rerun your SELECT query.
4. Write a SELECT statement that selects all of the columns for the catalog view that returns information about foreign keys. How many foreign keys are defined in the AP database?
5. Using the Management Studio, modify the InvoiceBasic view created in exercise 1 to sort the result set by VendorName. What clause does the system automatically code to allow the use of an ORDER BY clause in the view?

How to code scripts

At the end of chapter 11, you saw a simple script that defines the AP database and the tables it contains. Now, this chapter teaches you how to code more complex scripts. With the skills you'll learn in this chapter, you'll be able to code scripts with functionality that's similar to the functionality provided by procedural programming languages like C#, Visual Basic, and Java.

If you have experience with another procedural programming language, you shouldn't have any trouble with the skills presented in this chapter. However, you should know that the programming power of Transact-SQL is limited when compared to other languages. That's because Transact-SQL is designed specifically to work with SQL Server databases rather than as a general-purpose programming language. For its intended use, Transact-SQL programming is powerful and flexible.

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
Perspective	454

An introduction to scripts

To start, this chapter reviews and expands on the script concepts you learned in chapter 11. Then, it summarizes the Transact-SQL statements you can use within scripts. Most of these statements will be presented in detail later in this chapter.

How to work with scripts

Most of the *scripts* you've created so far in this book have consisted of a single SQL statement. However, a script can include any number of statements, and those statements can be divided into one or more *batches*. To indicate the end of a batch, you code a GO command. The script in figure 14-1, for example, consists of two batches. The first one creates a database, and the second one creates three tables in that database.

Because the new database must exist before you can add tables to it, the CREATE DATABASE statement must be coded in a separate batch that's executed before the CREATE TABLE statements. By contrast, the three CREATE TABLE statements don't have to be in separate batches. However, notice that these three statements are coded in a logical sequence within the second batch. In this case, the CommitteeAssignments table references the other two tables, so I created the other tables first. If I had created the CommitteeAssignments table first, I couldn't have declared the foreign key constraints. In that case, I would have had to add these constraints in an ALTER TABLE statement after the other two tables were created.

Although you don't have to code the CREATE TABLE statement in a separate batch, you do have to code the five statements listed in this figure in separate batches. Each of these statements must be the first and only statement in the batch. You learned how to code the CREATE VIEW statement in the last chapter, you'll learn how to code the CREATE PROCEDURE, CREATE FUNCTION, and CREATE TRIGGER statements in the next chapter, and you'll learn how to code the CREATE SCHEMA statement in chapter 17.

Before I go on, you should realize that GO isn't a Transact-SQL statement. Instead, it's a command that's interpreted by two of the software tools that are included with SQL Server: the Management Studio and the SQLCMD utility. When one of these tools encounters a GO command, it sends the preceding statements to the server to be executed. You already know how to use the Management Studio, and you'll learn the basics of working with the SQLCMD utility later in this chapter.

A script with two batches

```
/*
Creates three tables in a database named ClubRoster.
Author: Bryan Syverson
Created: 2008-08-12
Modified: 2016-09-26
*/

CREATE DATABASE ClubRoster;
GO

USE ClubRoster;

CREATE TABLE Members
(MemberID int NOT NULL IDENTITY PRIMARY KEY,
LastName varchar(75) NOT NULL,
FirstName varchar(50) NOT NULL,
MiddleName varchar(50) NULL);

CREATE TABLE Committees
(CommitteeID int NOT NULL IDENTITY PRIMARY KEY,
CommitteeName varchar(50) NOT NULL);

CREATE TABLE CommitteeAssignments
(MemberID int NOT NULL REFERENCES Members(MemberID),
CommitteeID int NOT NULL REFERENCES Committees(CommitteeID));
```

Statements that must be in their own batch

CREATE VIEW	CREATE PROCEDURE	CREATE FUNCTION
CREATE TRIGGER	CREATE SCHEMA	

Description

- A *script* is a series of SQL statements that you can store in a file. Each script can contain one or more *batches* that are executed as a unit.
- To signal the end of a batch, you use the GO command. A GO command isn't required after the last batch in a script or for a script that contains a single batch.
- If a statement must be executed before the statements that follow can succeed, you should include a GO command after it.
- The statements within a batch are executed in the order that they appear in the batch. Because of that, you need to code statements that depend on other statements after the statements they depend on.
- If you create a database within a script, you have to execute the batch that contains the CREATE DATABASE statement before you can execute other statements that refer to the database.
- The five statements listed above (CREATE VIEW, CREATE PROCEDURE, CREATE FUNCTION, CREATE TRIGGER, and CREATE SCHEMA) can't be combined with other statements in a batch.
- If a script will be used with a production database, you should include documentation as shown above. Additional information should be included when appropriate.

Figure 14-1 How to work with scripts

The Transact-SQL statements for script processing

Figure 14-2 presents the Transact-SQL statements used to process scripts. These statements, which are sometimes referred to as *T-SQL statements*, are specific to SQL Server. You'll learn how to code many of these statements throughout this chapter.

Two statements I want to present right now are USE and PRINT. You can see both of these statements in the script presented in this figure. You use the USE statement to change the current database within a script. In this example, the USE statement makes the AP database the current database. That way, you don't have to worry about setting the current database using the drop-down list in the Management Studio. And when you create stored procedures, functions, and triggers as you'll learn in the next chapter, you have to use the USE statement.

You use the PRINT statement to return a message to the client. If the client is the Management Studio, for example, the message is displayed in the Messages tab of the Query Editor. The script in this figure includes two PRINT statements. Notice that the first statement uses concatenation to combine a literal string with the value of a variable. You'll learn how to work with variables as well as the other statements in this script in a moment.

Two statements I won't present in this chapter are GOTO and RETURN. I recommend that you don't use the GOTO statement because it can make your scripts difficult to follow. And the RETURN statement is used most often with stored procedures, so I'll present it in the next chapter.

Transact-SQL statements for controlling the flow of execution

Keyword	Description
IF...ELSE	Controls the flow of execution based on a condition.
BEGIN...END	Defines a statement block.
WHILE	Repeats statements while a specific condition is true.
BREAK	Exits the innermost WHILE loop.
CONTINUE	Returns to the beginning of a WHILE loop.
TRY...CATCH	Controls the flow of execution when an error occurs.
GOTO	Unconditionally changes the flow of execution.
RETURN	Exits unconditionally.

Other Transact-SQL statements for script processing

Keyword	Description
USE	Changes the database context to the specified database.
PRINT	Returns a message to the client.
DECLARE	Declares a local variable.
SET	Sets the value of a local variable or a session variable.
EXEC	Executes a dynamic SQL statement or stored procedure.

The syntax of the USE statement

```
USE database
```

The syntax of the PRINT statement

```
PRINT string_expression
```

A script that uses some of the statements shown above

```
USE AP;
DECLARE @TotalDue money;
SET @TotalDue = (SELECT SUM(InvoiceTotal - PaymentTotal - CreditTotal)
                 FROM Invoices);
IF @TotalDue > 0
    PRINT 'Total invoices due = $' + CONVERT(varchar,@TotalDue,1);
ELSE
    PRINT 'Invoices paid in full';
```

Description

- These statements are used within SQL scripts to add functionality similar to that provided by procedural programming languages.
- These statements are part of the Transact-SQL, or *T-SQL*, language and aren't available on SQL-based systems other than SQL Server.

Figure 14-2 The Transact-SQL statements for script processing

How to work with variables and temporary tables

If you need to store values within a script, you can store them in scalar variables, table variables, or temporary tables. You'll learn how to use all three of these techniques in the topics that follow. In addition, you'll see a comparison of the different types of SQL Server objects that you can use to work with table data so you'll know when to use each type.

How to work with scalar variables

Figure 14-3 presents the DECLARE and SET statements that you use to work with *variables*. Specifically you use these statements to work with *scalar variables*, which can contain a single value. You use the DECLARE statement to create a variable and specify the type of data it can contain, and you use the SET statement to assign a value to a variable.

The variables you create using the DECLARE statement are also known as *local variables*. That's because a variable's scope is limited to a single batch. In other words, you can't refer to a variable from outside the batch. Variables are also described as local to distinguish them from *global variables*, which is an obsolete term for system functions. You'll learn about some of the system functions later in this chapter.

You can also assign a value to a variable within the select list of a SELECT statement. To do that, you use the alternate syntax shown in this figure. Although you can accomplish the same thing by using a SET statement to assign the result of a SELECT query to the variable, the alternate syntax usually results in more readable code. In addition, when you use a SELECT statement, you can assign values to two or more variables with a single statement.

The script shown in this figure uses five variables to calculate the percent difference between the minimum and maximum invoices for a particular vendor. This script starts by declaring all of these variables. Then, it assigns values to two of the variables using SET statements. Notice that the second SET statement assigns the result of a SELECT statement to the variable, and the value of the first variable is used in the WHERE clause of that SELECT statement. The SELECT statement that follows this SET statement uses the alternate syntax to assign values to two more variables. Then, the next SET statement assigns the result of an arithmetic expression to the final variable. Finally, PRINT statements are used to display the values of four of the variables.

Although you can use a variable in any expression, you can't use it in place of a keyword or an object name. For example, this use is invalid:

```
DECLARE @TableNameVar varchar(128);
SET @TableNameVar = 'Invoices';
SELECT * FROM @TableNameVar;
```

Later in this chapter, however, you'll learn how to execute a SQL statement like this one using dynamic SQL.

The syntax of the DECLARE statement for scalar variables

```
DECLARE @variable_name_1 data_type [, @variable_name_2 data_type]...
```

The syntax of the SET statement for a scalar variable

```
SET @variable_name = expression
```

An alternate syntax for setting a variable's value in a select list

```
SELECT @variable_name_1 = column_specification_1
[, @variable_name_2 = column_specification_2]...
```

A SQL script that uses variables

```
USE AP;
DECLARE @MaxInvoice money, @MinInvoice money;
DECLARE @PercentDifference decimal(8,2);
DECLARE @InvoiceCount int, @VendorIDVar int;

SET @VendorIDVar = 95;
SET @MaxInvoice = (SELECT MAX(InvoiceTotal) FROM Invoices
WHERE VendorID = @VendorIDVar);
SELECT @MinInvoice = MIN(InvoiceTotal), @InvoiceCount = COUNT(*)
FROM Invoices
WHERE VendorID = @VendorIDVar;
SET @PercentDifference = (@MaxInvoice - @MinInvoice) / @MinInvoice * 100;

PRINT 'Maximum invoice is $' + CONVERT(varchar,@MaxInvoice,1) + '.';
PRINT 'Minimum invoice is $' + CONVERT(varchar,@MinInvoice,1) + '.';
PRINT 'Maximum is ' + CONVERT(varchar,@PercentDifference) +
'% more than minimum.';
PRINT 'Number of invoices: ' + CONVERT(varchar,@InvoiceCount) + '.'
```

The response from the system

```
Maximum invoice is $46.21.
Minimum invoice is $16.33.
Maximum is 182.97% more than minimum.
Number of invoices: 6.
```

Description

- A *variable* is used to store data. To create a variable, you use the DECLARE statement. The initial value of a variable is always null.
- A *scalar variable* is defined with a standard data type and contains a single value. You can also create table variables to store an entire result set.
- The name of a variable must always start with an at sign (@). Whenever possible, you should use long, descriptive names for variables.
- The scope of a variable is the batch in which it's defined, which means that it can't be referred to from outside that batch. Because of that, variables are often called *local variables*.
- To assign a value to a variable, you can use the SET statement. Alternatively, you can use the SELECT statement to assign a value to one or more variables.
- You can use a variable in any expression, but you can't use it in place of an object name or a keyword.

How to work with table variables

Figure 14-4 presents the syntax of the DECLARE statement you use to create table variables. A *table variable* is a variable that can store the contents of an entire table. To create this type of variable, you specify the table data type in the DECLARE statement rather than one of the standard SQL data types. Then, you define the columns and constraints for the table using the same syntax that you use for the CREATE TABLE statement.

The script shown in this figure illustrates how you might use a table variable. Here, a DECLARE statement is used to create a table variable named @BigVendors that contains two columns: VendorID and VendorName. Then, an INSERT statement is used to insert all of the rows from the Vendors table for vendors that have invoices totaling over \$5000 into this table variable. Finally, a SELECT statement is used to retrieve the contents of the table variable.

Notice that the table variable in this example is used in place of a table name in the INSERT and SELECT statements. You can also use a table variable in place of a table name in an UPDATE or DELETE statement. The only place you can't use a table variable instead of a table name is in the INTO clause of a SELECT INTO statement.

The syntax of the DECLARE statement for a table variable

```
DECLARE @table_name TABLE  
  (column_name_1 data_type [column_attributes]  
  [, column_name_2 data_type [column_attributes]]...  
  [, table_attributes])
```

A SQL script that uses a table variable

```
USE AP;  
  
DECLARE @BigVendors table  
  (VendorID int,  
  VendorName varchar(50));  
  
INSERT @BigVendors  
SELECT VendorID, VendorName  
FROM Vendors  
WHERE VendorID IN  
  (SELECT VendorID FROM Invoices WHERE InvoiceTotal > 5000);  
  
SELECT * FROM @BigVendors;
```

The result set

	VendorID	VendorName
1	72	Data Reproductions Corp
2	99	Bertelsmann Industry Svcs. Inc
3	104	Digital Dreamworks
4	110	Malloy Lithographing Inc

Description

- A *table variable* can store an entire result set rather than a single value. To create a table variable, use a DECLARE statement with the table data type.
- You use the same syntax for defining the columns of a table variable as you do for defining a new table with the CREATE TABLE statement. See figure 11-4 in chapter 11 for details.
- Like a scalar variable, a table variable has local scope, so it's available only within the batch where it's declared.
- You can use a table variable like a standard table within SELECT, INSERT, UPDATE, and DELETE statements. The exception is that you can't use it within the INTO clause of a SELECT INTO statement.

Figure 14-4 How to work with table variables

How to work with temporary tables

In addition to table variables, you can use *temporary tables* to store table data within a script. Temporary tables are useful for storing table data within a complex script. In addition, they provide a way for you to test queries against temporary data rather than permanent data.

Unlike a table variable, a temporary table exists for the duration of the database session in which it's created. If you create a temporary table in the Management Studio's Query Editor, for example, it exists as long as the Query Editor is open. As a result, you can refer to the table from more than one script.

Figure 14-5 presents two scripts that use temporary tables. The first script creates a temporary table named #TopVendors using a SELECT INTO query. This temporary table contains the VendorID and average invoice total for the vendor with the greatest average. Then, the second SELECT statement joins the temporary table with the Invoices table to get the date of the most recent invoice for that vendor. Note, however, that you could have created the same result set using a derived table like this:

```
WITH TopVendors AS
(
    SELECT TOP 1 VendorID, AVG(InvoiceTotal) AS AvgInvoice
    FROM Invoices
    GROUP BY VendorID
    ORDER BY AvgInvoice DESC
)
SELECT Invoices.VendorID, MAX(InvoiceDate) AS LatestInv
FROM Invoices JOIN TopVendors
    ON Invoices.VendorID = TopVendors.VendorID
GROUP BY Invoices.VendorID;
```

Because derived tables are more efficient to use than temporary tables, you should use them whenever possible.

The second script in this figure shows another use of a temporary table. This script creates a temporary table that contains two columns: an identity column and a character column with a nine-digit default value that's generated using the RAND function. Then, the script inserts two rows into this table using the default values. Finally, the script uses a SELECT statement to retrieve the contents of the table. A script like this can be useful during testing.

In these examples, the name of a temporary table begins with a number sign (#). If the name begins with a single number sign, the table is defined as a *local temporary table*, which means that it's visible only to the database session in which it's created. However, you can also create temporary tables that are visible to all open database sessions, called *global temporary tables*. To create a global temporary table, code two number signs at the beginning of the table name.

When a database session ends, any temporary tables created during that session are deleted. If you want to delete a temporary table before the session ends, however, you can do that by issuing a DROP TABLE statement.

A script that uses a local temporary table instead of a derived table

```
SELECT TOP 1 VendorID, AVG(InvoiceTotal) AS AvgInvoice
INTO #TopVendors
FROM Invoices
GROUP BY VendorID
ORDER BY AvgInvoice DESC;

SELECT Invoices.VendorID, MAX(InvoiceDate) AS LatestInv
FROM Invoices JOIN #TopVendors
    ON Invoices.VendorID = #TopVendors.VendorID
GROUP BY Invoices.VendorID;
```

The result set

	VendorID	LatestInv
1	110	2020-01-31

A script that creates a global temporary table of random numbers

```
CREATE TABLE ##RandomSSNs
(
    SSN_ID int      IDENTITY,
    SSN     char(9) DEFAULT
            LEFT(CAST(CAST(CEILING(RAND()*10000000000)AS bigint)AS varchar),9)
);

INSERT ##RandomSSNs VALUES (DEFAULT);
INSERT ##RandomSSNs VALUES (DEFAULT);

SELECT * FROM ##RandomSSNs;
```

The result set

	SSN_ID	SSN
1	1	217589782
2	2	439515826

Description

- A *temporary table* exists only during the current database session. In the Management Studio, that means that the table is available until you close the window where you created the table.
- Temporary tables are stored in the system database named tempdb.
- If you need to drop a temporary table before the end of the current session, you can do that using the DROP TABLE statement.
- Temporary tables are useful for testing queries or for storing data temporarily in a complex script.
- A *local temporary table* is visible only within the current session, but a *global temporary table* is visible to all sessions. To identify a local temporary table, you prefix the name with a number sign (#). To identify a global temporary table, you prefix the name with two number signs (##). Temporary table names are limited to 116 characters.
- Because derived tables result in faster performance than temporary tables, you should use derived tables whenever possible. See figure 14-6 for details.

Figure 14-5 How to work with temporary tables

A comparison of the five types of Transact-SQL table objects

Now that you've learned about table variables and temporary tables, you might want to consider when you'd use them within a script and when you'd create a new standard table or view or simply use a derived table instead. Figure 14-6 presents a comparison of these five types of table objects. Note that although a view isn't technically a table, I've included it in this figure because it can be used in place of a table.

One of the biggest differences between these objects is their *scope*, which determines where it can be used in a script. Because standard tables and views are stored permanently within a database, they have the broadest scope and can be used anywhere, including in other scripts on the current connection or other scripts on other connections. By contrast, a derived table exists only while the query that creates it is executing. Because of that, a derived table can't be referred to from outside the query. As you've just learned, temporary tables and table variables fall somewhere in between.

Another difference between the five table types is where they're stored. Like standard tables, temporary tables are stored on disk. By contrast, table variables and derived tables are stored in memory if they're relatively small. Because of that, table variables and derived tables usually take less time to create and access than standard or temporary tables.

Although a view is also stored on disk, it can be faster to use than any of the other table objects. That's because it's simply a precompiled query, so it takes less time to create and access than an actual table. However, with the other table objects, you can insert, update, or delete data without affecting any of the base tables in your database, which isn't true of a view. For this reason, you can't use a view in the same way as the other table objects. But if you find that you're creating a table object that doesn't need to be modified within your script, then you should be defining it as a view instead.

In most scripts, table variables and temporary tables can be used interchangeably. Since a script that uses a table variable will outperform the same script with a temporary table, you should use table variables whenever possible. However, table variables are dropped when the batch finishes execution. So if you need to use the table in other batches, you'll need to use a temporary table instead.

The five types of Transact-SQL table objects

Type	Scope
Standard table	Available within the system until explicitly deleted.
Temporary table	Available within the system while the current database session is open.
Table variable	Available within a script while the current batch is executing.
Derived table	Available within a statement while the current statement is executing.
View	Available within the system until explicitly deleted.

Description

- Within a Transact-SQL script, you often need to work with table objects other than the base tables in your database.
- The *scope* of a table object determines what code has access to that table.
- Standard tables and views are stored permanently on disk until they are explicitly deleted, so they have the broadest scope and are therefore always available for use.
- Derived tables and table variables are generally stored in memory, so they can provide the best performance. By contrast, standard tables and temporary tables are always stored on disk and therefore provide slower performance.
- To improve the performance of your scripts, use a derived table instead of creating a table variable. However, if you need to use the table in other batches, create a temporary table. Finally, if the data needs to be available to other connections to the database, create a standard table or, if possible, a view.
- Although a view isn't a table, it can be used like one. Views provide fast performance since they're predefined, and high availability since they're permanent objects. For these reasons, you should try to use a view rather than create a table whenever that's possible. However, if you need to insert, delete, or update the data in the table object without affecting the base tables of your database, then you can't use a view.
- A common table expression (CTE) is a type of derived table. For more information about CTEs, see chapter 6.

Figure 14-6 A comparison of the five types of Transact-SQL tables

How to control the execution of a script

The ability to control the execution of a program is an essential feature of any procedural programming language. T-SQL provides three basic control structures that you can use within scripts. You can use the first one to perform conditional processing, you can use the second one to perform repetitive processing, and you can use the third one to handle errors. You'll learn how to use the statements that implement these structures in the topics that follow.

How to perform conditional processing

To execute a statement or a block of statements based on a condition, you use the IF...ELSE statement. This statement is presented in figure 14-7. When an IF...ELSE statement is executed, SQL Server evaluates the conditional expression after the IF keyword. If this condition is true, the statement or block of statements after the IF keyword is executed. Otherwise, the statement or block of statements after the ELSE keyword is executed if this keyword is included.

The first script in this figure uses a simple IF statement to test the value of a variable that's assigned in a SELECT statement. This variable contains the oldest invoice due date in the Invoices table. If this value is less than the current date, the PRINT statement that follows the IF keyword is executed. Otherwise, no action is taken.

In the second script, the logic of the first script has been enhanced. Here, a block of statements is executed if the oldest due date is less than the current date. Notice that this block of statements begins with the BEGIN keyword and ends with the END keyword. In addition, an ELSE clause has been added. Then, if the oldest due date is greater than or equal to the current date, a PRINT statement is executed to indicate that none of the invoices are overdue.

Notice the comment that follows the ELSE keyword. This comment describes the expression that would result in this portion of code being executed. Although it isn't required, this programming practice makes it easier to find and debug logical errors, especially if you're *nesting* IF...ELSE statements within other IF...ELSE statements.

The syntax of the IF...ELSE statement

```
IF Boolean_expression
    {statement|BEGIN...END}
[ELSE
    {statement|BEGIN...END}]
```

A script that tests for outstanding invoices with an IF statement

```
USE AP;
DECLARE @EarliestInvoiceDue date;
SELECT @EarliestInvoiceDue = MIN(InvoiceDueDate) FROM Invoices
    WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
IF @EarliestInvoiceDue < GETDATE()
    PRINT 'Outstanding invoices overdue!';
```

The response from the system

Outstanding invoices overdue!

An enhanced version of the same script that uses an IF...ELSE statement

```
USE AP;
DECLARE @MinInvoiceDue money, @MaxInvoiceDue money;
DECLARE @EarliestInvoiceDue date, @LatestInvoiceDue date;
SELECT @MinInvoiceDue = MIN(InvoiceTotal - PaymentTotal - CreditTotal),
    @MaxInvoiceDue = MAX(InvoiceTotal - PaymentTotal - CreditTotal),
    @EarliestInvoiceDue = MIN(InvoiceDueDate),
    @LatestInvoiceDue = MAX(InvoiceDueDate)
FROM Invoices
WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;
IF @EarliestInvoiceDue < GETDATE()
    BEGIN
        PRINT 'Outstanding invoices overdue!';
        PRINT 'Dated ' + CONVERT(varchar,@EarliestInvoiceDue,1) +
            ' through ' + CONVERT(varchar,@LatestInvoiceDue,1) + '.';
        PRINT 'Amounting from $' + CONVERT(varchar,@MinInvoiceDue,1) +
            ' to $" + CONVERT(varchar,@MaxInvoiceDue,1) + '.';
    END;
ELSE --@EarliestInvoiceDue >= GETDATE()
    PRINT 'No overdue invoices.';
```

The response from the system

Outstanding invoices overdue!
 Dated 02/09/20 through 02/29/20.
 Amounting from \$30.75 to \$19,351.18.

Description

- You use the IF...ELSE statement to test a conditional expression. If that expression is true, the statements that follow the IF keyword are executed. Otherwise, the statements that follow the ELSE keyword are executed if that keyword is included.
- If you need to execute two or more SQL statements within an IF or ELSE clause, enclose them within a BEGIN...END block.
- You can *nest* IF...ELSE statements within other IF...ELSE statements. Although SQL Server doesn't limit the number of nested levels, you should avoid nesting so deeply that your script becomes difficult to read.

How to test for the existence of a database object

Frequently, you'll need to write scripts that create and work with database objects. If you try to create an object that already exists, SQL Server will return an error. Similarly, SQL Server will return an error if you try to work with an object that doesn't exist. To avoid these types of errors, you should check for the existence of an object before you create or work with it.

If you're working with SQL Server 2016 or later, you can add the IF EXISTS clause to a DROP statement to check whether the object exists before you drop it. In figure 14-8, for instance, the first example adds this clause to a DROP statement that drops a database. As a result, if this database exists, it's dropped. If not, the script continues without returning an error. Although this example shows how to work with a database, the IF EXISTS clause also works with statements that drop other database objects such as tables, views, stored procedures, user-defined functions, and triggers.

This example begins with a USE statement to change the current database to something other than the database you're testing. That's because a database can't be deleted if it's currently in use.

If you're working with an older version of SQL Server, or if you want to perform another task besides dropping an object, you can use the OBJECT_ID function to check for the existence of a table, view, stored procedure, user-defined function, or trigger. Or, you use the DB_ID function to check for the existence of a database. If the specified object exists, these functions return the unique identification number assigned to that object by SQL Server. Otherwise, they return a null value.

You can use these functions within an IF...ELSE statement to test for a null return value. For instance, the second example uses the DB_ID function to test for the existence of a database. If the database already exists, this example executes a DROP DATABASE statement to delete it.

The third example tests for the existence of a table name InvoiceCopy. Then, if the table exists, the example executes a DROP TABLE statement to delete it. However, when you use the OBJECT_ID function, you may not know what type of object you're dealing with. For example, InvoiceCopy could also be the name of a view or a stored procedure. In that case, the DROP TABLE statement would cause an error.

To avoid this situation, you can use the technique shown in the fourth example. Instead of using a function, this example uses information in the catalog view named tables to determine if a table named InvoiceCopy exists. If it does, the table is deleted. Otherwise, no action is taken. You can use similar code to check for the existence of a view using the sys.views catalog view.

The last example tests for the existence of a temporary table. Here, the table name is qualified with the name of the database that contains temporary tables, tempdb. You can omit the schema qualification, though, since this is a system database.

The syntax for the IF EXISTS clause (SQL Server 2016 and later)

```
DROP OBJECT_TYPE IF EXISTS object_name;
```

An example that uses the IF EXISTS clause

```
USE master;
DROP DATABASE IF EXISTS TestDB;
```

The syntax of the OBJECT_ID function

```
OBJECT_ID('object')
```

The syntax of the DB_ID function

```
DB_ID('database')
```

Examples that use the OBJECT_ID and DB_ID functions

Code that tests whether a database exists before it deletes it

```
USE master;
IF DB_ID('TestDB') IS NOT NULL
    DROP DATABASE TestDB;

CREATE DATABASE TestDB;
```

Code that tests for the existence of a table

```
IF OBJECT_ID('InvoiceCopy') IS NOT NULL
    DROP TABLE InvoiceCopy;
```

Another way to test for the existence of a table

```
IF EXISTS (SELECT * FROM sys.tables
            WHERE name = 'InvoiceCopy')
    DROP TABLE InvoiceCopy;
```

Code that tests for the existence of a temporary table

```
IF OBJECT_ID('tempdb..#AllUserTables') IS NOT NULL
    DROP TABLE #AllUserTables;
```

Description

- With SQL Server 2016 and later, you can add the IF EXISTS clause to a DROP statement to check whether an object exists before you drop it.
- With earlier versions of SQL Server, you can use the OBJECT_ID and DB_ID functions within IF statements to check whether an object exists.
- You can use the OBJECT_ID function to check for the existence of a table, view, stored procedure, user-defined function, or trigger. You use the DB_ID function to check for the existence of a database. Both functions return a null value if the object doesn't exist. Otherwise, they return the object's identification number.
- To test for the existence of a temporary table, you must qualify the table name with the database that contains it: tempdb. Since this is a system database, though, you can omit the schema name as shown above.

Figure 14-8 How to test for the existence of a database object

How to perform repetitive processing

In some cases, you'll need to repeat a statement or a block of statements while a condition is true. To do that, you use the WHILE statement that's presented in figure 14-9. This coding technique is referred to as a *loop*.

The script in this figure illustrates how the WHILE statement works. Here, a WHILE loop is used to adjust the credit amount of each invoice in the Invoices table that has a balance due until the total balance due is less than \$20,000. Although this example is unrealistic, it will help you understand how the WHILE statement works. A more realistic example would be to use a WHILE statement to process cursors, which you'll learn about in the next figure.

This script starts by creating a copy of the Invoices table named InvoiceCopy that contains just the invoices that have a balance due. Since the WHILE statement will change the data in the table, this prevents corruption of the data in the source table. Then, the expression in the WHILE statement uses a SELECT statement to retrieve the sum of the invoice balances in this table. If the sum is greater than or equal to 20,000, the block of statements that follows is executed. Otherwise, the loop ends.

The UPDATE statement within the WHILE loop adds five cents to the CreditTotal column of each invoice that has a balance due. (Although the table initially contains only invoices that have a balance due, that may change as credits are applied to the invoices within the loop.) Then, an IF statement tests the maximum credit amount in the table to see if it's more than 3000. If it is, a BREAK statement is used to terminate the loop. Because this statement can make your scripts difficult to read and debug, I recommend you use it only when necessary. In this case, it's used only for illustrative purposes.

If the maximum credit total is less than or equal to 3000, the CONTINUE statement is executed. This statement causes control to return to the beginning of the loop. Then, the condition for the loop is tested again, and if it's true, the statements within the loop are processed again.

Note that because the CONTINUE statement is the last statement in the loop, it's not required. That's because control will automatically return to the beginning of the loop after the last statement in the loop is executed. For example, this code would produce the same result:

```
BEGIN
  ...
  IF (SELECT MAX(CreditTotal) FROM #InvoiceCopy) > 3000
    BREAK;
END;
```

Sometimes, though, the CONTINUE statement can clarify the logic of an IF statement, as it does in the example in this figure. In addition, since this statement returns control to the beginning of the loop, it can be used in an IF clause to bypass the remaining statements in the loop. However, like the BREAK statement, this makes your code confusing to read, so I recommend you code your IF statements in such a way that you avoid using the CONTINUE statement whenever possible.

The syntax of the WHILE statement

```
WHILE expression
  {statement|BEGIN...END}
  [BREAK]
  [CONTINUE]
```

A script that tests and adjusts credit amounts with a WHILE loop

```
USE AP;
IF OBJECT_ID('tempdb..#InvoiceCopy') IS NOT NULL
  DROP TABLE #InvoiceCopy;

SELECT * INTO #InvoiceCopy FROM Invoices
WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0;

WHILE (SELECT SUM(InvoiceTotal - CreditTotal - PaymentTotal)
       FROM #InvoiceCopy) >= 20000
BEGIN
  UPDATE #InvoiceCopy
  SET CreditTotal = CreditTotal + .05
  WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0;

  IF (SELECT MAX(CreditTotal) FROM #InvoiceCopy) > 3000
    BREAK;
  ELSE --(SELECT MAX(CreditTotal) FROM #InvoiceCopy) <= 3000
    CONTINUE;
END;

SELECT InvoiceDate, InvoiceTotal, CreditTotal
FROM #InvoiceCopy;
```

The result set

	InvoiceDate	InvoiceTotal	CreditTotal
1	2020-01-10	85.31	0.00
2	2020-01-18	52.25	0.00
3	2020-01-21	579.42	0.00
4	2020-01-21	59.97	0.00

Description

- To execute a SQL statement repeatedly, you use the WHILE statement. This statement is executed as long as the conditional expression in the WHILE clause is true.
- If you need to execute two or more SQL statements within a WHILE *loop*, enclose the statements within BEGIN and END keywords.
- To exit from a WHILE loop immediately without testing the expression, use the BREAK statement. To return to the beginning of a WHILE loop without executing any additional statements in the loop, use the CONTINUE statement.

Warning

- This script takes a few seconds to execute.

Figure 14-9 How to perform repetitive processing

How to use a cursor

By default, SQL statements work with an entire result set rather than individual rows. However, you may sometimes need to work with the data in a result set one row at a time. To do that, you can use a *cursor* as described in figure 14-10.

In this figure, the script begins by declaring three variables. Then, it assigns a value of 0 to the third variable, @UpdateCount.

After declaring the variables, this code declares a cursor named Invoices_Cursor. Within this declaration, this code uses a SELECT statement to define the result set for this cursor. This result set contains two columns from the Invoices table and all of the rows that have a balance due.

Next, this code uses the OPEN statement to open the cursor. Then, it uses a FETCH statement to get the column values from the first row and store them in the variables declared earlier in the script.

After getting the values from the first row, this script uses a WHILE loop to loop through each row in the cursor. To do that, this WHILE loop checks the value of the @@FETCH_STATUS system function at the top of the loop. If this function returns a value that is not equal to -1, the loop continues. Otherwise, the end of the result set has been reached, so the loop exits. This loop works correctly because a second FETCH statement is coded at the end of the loop.

Within the loop, an IF statement checks whether the value of the InvoiceTotal column for the current row is greater than 1000. If it is, an UPDATE statement adds 10% of the InvoiceTotal column to the CreditTotal column for the row, and a SET statement increments the count of the number of rows that have been updated.

After the WHILE loop, this code closes and deallocates the cursor. Then, the first PRINT statement prints a blank line, and the second PRINT statements prints the number of rows that have been updated.

Before you use a cursor to work with individual rows in a result set, you should consider other solutions. That's because standard database access is faster and uses fewer server resources than cursor-based access. For example, you can accomplish the same update as the stored procedure in this figure with this UPDATE statement:

```
UPDATE Invoices
SET CreditTotal = CreditTotal + (InvoiceTotal * .1)
WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0
AND InvoiceTotal > 1000
```

However, if you encounter a situation where it makes sense to use a cursor, the skills presented in this figure should help you do that.

The syntax

Declare a cursor

```
DECLARE cursor_name CURSOR FOR select_statement;
```

Open the cursor

```
OPEN cursor_name;
```

Get column values from the row and store them in a series of variables

```
FETCH NEXT FROM cursor_name INTO @variable1[, @variable2[, @variable3]...];
```

Close and deallocate the cursor

```
CLOSE cursor_name;
```

```
DEALLOCATE cursor_name;
```

A script that uses a cursor

```
USE AP;

DECLARE @InvoiceIDVar int, @InvoiceTotalVar money, @UpdateCount int;
SET @UpdateCount = 0;

DECLARE Invoices_Cursor CURSOR
FOR
    SELECT InvoiceID, InvoiceTotal FROM Invoices
    WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0;

OPEN Invoices_Cursor;

FETCH NEXT FROM Invoices_Cursor INTO @InvoiceIDVar, @InvoiceTotalVar;
WHILE @@FETCH_STATUS <> -1
BEGIN
    IF @InvoiceTotalVar > 1000
    BEGIN
        UPDATE Invoices
        SET CreditTotal = CreditTotal + (InvoiceTotal * .1)
        WHERE InvoiceID = @InvoiceIDVar;

        SET @UpdateCount = @UpdateCount + 1;
    END;
    FETCH NEXT FROM Invoices_Cursor INTO @InvoiceIDVar, @InvoiceTotalVar;
END;

CLOSE Invoices_Cursor;
DEALLOCATE Invoices_Cursor;

PRINT '';
PRINT CONVERT(varchar, @UpdateCount) + ' row(s) updated.';
```

The response from the system when the script is run

2 row(s) updated.

Description

- The @@FETCH_STATUS system function returns 0 if the row was fetched successfully or -1 if the row can't be fetched because the end of the result set has been reached.

How to handle errors

To handle errors in a SQL Server script, you can use the TRY...CATCH statement shown in figure 14-11. Handling errors is often referred to as *error handling* or *exception handling*, and the TRY...CATCH statement works similarly to the exception handling statements that are available from the .NET languages such as C# and Visual Basic.

To start, you code the TRY block around any statements that might cause an error to be raised. A TRY block begins with the BEGIN TRY keywords and ends with the END TRY keywords. In this figure, for example, you can see that a TRY block is coded around an INSERT statement and a PRINT statement.

Immediately following the TRY block, you must code a single CATCH block. A CATCH block begins with the BEGIN CATCH keywords and ends with the END CATCH keywords. Within the CATCH block, you can include any statements that handle the error that might be raised in the TRY block. In this figure, for example, the first statement in the CATCH block uses a PRINT statement to display a simple message that indicates that the INSERT statement in the TRY block did not execute successfully. Then, the second PRINT statement uses two functions that are designed to work within a CATCH block to provide more detailed information about the error. All four of the functions you can use within a CATCH block are presented in this figure. Although it's common to use a CATCH block to display information to the user, you can also use a CATCH block to perform other error handling tasks such as writing information about the error to a log table or rolling back a transaction.

In this figure, the INSERT statement that's coded within the TRY block provides a vendor ID that doesn't exist. As a result, when SQL Server attempts to execute this statement, a foreign key constraint will be violated and an error will be raised. Then, program execution will skip over the PRINT statement that follows the INSERT statement and jump into the CATCH block. This causes the message that's shown in this figure to be displayed. However, if the INSERT statement had executed successfully, program execution would have continued by executing the PRINT statement immediately following the INSERT statement and skipping the CATCH block. In that case, this code would have displayed a message indicating that the INSERT statement executed successfully.

When coding TRY...CATCH statements, you may find that some types of errors aren't handled. In particular, errors with a low severity are considered warnings and aren't handled. Conversely, errors with a high severity often cause the database connection to be closed, which prevents them from being handled.

Another thing to keep in mind when coding TRY...CATCH statements is that they must be coded within a single batch, stored procedure, or trigger. In other words, you can't code a TRY block that spans multiple batches within a script. However, you can nest one TRY...CATCH statement within another. For example, if a CATCH block contains complex code that inserts error data into a log table, you may want to code a TRY...CATCH statement within that CATCH block to catch any errors that might occur there.

The syntax of the TRY...CATCH statement

```
BEGIN TRY
    {sql_statement|statement_block}
END TRY
BEGIN CATCH
    {sql_statement|statement_block}
END CATCH
```

Functions you can use within a CATCH block

Function	Description
<code>ERROR_NUMBER()</code>	Returns the error number.
<code>ERROR_MESSAGE()</code>	Returns the error message.
<code>ERROR_SEVERITY()</code>	Returns the severity of the error.
<code>ERROR_STATE()</code>	Returns the state of the error.

A script that uses a TRY...CATCH statement

```
BEGIN TRY
    INSERT Invoices
    VALUES (799, 'ZXX-799', '2020-03-07', 299.95, 0, 0,
            1, '2020-04-06', NULL);
    PRINT 'SUCCESS: Record was inserted.';
END TRY
BEGIN CATCH
    PRINT 'FAILURE: Record was not inserted.';
    PRINT 'Error ' + CONVERT(varchar, ERROR_NUMBER(), 1)
        + ':' + ERROR_MESSAGE();
END CATCH;
```

The message that's displayed

```
FAILURE: Record was not inserted.
Error 547: The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Invoices_Vendors". The conflict occurred in database "AP", table
"dbo.Vendors", column 'VendorID'.
```

Description

- You can use the TRY...CATCH statement to provide *error handling* (also known as *exception handling*). This works similarly to exception handling statements provided by C# and Visual Basic.
- A TRY block must be followed immediately by a single CATCH block.
- When an error occurs in a statement within a TRY block, control is passed to the CATCH block where the error can be processed. If no error occurs inside the TRY block, the CATCH block is skipped.
- Errors that have a severity of 10 or lower are considered warnings and are not handled by TRY...CATCH blocks. Errors that have a severity of 20 or higher and cause the database connection to be closed are not handled by TRY...CATCH blocks.
- Within a CATCH block, you can use the functions shown in this figure to return data about the error that caused the CATCH block to be executed.

How to use surround-with snippets

In chapter 11, you learned how to use snippets to help you code statements for creating database objects. In addition to these snippets, you can use *surround-with snippets* to enclose a block of statements in a BEGIN...END, IF, or WHILE statement. Figure 14-12 shows how surround-with snippets work.

The first screen in this figure shows part of the code from the script in figure 14-9. If you look back at that figure, you'll see that the UPDATE and IF...ELSE statements were executed within a WHILE loop. In this figure, I used a surround-with snippet to add the WHILE statement. To do that, I selected the statements I wanted to include in the loop and then selected the snippet for the WHILE statement as described in this figure.

When you insert a snippet for an IF or WHILE statement, a BEGIN...END statement is added automatically. That's true regardless of the number of statements you selected. That makes it easy to add statements to the block later on.

After you insert a snippet for an IF or WHILE statement, you have to enter a condition for the statement. To make that easy to do, the placeholder for the condition is highlighted. Then, you can just replace the placeholder with the appropriate condition.

The list of surround-with snippets

A screenshot of the SQL Server Management Studio (SSMS) interface. A context menu is open over a block of T-SQL code. The menu title is 'Surround With: |'. Below the title, there are three options: 'Begin', 'If', and 'While'. The 'While' option is highlighted with a blue border. A tooltip for the 'While' option states: 'Code Snippet for While loop.' The code itself is as follows:

```

SELECT * INTO #InvoiceCopy FROM Invoices
WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0;

UPDATE #InvoiceCopy
SET CreditTotal = CreditTotal + .05
WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0;

IF (SELECT MAX(CreditTotal) FROM #InvoiceCopy) > 3000
    BREAK;
ELSE --(SELECT MAX(CreditTotal) FROM #InvoiceCopy) <= 3000
    CONTINUE; Surround With: |

```

The code after the snippet is inserted

A screenshot of the SSMS interface showing the same T-SQL code after the 'While' snippet has been inserted. The code now includes a BEGIN...END block and a WHILE loop structure:

```

SELECT * INTO #InvoiceCopy FROM Invoices
WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0;

WHILE (Condition)
BEGIN

    UPDATE #InvoiceCopy
    SET CreditTotal = CreditTotal + .05
    WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0;

    IF (SELECT MAX(CreditTotal) FROM #InvoiceCopy) > 3000
        BREAK;
    ELSE --(SELECT MAX(CreditTotal) FROM #InvoiceCopy) <= 3000
        CONTINUE;

END

```

Description

- *Surround-with snippets* make it easy to enclose a block of statements in a BEGIN...END, IF, or WHILE statement.
- To insert a surround-with snippet, select the statements you want to enclose. Then, right-click on the statements, select the Surround With command from the resulting menu, and select the snippet you want to insert from the list that's displayed.
- To select a snippet from the list, double-click on it. Alternatively, you can use the Up and Down arrow keys to select the snippet and then press the Tab or Enter key.
- If you insert a snippet for an IF or WHILE statement, you will need to complete the statement by replacing the highlighted condition. You can also enter an ELSE clause for an IF statement.
- When you insert a snippet for an IF or WHILE statement, a BEGIN...END statement is added automatically.

Figure 14-12 How to use surround-with snippets

Advanced scripting techniques

The remaining topics of this chapter present some additional techniques you can use in the scripts you write. Here, you'll learn how to use some of the system functions that come with SQL Server, change some of the settings for the current session, use dynamic SQL, and use a command line utility to execute SQL statements and scripts. In addition, you'll see a complete script that uses many of the techniques presented in this chapter.

How to use the system functions

Figure 14-13 presents some of the Transact-SQL *system functions*. These functions are particularly helpful for writing Transact-SQL scripts. For example, the script shown in this figure illustrates how you might use the @@IDENTITY and @@ROWCOUNT functions. This script starts by inserting a row into the Vendors table. Because the VendorID column in that table is defined as an identity column, SQL Server generates the value of this column automatically. Then, the script uses the @@IDENTITY function to retrieve this value so it can insert an invoice for the new vendor. Before it does that, though, it uses the @@ROWCOUNT function to determine if the vendor was inserted successfully.

Notice that this script stores the values returned by the @@IDENTITY and @@ROWCOUNT functions in variables named @MyIdentity and @MyRowCount. Alternatively, the script could have used the system functions directly in the IF and VALUES clauses. However, the values returned by these functions can change each time a SQL statement is executed on the system, so it usually makes sense to store these values in variables immediately after you execute a SQL statement.

You can use the @@ERROR function to get the error number returned by the most recent SQL statement. If you use the TRY...CATCH statement as shown in figure 14-11, you won't need to use this function. However, you may see it used in scripts that were written with earlier versions of SQL Server.

The other functions I want to point out right now are the @@SERVERNAME, HOST_NAME, and SYSTEM_USER functions. The values returned by these functions can vary depending on who enters them and where they're entered. Because of that, they're often used to identify who entered or modified a row. For example, you could define a table with a column that defaults to the SYSTEM_USER function like this:

```
CREATE TABLE #SysFunctionEx  
  (EntryDBUser varchar(128) DEFAULT SYSTEM_USER);
```

This would cause the user name to be inserted automatically when each new row was added to the table.

At this point, you may be wondering why the names of some of the system functions start with two at signs (@@) and some don't. Those with @@ in their names have been a part of the T-SQL dialect for a long time and used to be called *global variables*. The other system functions have been added to T-SQL more recently.

Some of the Transact-SQL system functions

Function name	Description
<code>@@IDENTITY</code>	Returns the last value generated for an identity column on the server. Returns NULL if no identity value was generated.
<code>IDENT_CURRENT('tablename')</code>	Similar to @@IDENTITY, but returns the last identity value that was generated for a specified table.
<code>@@ROWCOUNT</code>	Returns the number of rows affected by the most recent SQL statement.
<code>@@ERROR</code>	Returns the error number generated by the execution of the most recent SQL statement. Returns 0 if no error occurred.
<code>@@SERVERNAME</code>	Returns the name of the local server.
<code>HOST_NAME()</code>	Returns the name of the current workstation.
<code>SYSTEM_USER</code>	Returns the name of the current user.

A script that inserts a new vendor and a new invoice

```
USE AP;
DECLARE @MyIdentity int, @MyRowCount int;

INSERT Vendors (VendorName, VendorAddress1, VendorCity, VendorState,
    VendorZipCode, VendorPhone, DefaultTermsID, DefaultAccountNo)
VALUES ('Peerless Binding', '1112 S Windsor St', 'Hallowell', 'ME',
    '04347', '(207) 555-1555', 4, 400);

SET @MyIdentity = @@IDENTITY;
SET @MyRowCount = @@ROWCOUNT;

IF @MyRowCount = 1
    INSERT Invoices
    VALUES (@MyIdentity, 'BA-0199', '2020-03-01', 4598.23,
        0, 0, 4, '2020-04-30', NULL);
```

The response from the system

(1 row(s) affected)

(1 row(s) affected)

Description

- The *system functions* return information about SQL Server values, objects, and settings. They can be used anywhere an expression is allowed.
- System functions are useful in writing scripts. In addition, some of these functions can be used to provide a value for a DEFAULT constraint on a column.
- System functions used to be called *global variables*, but that name is no longer used.
- In general, it's better to store the value returned by a system function in a variable than to use the system function directly. That's because the value of a system function can change when subsequent statements are executed.

Figure 14-13 How to use the system functions

How to change the session settings

Each time you start a new session, SQL Server sets the settings for that session to the defaults. If that's not what you want, you can change the settings using the SET statements presented in figure 14-14. Although SQL Server provides a variety of other statements, these are the ones you're most likely to use. And you're likely to use these only under special circumstances.

For example, because the default format for entering dates is "mdy," 03/06/20 is interpreted as March 6, 2020. If this date is being inserted from another data source, however, that data source could have used a date format where the year is entered first, followed by the month and the day. In that case, you could use a SET statement like the one shown in this figure to change the date format of the current session to "ymd." Then, the date would be interpreted as June 20, 2003.

The ANSI_NULLS option determines how null values are compared. By default, this option is set to ON, in which case you can't compare a value to the NULL keyword using a comparison operator. In that case,

`PaymentDate = NULL`

is always Unknown rather than True or False, even if PaymentDate contains a null value. To determine if a column contains a null value, you must use the IS NULL or IS NOT NULL clause. If you set the ANSI_NULLS option to OFF, however, the expression shown above would return True if PaymentDate contains a null value, and it would return False otherwise. Because a future version of SQL Server will require that the ANSI_NULLS option is always set to on, I recommend that you don't set this option to OFF.

The SET ROWCOUNT statement limits the number of rows that are processed by subsequent queries. For a SELECT query, this works the same as coding a TOP clause. However, since most other dialects of SQL don't support the TOP clause, you'll often see SET ROWCOUNT used in the code of other SQL programmers. Be aware, though, that this session setting affects all queries, including action queries and queries stored within views and stored procedures. Since this can cause unexpected results, I recommend that you avoid modifying this session setting and use the TOP clause instead.

Note that with a future version of SQL Server, SET ROWCOUNT won't affect actions queries. Because of that, you'll have to use the TOP clause to limit the rows processed by INSERT, UPDATE, and DELETE statements. Since you don't typically use the TOP clause with these statements, though, we don't present it in this book. If you want to learn more about using it with these statements, you can refer to the SQL Server documentation.

Transact-SQL statements for changing session settings

Statement	Description
<code>SET DATEFORMAT format</code>	Sets the order of the parts of a date (month/day/year) for entering date/time data. The default is mdy, but any permutation of m, d, and y is valid.
<code>SET NOCOUNT {ON OFF}</code>	Determines whether SQL Server returns a message indicating the number of rows that were affected by a statement. OFF is the default.
<code>SET ANSI_NULLS {ON OFF}</code>	Determines how SQL Server handles equals (=) and not equals (<>) comparisons with null values. The default is ON, in which case “WHERE column = NULL” will always return an empty result set, even if there are null values in the column.
<code>SET ANSI_PADDING {ON OFF}</code>	Determines how SQL Server stores char and varchar values that are smaller than the maximum size for a column or that contain trailing blanks. Only affects new column definitions. The default is ON, which causes char values to be padded with blanks. In addition, trailing blanks in varchar values are not trimmed. If this option is set to OFF, char values that don’t allow nulls are padded with blanks, but blanks are trimmed from char values that allow nulls as well as from varchar values.
<code>SET ROWCOUNT number</code>	Limits the number of rows that are processed by a query. The default setting is 0, which causes all rows to be processed.

A statement that changes the date format

```
SET DATEFORMAT ymd;
```

Description

- You use the SET statement to change configuration settings for the current session. These settings control the way queries and scripts execute.
- If the ANSI_NULLS option is set to ON, you can only test for null values in a column by using the IS NULL clause. See figure 3-15 in chapter 3 for details.
- In a future version of SQL Server, the ANSI_NULLS and ANSI_PADDING options will always be on and you won’t be able to turn them off. Because of that, you shouldn’t use these options in new scripts that you write.
- Instead of using the SET ROWCOUNT statement to limit the numbers of rows that are processed by a query, you should use the TOP clause. See chapter 3 for information on how to use this clause with the SELECT statement. For information on how to use it with the INSERT, UPDATE, and DELETE statements, see the SQL Server documentation.
- In a future version of SQL Server, the SET ROWCOUNT statement won’t affect INSERT, UPDATE, and DELETE statements.
- For a complete list of the Transact-SQL statements for changing session settings, see the topic on the SET statement in the SQL Server documentation.

Figure 14-14 How to change the session settings

How to use dynamic SQL

So far, the scripts you've seen in this chapter have contained predefined SQL statements. In other words, the statements don't change from one execution of the script to another other than for the values of variables used in the statements. However, you can also define an SQL statement as a script executes. Then, you use the EXEC statement shown in figure 14-15 to execute the *dynamic SQL*. Notice that EXEC is an abbreviation for EXECUTE.

The EXEC statement executes a string that contains a SQL statement. To illustrate, the first script in this figure executes a SELECT statement against a table that's specified at run time. To do that, it concatenates the literal string "SELECT * FROM " with the value of a variable named @TableNameVar. If you think about it, you'll realize that there's no other way to do this in SQL. Of course, it would have been easier to submit the simple query shown here directly rather than to use dynamic SQL. However, a more complex script might use an IF...ELSE statement to determine the table that's used in the query. In that case, dynamic SQL can make the script easier to code.

The second script in this figure is more complicated. It creates a table with columns that represent each vendor with outstanding invoices. That means that the number of columns in the new table will vary depending on the current values in the Invoices table.

This script starts by creating a variable named @DynamicSQL that will store the SQL string that's executed. Next, the new table to be created is deleted if it already exists. Then, a SET statement assigns the beginning of the SQL string to @DynamicSQL, which includes the CREATE TABLE statement, the name of the new table, and an opening parenthesis. The SELECT statement that follows concatenates the name of each column (the name of the current vendor) and data type to the SQL string. Finally, the second SET statement concatenates a closing parenthesis and a semicolon to the string, and the string is executed.

Notice that for each row retrieved by the SELECT statement, the variable @DynamicSQL is concatenated with its previous value. Although this syntax is valid for any query, it isn't useful except when generating dynamic SQL as shown here. Also notice that the name of each vendor is enclosed in brackets. That's because many of the vendors have spaces or other special characters in their names, which aren't allowed in column names unless they're delimited.

This figure also shows the SQL statement that's created by one execution of this script along with the contents of the table that's created. Although this table isn't useful the way it is, it could be used to cross-tabulate data based on the Vendors table. For example, each row of this table could represent a date and the Boolean value in each column could represent whether the vendor has an invoice that's due on that date. Since more than one vendor's invoice can be due on the same date, a cross-tabulation is a good representation of this data.

As you may have noticed, the SQL string that's generated by this script has an extra comma following the last column specification. Fortunately, the CREATE TABLE statement ignores this extra comma without generating an error. However, if you wanted to, you could eliminate this comma by using the LEFT function before concatenating the closing parenthesis.

The syntax of the EXEC statement

```
{EXEC|EXECUTE} ('SQL_string')
```

A script that uses an EXEC statement

```
USE AP;
DECLARE @TableNameVar varchar(128);
SET @TableNameVar = 'Invoices';
EXEC ('SELECT * FROM ' + @TableNameVar + ');';
```

The contents of the SQL string at execution

```
SELECT * FROM Invoices;
```

A script that creates a table with one column for each vendor with a balance due

```
USE AP;
DECLARE @DynamicSQL varchar(8000);

IF OBJECT_ID('XtabVendors') IS NOT NULL
    DROP TABLE XtabVendors;

SET @DynamicSQL = 'CREATE TABLE XtabVendors (
    SELECT @DynamicSQL = @DynamicSQL + '[' + VendorName + '] bit,
    FROM Vendors
    WHERE VendorID IN
        (SELECT VendorID
        FROM Invoices
        WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0)
    ORDER BY VendorName;
SET @DynamicSQL = @DynamicSQL + ')';

EXEC (@DynamicSQL);

SELECT * FROM XtabVendors;
```

The contents of the SQL string

```
CREATE TABLE XtabVendors ([Blue Cross] bit,[Cardinal Business Media, Inc.] bit,[Data Reproductions Corp] bit,[Federal Express Corporation] bit,[Ford Motor Credit Company] bit,[Ingram] bit,[Malloy Lithographing Inc] bit,);
```

The result set

Blue Cross	Cardinal Business Media, Inc.	Data Reproductions Corp	Federal Express Corporation	Ford M
<				>

Description

- The EXEC statement executes the SQL statement contained in a string. Because you define the SQL string within the script, you can create and execute SQL code that changes each time the script is run. This is called *dynamic SQL*.
- You can use dynamic SQL to perform operations that can't be accomplished using any other technique.

Figure 14-15 How to use dynamic SQL

A script that summarizes the structure of a database

Figure 14-16 presents a script that you can use to summarize the structure of a database. This script illustrates many of the techniques you learned in this chapter. It also shows how you might use some of the catalog views you learned about in the last chapter.

This script starts by dropping the temporary table named #TableSummary if it already exists. Then, it recreates this table using a SELECT INTO statement and data from three catalog views named tables, columns, and types. The tables view provides information about the tables in the current database. The columns view provides information about the columns in the current database, including the ID of the table that contains the column and the ID of the column's data type. And the types view contains information about data types.

As you can see in this figure, the tables and columns catalog views are joined to get the name of the table that contains a column. In addition, the columns and types catalog views are joined together to get the name of a column's data type. Then, the WHERE clause excludes three tables by name, including the two temporary tables created by this script and the system table named dtproperties.

Next, this script drops and recreates another temporary table named #AllUserTables. This table will be used to generate the row count for each table. It has two columns: an identity column and a column for the table name. The INSERT statement that follows populates this table with the same list of table names that was inserted into the #TableSummary table.

A script that creates a summary of the tables in a database **Page 1**

```
/*
Creates and queries a table, #TableSummary, that lists
the columns for each user table in the database, plus
the number of rows in each table.

Author: Bryan Syverson
Created: 2008-07-02
Modified: 2016-07-16
*/

USE AP;

IF OBJECT_ID('tempdb..#TableSummary') IS NOT NULL
    DROP TABLE #TableSummary;

SELECT sys.tables.name AS TableName, sys.columns.name AS ColumnName,
       sys.types.name AS Type
INTO #TableSummary
FROM sys.tables
    JOIN sys.columns ON sys.tables.object_id = sys.columns.object_id
    JOIN sys.types ON sys.columns.system_type_id = sys.types.system_type_id
WHERE sys.tables.name IN
    (SELECT name
     FROM sys.tables
     WHERE name NOT IN ('dtproperties', 'TableSummary', 'AllUserTables'));

IF OBJECT_ID('tempdb..#AllUserTables') IS NOT NULL
    DROP TABLE #AllUserTables;

CREATE TABLE #AllUserTables
(TableID int IDENTITY, TableName varchar(128));
GO

INSERT #AllUserTables (TableName)
SELECT name
FROM sys.tables
WHERE name NOT IN ('dtproperties', 'TableSummary', 'AllUserTables');
```

Description

- A SELECT INTO statement is used to retrieve information from the tables, columns, and types catalog views and store it in a temporary table named #TableSummary. This table has one row for each column in each table of the database that includes the table name, column name, and data type.
- A CREATE TABLE statement is used to create a temporary table named #AllUserTables. Then, an INSERT statement is used to insert rows into this table that contain the name of each table in the database. This information is retrieved from the catalog view named tables. Each row also contains a sequence number that's generated by SQL Server.
- The system table named dtproperties and the two temporary tables themselves are omitted from both SELECT queries.

Figure 14-16 A script that summarizes the structure of a database (part 1 of 2)

Part 2 of this script includes a WHILE loop that uses dynamic SQL to insert an additional row into #TableSummary for each table in #AllUserTables. Each of these rows indicates the total number of rows in one of the base tables. The @LoopMax variable used by this loop is set to the maximum value of the TableID column in #AllUserTables. The @LoopVar variable is set to 1, which is the minimum value of TableID. The WHILE loop uses @LoopVar to step through the rows of #AllUserTables.

Within the loop, the SELECT statement sets @TableNameVar to the value of the TableName column for the current table. Then, @ExecVar is built by concatenating each of the clauses of the final SQL string. This string consists of three statements. The DECLARE statement is used to create a variable named @CountVar that will store the number of rows in the current table. Note that because this variable is created within the dynamic SQL statement, its scope is limited to the EXEC statement. In other words, it isn't available to the portion of the script outside of the EXEC statement.

The SELECT statement within the dynamic SQL statement retrieves the row count from the current table and stores it in the @CountVar variable. Then, the INSERT statement inserts a row into the #TableSummary table that includes the table name, a literal value that indicates that the row contains the row count, and the number of rows in the table. You can see the contents of the SQL string that's created for one table, the ContactUpdates table, in this figure.

After the SQL string is created, it's executed using an EXEC statement. Then, @LoopVar is increased by 1 and the loop is executed again. When the loop completes, the script executes a SELECT statement that retrieves the data from the #TableSummary table. That result set is also shown in this figure.

A script that creates a summary of the tables in a database**Page 2**

```

DECLARE @LoopMax int, @LoopVar int;
DECLARE @TableNameVar varchar(128), @ExecVar varchar(1000);

SELECT @LoopMax = MAX(TableID) FROM #AllUserTables;

SET @LoopVar = 1;

WHILE @LoopVar <= @LoopMax
BEGIN
    SELECT @TableNameVar = TableName
    FROM #AllUserTables
    WHERE TableID = @LoopVar;
    SET @ExecVar = 'DECLARE @CountVar int; ';
    SET @ExecVar = @ExecVar + 'SELECT @CountVar = COUNT(*) ';
    SET @ExecVar = @ExecVar + 'FROM ' + @TableNameVar + '; ';
    SET @ExecVar = @ExecVar + 'INSERT #TableSummary ';
    SET @ExecVar = @ExecVar + 'VALUES ('' + @TableNameVar + ''',';
    SET @ExecVar = @ExecVar + '''*Row Count*'','');
    SET @ExecVar = @ExecVar + ' @CountVar);';
    EXEC (@ExecVar);
    SET @LoopVar = @LoopVar + 1;
END;

SELECT * FROM #TableSummary
ORDER BY TableName, ColumnName;

```

The contents of the SQL string for one iteration of the loop

```

DECLARE @CountVar int; SELECT @CountVar = COUNT(*) FROM ContactUpdates;
INSERT #TableSummary VALUES ('ContactUpdates','*Row Count*', @CountVar);

```

The result set

	TableName	ColumnName	Type
30	InvoiceLineItems	*Row Count*	118
31	InvoiceLineItems	AccountNo	int
32	InvoiceLineItems	InvoiceID	int
33	InvoiceLineItems	InvoiceLineItemA...	money
34	InvoiceLineItems	InvoiceLineItemD...	varchar
35	InvoiceLineItems	InvoiceSequence	smallint
36	Invoices	*Row Count*	114
37	Invoices	CreditTotal	money

Description

- The WHILE statement loops through the tables in the #AllUserTables table. For each table, it creates a dynamic SQL string that contains a SELECT statement and an INSERT statement. The SELECT statement retrieves the number of rows in the table, and the INSERT statement inserts a new row into the #TableSummary table that indicates the number of rows.
- The final SELECT statement retrieves all of the rows and columns from the #TableSummary table sorted by column name within table name.

Figure 14-16 A script that summarizes the structure of a database (part 2 of 2)

How to use the SQLCMD utility

SQL Server comes with a command line utility known as the *SQLCMD utility*. Unlike the Management Studio you've used throughout this book, the SQLCMD utility lets you enter and execute scripts from a command line. One advantage of the SQLCMD utility is that it provides a way to run a SQL script from a DOS batch file.

Figure 14-17 presents an example of a Command Prompt window running a SQLCMD session. To open a session, you enter "sqlcmd" at the command prompt, followed by the appropriate command line switches. To start a session, you must begin by using the -S switch to specify a valid server. Then, if you want to connect to SQL Server using Windows authentication, the only command line switch you need is -E as shown in this figure. If you connect using SQL Server authentication, though, you'll need to enter switches for the user name and password like this:

```
sqlcmd -S localhost\SQLExpress -U joel -P Top$Secret
```

You can also omit the password switch and the SQLCMD utility will prompt you for your password. This improves security because, unlike the Command Prompt window, the SQLCMD utility doesn't display the password on the screen.

Once you're connected, you can type one SQL statement per line as shown. Then, to execute the statement you've entered, you enter a GO command. When you're done, you can close the SQLCMD session by entering the EXIT command. Then, you're returned to the command prompt.

You can also execute a script that's stored in a file on disk. To do that, you use the -i switch. To save the response from the server to a file, you use the -o switch. For example, this command would execute the script contained in a file named test.sql and save the result set in a file named test.txt:

```
sqlcmd -S localhost\SQLExpress -i test.sql -o test.txt
```

Note that the response also includes any result sets that are created. As a result, if the script stored in the test.sql file contains a SELECT statement, the result set returned by that SELECT statement will be stored in text format in the test.txt file.

Although the SQLCMD utility provides an easy way to run T-SQL scripts from a DOS command line, SQL Server also provides support for Microsoft's command line tool, *Microsoft Windows PowerShell*. PowerShell is a powerful scripting tool that makes it possible to automate complex administrative tasks across multiple servers. However, due to its power and complexity, PowerShell has a steep learning curve. As a result, if you aren't familiar with it already, you'll only want to use it if you can't accomplish the task using a DOS batch file and a T-SQL script.

A Command Prompt window running the SQLCMD utility

```

C:\ Command Prompt - sqlcmd -S localhost\SQLExpress -E - SQLCMD
Microsoft Windows [Version 10.0.17763.503]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Anne>sqlcmd -S localhost\SQLExpress -E
1> use ap
2> go
Changed database context to 'AP'.
1> select invoicedate, invoicetotal from invoices where vendorid = 122
2> go
invoicedate      invoicetotal
-----
2019-10-08        3813.3300
2019-12-01        2765.3600
2019-12-08        2184.1100
2019-12-12        2312.2000
2019-12-16        2115.8100
2019-12-20        1927.5400
2020-01-01        2318.0300
2020-01-23        2051.5900
2020-01-24        3689.9900

(9 rows affected)
1>

```

Command line switches

Switch	Function
-?	Show a summary of all command line switches.
-E	Use a trusted connection (Windows authentication mode).
-L	List the names of the available servers.
-S server_name	Log in to a specific server.
-U user_name	Log in as a specific user (SQL Server authentication mode).
-P password	Specify the password in the command line (SQL Server authentication mode).
-Q "query"	Execute the specified query, then exit.
-i file_name	Specify the name of the script file to be executed.
-o file_name	Specify an output file in which to save responses from the system.

Description

- You can use the *SQLCMD utility* to run T-SQL scripts from a command line. This provides a way to use a DOS batch file to run a script.
- To open a Command Prompt window, select Command Prompt from the Start menu. On Windows 10, it's in the Windows System group.
- To start the SQLCMD utility, enter “sqlcmd” at the C:\> prompt along with the appropriate command line switches.
- You must begin most commands with the -S switch to specify the name of a valid server.
- To log in, you can use the -E switch for Windows authentication, or you can use the -U and -P switches for SQL Server authentication.
- Once you've started the SQLCMD utility and logged in, you can enter the statements you want to execute followed by the GO command.
- To exit from the SQLCMD utility, enter “exit” at the SQLCMD prompt.

Figure 14-17 How to use the SQLCMD utility

Perspective

In this chapter, you've learned how to code procedural scripts in T-SQL. By using the techniques you've learned here, you'll be able to code scripts that are more general, more useful, and less susceptible to failure. In particular, when you use dynamic SQL, you'll be able to solve problems that can't be solved using any other technique.

In the next chapter, you'll expand on what you've learned here by learning how to code stored procedures, functions, and triggers. These objects are basically one-batch scripts that are stored with the database. But they provide special functionality that gives you greater control over a database, who has access to it, and how they can modify it.

Terms

script	nested IF...ELSE statements
batch	WHILE loop
T-SQL statement	cursor
variable	error handling
scalar variable	exception handling
local variable	surround-with snippets
table variable	system function
temporary table	global variable
local temporary table	dynamic SQL
global temporary table	SQLCMD
scope	Windows PowerShell

Exercises

1. Write a script that declares and sets a variable that's equal to the total outstanding balance due. If that balance due is greater than \$10,000.00, the script should return a result set consisting of VendorName, InvoiceNumber, InvoiceDueDate, and Balance for each invoice with a balance due, sorted with the oldest due date first. If the total outstanding balance due is less than \$10,000.00, the script should return the message "Balance due is less than \$10,000.00."
2. The following script uses a derived table to return the date and invoice total of the earliest invoice issued by each vendor. Write a script that generates the same result set but uses a temporary table in place of the derived table. Make sure your script tests for the existence of any objects it creates.

```
USE AP;
```

```
SELECT VendorName, FirstInvoiceDate, InvoiceTotal
FROM Invoices JOIN
    (SELECT VendorID, MIN(InvoiceDate) AS FirstInvoiceDate
     FROM Invoices
     GROUP BY VendorID) AS FirstInvoice
    ON (Invoices.VendorID = FirstInvoice.VendorID AND
        Invoices.InvoiceDate = FirstInvoice.FirstInvoiceDate)
JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
ORDER BY VendorName, FirstInvoiceDate;
```

3. Write a script that generates the same result set as the code shown in exercise 2, but uses a view instead of a derived table. Also write the script that creates the view. Make sure that your script tests for the existence of the view. The view doesn't need to be redefined each time the script is executed.
4. Write a script that uses dynamic SQL to return a single column that represents the number of rows in the first table in the current database. The script should automatically choose the table that appears first alphabetically, and it should exclude tables named dtproperties and sysdiagrams. Name the column CountOfTable, where Table is the chosen table name.

Hint: Use the sys.tables catalog view.

How to code stored procedures, functions, and triggers

Now that you've learned how to work with scripts, you know that procedural statements can help you manage a database and automate tasks. In this chapter, you'll learn how to extend this functionality by creating database objects that store program code within a database. The three types of programs discussed in this chapter provide a powerful and flexible way to control how a database is used.

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
Perspective	506

Procedural programming options in Transact-SQL

Figure 15-1 presents the four types of procedural programs you can code using Transact-SQL. Each program type contains SQL statements. However, they differ by how they’re stored and executed.

Scripts

Of the four types of procedural programs, only scripts can contain two or more batches. That’s because only scripts can be executed by SQL Server tools such as the Management Studio and the SQLCMD utility. In addition, only scripts are stored in files outside of the database. For these reasons, scripts tend to be used most often by SQL Server programmers and database administrators.

Stored procedures, user-defined functions, and triggers

The other three types of procedural programs—*stored procedures*, *user-defined functions*, and *triggers*—are executable database objects. This means that each is stored within the database. To create these objects, you use the DDL statements you’ll learn about in this chapter. Then, these objects remain as a part of the database until they’re explicitly dropped.

Stored procedures, user-defined functions, and triggers differ by how they’re executed. Stored procedures and user-defined functions can be run from any database connection that can run a SQL statement. By contrast, triggers run automatically in response to the execution of an action query on a specific table.

Stored procedures are frequently written by SQL programmers for use by end users or application programmers. If you code stored procedures in this way, you can simplify the way these users interact with a database. In addition, you can provide access to a database exclusively through stored procedures. This gives you tight control over the security of the data.

Both user-defined functions and triggers are used more often by SQL programmers than by application programmers or end users. SQL programmers often use their own functions within the scripts, stored procedures, and triggers they write. Since triggers run in response to an action query, programmers use them to help prevent errors caused by inconsistent or invalid data.

Stored procedures, functions, and triggers also differ by whether or not they can use parameters. *Parameters* are values that can be passed to or returned from a procedure. Both stored procedures and user-defined functions can use parameters, but triggers can’t.

A comparison of the different types of procedural SQL programs

Type	Batches	How it's stored	How it's executed	Accepts parameters
Script	Multiple	In a file on a disk	From within a client tool such as the Management Studio or SQLCMD	No
Stored procedure	One only	In an object in the database	By an application or within a SQL script	Yes
User-defined function	One only	In an object in the database	By an application or within a SQL script	Yes
Trigger	One only	In an object in the database	Automatically by the server when a specific action query is executed	No

Description

- You can write procedural programs with Transact-SQL using scripts, stored procedures, user-defined functions, and triggers.
- Scripts are useful for those users with access to the SQL Server client tools, such as the Management Studio. Typically, these tools are used by SQL programmers and DBAs, not by application programmers or end users.
- Stored procedures, user-defined functions, and triggers are all executable database objects that contain SQL statements. Although they differ in how they're executed and by the kinds of values they can return, they all provide greater control and better performance than a script.
- *Stored procedures* give the SQL programmer control over who accesses the database and how. Since some application programmers don't have the expertise to write certain types of complex SQL queries, stored procedures can simplify their use of the database.
- *User-defined functions* are most often used by SQL programmers within the stored procedures and triggers that they write, although they can also be used by application programmers and end users.
- *Triggers* are special procedures that execute when an action query, such as an INSERT, UPDATE, or DELETE statement, is executed. Like constraints, you can use triggers to prevent database errors, but triggers give you greater control and flexibility.
- Since procedures, functions, and triggers are database objects, the SQL statements you use to create, delete, and modify them are considered part of the DDL.

Figure 15-1 Procedural programming options in Transact-SQL

How to code stored procedures

A *stored procedure* is a database object that contains one or more SQL statements. In the topics that follow, you'll learn how to create and use stored procedures. In addition, you'll learn how to use some of the stored procedures provided by SQL Server.

An introduction to stored procedures

Figure 15-2 presents a script that creates a stored procedure, also called an *sproc* or just a *procedure*. To do that, you use the CREATE PROC statement. You'll learn the details of coding this statement in a moment.

The first time a procedure is executed, each SQL statement it contains is compiled and executed to create an *execution plan*. Then, the procedure is stored in compiled form within the database. For each subsequent execution, the SQL statements are executed without compilation, because they're *precompiled*. This makes the execution of a stored procedure faster than the execution of an equivalent SQL script.

To execute, or *call*, a stored procedure, you use the EXEC statement. If the EXEC statement is the first line in a batch, you can omit the EXEC keyword and just code the procedure name. Since this can lead to code that's confusing to read, however, I recommend that you include the EXEC keyword.

The script in this figure creates a stored procedure named spInvoiceReport. This procedure consists of a single statement: a SELECT statement that retrieves data from the Vendors and Invoices tables. As you'll see in the topics that follow, however, a stored procedure can contain more than one statement, along with the same procedural code used in scripts.

When you execute the script in this figure, you create the stored procedure. The response from the system shows that the procedure was created successfully. Then, when you execute the stored procedure, the result set retrieved by the SELECT statement is returned.

As you can see, a user or program that calls this procedure doesn't need to know the structure of the database to use the stored procedure. This simplifies the use of the database by eliminating the need to know SQL and the need to understand the structure of the database.

As you'll learn in chapter 17, you can allow a user or program to call specific stored procedures but not to execute other SQL statements. By doing this, you can secure your database by restricting access to only those rows, columns, and tables that you provide access to through the stored procedures. For those systems where security is critical, this can be the best way to secure the data.

A script that creates a stored procedure

```
USE AP;
GO
CREATE PROC spInvoiceReport
AS

SELECT VendorName, InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0
ORDER BY VendorName;
```

The response from the system

Commands completed successfully.

A statement that calls the procedure

```
EXEC spInvoiceReport;
```

The result set created by the procedure

	VendorName	InvoiceNumber	InvoiceDate	InvoiceTotal
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	963253264	2020-01-18	52.25
5	Federal Express Corporation	263253268	2020-01-21	59.97

Description

- A stored procedure is an executable database object that contains SQL statements. A stored procedure is also called a *sproc* (pronounced either as one word or as “ess-proc”) or a *procedure*.
- Stored procedures are *precompiled*. That means that the *execution plan* for the SQL code is compiled the first time the procedure is executed and is then saved in its compiled form. For this reason, stored procedures execute faster than an equivalent SQL script.
- You use the EXEC statement to run, or *call*, a procedure. If this statement is the first line in a batch, you can omit the EXEC keyword and code just the procedure name. To make your code easier to read, however, you should always include the EXEC keyword.
- You can call a stored procedure from within another stored procedure. You can even call a stored procedure from within itself. This technique, called a *recursive call* or *recursion*, is seldom used in SQL programming.
- One of the advantages of using procedures is that application programmers and end users don’t need to know the structure of the database or how to code SQL.
- Another advantage of using procedures is that they can restrict and control access to a database. If you use procedures in this way, you can prevent both accidental errors and malicious damage.

Figure 15-2 An introduction to stored procedures

How to create a stored procedure

Figure 15-3 presents the syntax of the CREATE PROC statement you use to create a stored procedure. You code the name of the procedure in the CREATE PROC clause. Note that stored procedure names can't be the same as the name of any other object in the database. To help distinguish a stored procedure from other database objects, it's a good practice to prefix its name with the letters *sp*.

When the CREATE PROC statement is executed, the syntax of the SQL statements within the procedure is checked. If you've made a coding error, the system responds with an appropriate message and the procedure isn't created.

Because the stored procedure is created in the current database, you need to change the database context by coding a USE statement before the CREATE PROC statement. In addition, CREATE PROC must be the first and only statement in the batch. Since the script in this figure creates the procedure after a USE and DROP PROC statement, for example, it has a GO command just before the CREATE PROC statement.

In addition to stored procedures that are stored in the current database, you can create *temporary stored procedures* that are stored in the tempdb database. These procedures exist only while the current database session is open, so they aren't used often. To identify a temporary stored procedure, prefix the name with one number sign (#) for a *local procedure* and two number signs (##) for a *global procedure*.

After the name of the procedure, you code declarations for any parameters it uses. You'll learn more about that in the figures that follow.

You can also code the optional WITH clause with the RECOMPILE option, the ENCRYPTION option, the EXECUTE_AS_clause option, or any combination of these options. The RECOMPILE option prevents the system from precompiling the procedure. That means that the execution plan for the procedure must be compiled each time it's executed, which will slow down most procedures. For this reason, you should generally omit this option.

Some procedures, however, might make use of unusual or atypical values. If so, the first compilation may result in an execution plan that isn't efficient for subsequent executions. In that case, the additional overhead involved in recompiling the procedure may be offset by the reduced query execution time. If you find that a stored procedure you've written performs erratically, you may want to try this option.

ENCRYPTION is a security option that prevents the user from being able to view the declaration of a stored procedure. Since the system stores the procedure as an object in the database, it also stores the code for the procedure. If this code contains information that you don't want the user to examine, you should use this option.

The EXECUTE_AS_clause option allows you to specify an EXECUTE AS clause to allow users to execute the stored procedure with a specified security context. For example, you can use this clause to allow users to execute the stored procedure with the same security permissions as you. That way, you can be sure that the stored procedure will work for the caller even if the caller doesn't have permissions to access all of the objects that you used within the stored procedure.

The syntax of the CREATE PROC statement

```
CREATE {PROC|PROCEDURE} procedure_name  
[parameter_declarations]  
[WITH [RECOMPILE] [, ENCRYPTION] [, EXECUTE_AS_clause]]  
AS sql_statements
```

A script that creates a stored procedure that copies a table

```
USE AP;  
IF OBJECT_ID('spCopyInvoices') IS NOT NULL  
    DROP PROC spCopyInvoices;  
GO  
  
CREATE PROC spCopyInvoices  
AS  
    IF OBJECT_ID('InvoiceCopy') IS NOT NULL  
        DROP TABLE InvoiceCopy;  
    SELECT *  
    INTO InvoiceCopy  
    FROM Invoices;
```

Description

- You use the CREATE PROC statement to create a stored procedure in the current database. The name of a stored procedure can be up to 128 characters and is typically prefixed with the letters *sp*.
- The CREATE PROC statement must be the first and only statement in a batch. If you're creating the procedure within a script, then, you must code a GO command following any statements that precede the CREATE PROC statement.
- To create a *temporary stored procedure*, prefix the procedure name with a number sign (#) for a *local procedure* or two number signs (##) for a *global procedure*. A temporary stored procedure only exists while the current database session is open.
- You can use *parameters* to pass one or more values from the calling program to the stored procedure or from the procedure to the calling program. See figures 15-4 and 15-5 for more information on working with parameters.
- The AS clause contains the SQL statements to be executed by the stored procedure. Since a stored procedure must consist of a single batch, a GO command is interpreted as the end of the CREATE PROC statement.
- The RECOMPILE option prevents the system from precompiling the procedure, which means that it has to be compiled each time it's run. Since that reduces system performance, you don't typically use this option.
- The ENCRYPTION option prevents users from viewing the code in a stored procedure. See figure 15-11 for more information on viewing stored procedures.
- The EXECUTE_AS_clause option allows users to execute the stored procedure with the permissions specified by the EXECUTE AS clause. For more information, look up "EXECUTE AS clause" in the SQL Server documentation.

Figure 15-3 How to create a stored procedure

How to declare and work with parameters

Figure 15-4 presents the syntax for declaring parameters in a CREATE PROC statement. Like a local variable, the name of a parameter must begin with an at sign (@). The data type for a parameter can be any valid SQL Server data type except for the table data type.

Stored procedures provide for two different types of parameters: input parameters and output parameters. An *input parameter* is passed to the stored procedure from the calling program. An *output parameter* is returned to the calling program from the stored procedure. You identify an output parameter with the OUTPUT keyword. If this keyword is omitted, the parameter is assumed to be an input parameter.

You can declare an input parameter so it requires a value or so its value is optional. The value of a *required parameter* must be passed to the stored procedure from the calling program or an error occurs. The value of an *optional parameter* doesn't need to be passed from the calling program. You identify an optional parameter by assigning a default value to it. Then, if a value isn't passed from the calling program, the default value is used. Although you can also code a default value for an output parameter, there's usually no reason for doing that.

You can also use output parameters as input parameters. That is, you can pass a value from the calling program to the stored procedure through an output parameter. However, that's an unusual way to use output parameters. To avoid confusion, you should use output parameters strictly for output.

Within the procedure, you use parameters like variables. Although you can change the value of an input parameter within the procedure, that change isn't returned to the calling program and has no effect on it. Instead, when the procedure ends, the values of any output parameters are returned to the calling program.

The syntax for declaring parameters

```
@parameter_name_1 data_type [= default] [OUTPUT]
[, @parameter_name_2 data_type [= default] [OUTPUT]]...
```

Typical parameter declarations

<pre>@DateVar date</pre>	-- Input parameter that accepts -- a date value
<pre>@VendorVar varchar(40) = NULL</pre>	-- Optional input parameter that accepts -- a character value
<pre>@InvTotal money OUTPUT</pre>	-- Output parameter that returns -- a monetary value

A CREATE PROC statement that uses an input and an output parameter

```
CREATE PROC spInvTotal1
    @DateVar date,
    @InvTotal money OUTPUT
AS
SELECT @InvTotal = SUM(InvoiceTotal)
FROM Invoices
WHERE InvoiceDate >= @DateVar;
```

A CREATE PROC statement that uses an optional parameter

```
CREATE PROC spInvTotal2
    @DateVar date = NULL
AS
IF @DateVar IS NULL
    SELECT @DateVar = MIN(InvoiceDate) FROM Invoices;
SELECT SUM(InvoiceTotal)
FROM Invoices
WHERE InvoiceDate >= @DateVar;
```

Description

- To declare a parameter within a stored procedure, you code the name of the parameter followed by its data type. The parameter name must start with an at sign (@), and the data type can be any type except table. Parameters are always local to the procedure.
- Input parameters* accept values passed from the calling program.
- Output parameters* store values that are passed back to the calling program. You identify an output parameter by coding the OUTPUT keyword after the parameter name and data type
- Optional parameters* are parameters that do not require that a value be passed from the calling program. To declare an optional parameter, you assign it a default value. Then, that value is used if one isn't passed from the calling program.
- A stored procedure can declare up to 2100 parameters. If you declare two or more parameters, the declarations must be separated by commas.
- It's a good programming practice to code your CREATE PROC statements so they list required parameters first, followed by optional parameters.

Figure 15-4 How to declare and work with parameters

How to call procedures with parameters

Figure 15-5 shows how you call procedures that use parameters. The stored procedure in this figure accepts two input parameters and returns one output parameter. As you can see, both of the input parameters are optional because each has a default value.

To pass parameter values to a stored procedure, you code the values in the EXEC statement after the procedure name. You can pass parameters to a stored procedure either by position or by name. The first EXEC statement in this figure passes the parameters *by position*. When you use this technique, you don't include the names of the parameters. Instead, the parameters are listed in the same order as they appear in the CREATE PROC statement. This is the most common way to call stored procedures that have a short list of parameters.

The second EXEC statement shows how you can pass the parameters *by name*. To do that, you include the names of the parameters as defined in the CREATE PROC statement. When you use this technique, you can list parameters in any order. If the procedure has many parameters, particularly if some of them are optional, passing parameters by name is usually easier than passing parameters by position.

The third EXEC statement in this figure shows how you can omit an optional parameter when you pass the parameters by name. To do that, you simply omit the optional parameter. By contrast, when you pass parameters by position, you can omit them only if they appear after the required parameters. This is illustrated by the last EXEC statement in this figure.

Notice that in all four of these examples, the EXEC statement is preceded by a DECLARE statement that creates a variable named @MyInvTotal. This variable is used to store the value of the output parameter that's returned from the stored procedure. As you can see, the name of this variable is included in each of the EXEC statements in this figure. In addition, the variable name is followed by the OUTPUT keyword, which identifies it as an output parameter.

A CREATE PROC statement that includes three parameters

```

CREATE PROC spInvTotal3
    @InvTotal money OUTPUT,
    @DateVar date = NULL,
    @VendorVar varchar(40) = '%'
AS

IF @DateVar IS NULL
    SELECT @DateVar = MIN(InvoiceDate) FROM Invoices;

SELECT @InvTotal = SUM(InvoiceTotal)
FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
WHERE (InvoiceDate >= @DateVar) AND
    (VendorName LIKE @VendorVar);

```

Code that passes the parameters by position

```

DECLARE @MyInvTotal money;
EXEC spInvTotal3 @MyInvTotal OUTPUT, '2020-01-01', 'P%';

```

Code that passes the parameters by name

```

DECLARE @MyInvTotal money;
EXEC spInvTotal3 @DateVar = '2020-01-01', @VendorVar = 'P%',
    @InvTotal = @MyInvTotal OUTPUT;

```

Code that omits one optional parameter

```

DECLARE @MyInvTotal money;
EXEC spInvTotal3 @VendorVar = 'M%', @InvTotal = @MyInvTotal OUTPUT;

```

Code that omits both optional parameters

```

DECLARE @MyInvTotal money;
EXEC spInvTotal3 @MyInvTotal OUTPUT;

```

Description

- To call a procedure that accepts parameters, you pass values to the procedure by coding them following the procedure name. You can pass the parameters by position or by name.
- To pass parameters *by position*, list them in the same order as they appear in the CREATE PROC statement and separate them with commas. When you use this technique, you can omit optional parameters only if they're declared after any required parameters.
- To pass parameters *by name*, code the name of the parameter followed by an equal sign and the value. You can separate multiple parameters with commas. When you use this technique, you can list the parameters in any order and you can easily omit optional parameters.
- To use an output parameter in the calling program, you must declare a variable to store its value. Then, you use the name of that variable in the EXEC statement, and you code the OUTPUT keyword after it to identify it as an output parameter.

Figure 15-5 How to call procedures with parameters

How to work with return values

In addition to passing output parameters back to the calling program, stored procedures also pass back a *return value*. By default, this value is zero. However, you can use a RETURN statement to return another number. For example, if a stored procedure updates rows, you may want to return the number of rows that have been updated. To do that, you can use the @@ROWCOUNT function described in chapter 14.

In figure 15-6, the stored procedure named spInvCount returns a count of the number of invoices that meet the conditions specified by the input parameters. These parameters are identical to the input parameters used by the stored procedure in figure 15-5. However, since this procedure uses a RETURN statement to return an integer value, there's no need to use an output parameter.

The script that calls the procedure uses a variable to store the return value. To do that, the name of the variable is coded after the EXEC keyword, followed by an equals sign and the name of the stored procedure. After the procedure returns control to the script, the script uses a PRINT statement to print the return value.

In this figure, the script gets the count of invoices where the invoice date is on or after January 1, 2020 and the vendor's name begins with P. Here, the return value indicates that 2 invoices match these specifications.

So, when should you use the RETURN statement to return values and when should you use output parameters? If a stored procedure needs to return a single integer value, many programmers prefer using a RETURN statement since the syntax for returning a value is more concise and intuitive than using output parameters. However, if a stored procedure needs to return other types of data, or if it needs to return multiple values, then a RETURN statement won't work. In that case, you can use output parameters, or you can use a function to return other data types (including result sets) as described later in this chapter. Of course, you can always use a RETURN statement together with output parameters whenever that makes sense.

The syntax of the RETURN statement for a stored procedure

```
RETURN [integer_expression]
```

A stored procedure that returns a value

```
CREATE PROC spInvCount
    @DateVar date = NULL,
    @VendorVar varchar(40) = '%'
AS

IF @DateVar IS NULL
    SELECT @DateVar = MIN(InvoiceDate) FROM Invoices;

DECLARE @InvCount int;

SELECT @InvCount = COUNT(InvoiceID)
FROM Invoices JOIN Vendors
    ON Invoices.VendorID = Vendors.VendorID
WHERE (InvoiceDate >= @DateVar) AND
    (VendorName LIKE @VendorVar);

RETURN @InvCount;
```

A script that calls the stored procedure

```
DECLARE @InvCount int;
EXEC @InvCount = spInvCount '2020-01-01', 'P%';
PRINT 'Invoice count: ' + CONVERT(varchar, @InvCount);
```

The response from the system

```
Invoice count: 2
```

Description

- The RETURN statement immediately exits the procedure and returns an optional integer value to the calling program. If you don't specify a value in this statement, the *return value* is zero.
- To use the return value in the calling program, you must declare a variable to store its value. Then, you code that variable name followed by an equals sign and the name of the procedure in the EXEC statement.

How to validate data and raise errors

In addition to using the TRY...CATCH statement to handle errors after they occur, you can also prevent errors before they occur by checking data before it's used to make sure it's valid. Checking data before it's used is often referred to as *data validation*, and it often makes sense to perform data validation within a stored procedure. Then, if the data is not valid, you can execute code that makes it valid, or you can return an error to the calling program. To return an error, it's often helpful to use the THROW statement. Then, if the calling program contains a TRY...CATCH statement, it can catch and handle the error. Otherwise, the client connection is terminated immediately.

Figure 15-7 presents the syntax of the THROW statement. The first parameter is the error number you want to assign to the error. The value of this parameter must be 50000 or greater, which identifies it as a custom error. You can use this value to indicate the type of error that occurred. The second parameter is simply the error message you want to display if the error is raised. And the third parameter is the state that you want to associate with the error. The state code is strictly informational and has no system meaning. You can use any value between 0 and 255 to represent the state that the system was in when the error was raised. In most cases, you'll just code 1 for this argument.

The stored procedure in this figure illustrates how the THROW statement works. This procedure checks the VendorID that's passed from the calling program before it performs the insert operation that's specified by the INSERT statement. That way, the system error that's raised when you try to insert a row with an invalid foreign key will never occur. Instead, if the VendorID value is invalid, the THROW statement will raise a custom error that provides a user-friendly message. In this case, the custom error contains a short message that indicates that the VendorID is not valid.

The calling script in this figure attempts to insert a row into the Invoices table with VendorID 799. Since this VendorID doesn't exist in the Vendors table, the insertion causes the custom error to be raised. As a result, program execution jumps into the CATCH block of the TRY...CATCH statement. This CATCH block prints a message that indicates that an error occurred, and it prints the message that's stored in the custom error. Then, the catch block uses an IF statement to check if the error number is greater than or equal to 50000. If so, it prints a message on the third line that indicates that the error is a custom error.

When you code a THROW statement within a block of statements, you should be aware that it must always be preceded by a semicolon. That's true even if the THROW statement is the first or only statement in the block. This is illustrated in the last example in this figure. Here, you can see that I've coded a semicolon on the line before the THROW statement. Of course, if the THROW statement is preceded by another statement in the block, you can just code the semicolon at the end of that statement.

The THROW statement was introduced with SQL Server 2012. In previous versions of SQL Server, you used the RAISERROR statement to perform a similar function. In addition to raising custom errors, this statement lets you raise system errors. It also lets you specify a severity level that indicates whether

The syntax of the THROW statement

```
THROW [error_number, message, state]
```

A stored procedure that tests for a valid foreign key

```
CREATE PROC spInsertInvoice
    @VendorID      int,   @InvoiceNumber  varchar(50),
    @InvoiceDate   date,   @InvoiceTotal   money,
    @TermsID       int,   @InvoiceDueDate date
AS
    IF EXISTS(SELECT * FROM Vendors WHERE VendorID = @VendorID)
        INSERT Invoices
        VALUES (@VendorID, @InvoiceNumber,
                 @InvoiceDate, @InvoiceTotal, 0, 0,
                 @TermsID, @InvoiceDueDate, NULL);
    ELSE
        THROW 50001, 'Not a valid VendorID!', 1;
```

A script that calls the procedure

```
BEGIN TRY
    EXEC spInsertInvoice
        799, 'ZXX-799', '2020-03-01', 299.95, 1, '2020-04-01';
END TRY
BEGIN CATCH
    PRINT 'An error occurred.';
    PRINT 'Message: ' + CONVERT(varchar, ERROR_MESSAGE());
    IF ERROR_NUMBER() >= 50000
        PRINT 'This is a custom error message.';
END CATCH;
```

The response from the system

```
An error occurred.
Message: Not a valid VendorID!
This is a custom error message.
```

The THROW statement coded within a block

```
BEGIN
    ;
    THROW 50001, 'Not a valid VendorID!', 1;
END;
```

Description

- The process of checking the values in one or more columns is known as *data validation*. It's a good practice to validate the data within a stored procedure whenever possible.
- The THROW statement manually raises an error. Unless this error is caught by a TRY...CATCH statement within the stored procedure, the error will be returned to the caller just like an error that's raised by the database engine.
- You use the state argument to identify how serious an error is. The severity of an error that's raised with the THROW statement is always 16.
- A THROW statement that's coded within a block must be preceded by a semicolon.
- A THROW statement that doesn't include any parameters must be coded in a CATCH block.

the error is informational, whether program execution should jump into a CATCH block, or whether the client connection should be terminated. Because you shouldn't throw system errors, though, and because you'll almost always want an error to be caught and handled, you can simplify your error handling by using THROW statements instead of RAISERROR statements.

A stored procedure that manages insert operations

Figure 15-8 presents a stored procedure that might be used by an application program that inserts new invoices into the Invoices table. This should give you a better idea of how you can use stored procedures.

This procedure starts with a comment that documents the stored procedure. This documentation includes the author's name, the date the procedure was created, the dates it was modified, who it was modified by, and a general description of the procedure's purpose. Since this procedure returns a value, the return value is briefly described in the comments too. Of course, you can include any other information that you feel is useful.

This procedure uses nine parameters that correspond to nine of the columns in the Invoices table. All of these parameters are input parameters, and each parameter is assigned the same data type as the matching column in the Invoices table. This means that if the calling program passes a value that can't be cast to the proper data type, an error will be raised as the procedure is called. In other words, this type of error won't be caught by the procedure.

If the calling program was to pass a value of 13-15-89 to the @InvoiceDate parameter, for example, an error would occur because this value can't be cast as a date. To handle this type of error within the procedure, you could define each parameter with the varchar data type. Then, the procedure could test for invalid data types and raise appropriate errors when necessary.

All of the input parameters are also assigned a default value of NULL. Since most of the columns in the Invoices table can't accept null values, this might seem like a problem. As you'll see in a minute, however, the procedure tests the value of each parameter before the insertion is attempted. Then, if the parameter contains an invalid value, an appropriate error is returned to the calling program and the insert operation is never performed. By coding the procedure this way, you can fix some errors by supplying default values, and you can return custom error messages for other errors.

A stored procedure that validates the data in a new invoice**Page 1**

```
/*
Handles insertion of new invoices into AP database,
including data validation.
Author:      Bryan Syverson
Created:    2002-07-17
Modified:   2008-07-29 by Joel Murach
            2020-01-31 by Anne Boehm
Return value: InvoiceID for the new row if successful,
               0 if unsuccessful
*/
USE AP;
GO

IF OBJECT_ID('spInsertInvoice') IS NOT NULL
    DROP PROC spInsertInvoice;
GO

CREATE PROC spInsertInvoice
    @VendorID      int = NULL,
    @InvoiceNumber  varchar(50) = NULL,
    @InvoiceDate    date = NULL,
    @InvoiceTotal   money = NULL,
    @PaymentTotal   money = NULL,
    @CreditTotal    money = NULL,
    @TermsID        int = NULL,
    @InvoiceDueDate date = NULL,
    @PaymentDate    date = NULL
AS
```

Description

- The nine parameters used in this procedure correspond to nine of the columns in the Invoices table.

Figure 15-8 A stored procedure that manages insert operations (part 1 of 3)

After the AS keyword, a series of IF statements are used to test the values in each input parameter. The first IF statement, for example, tests the value of @VendorID to determine if that vendor exists in the Vendors table. If not, a THROW statement is used to raise a custom error that indicates that the VendorID is invalid. This statement also exits the stored procedure and returns the custom error to the calling program.

For the first IF statement, the custom error contains a short message that says, “Invalid VendorID.” However, this message could easily be enhanced to display a more helpful and descriptive message such as “The specified VendorID value does not exist in the Vendors table. Please specify a valid VendorID.”

The next five IF statements check for null values. In addition, the IF statement for the @InvoiceDate parameter checks to be sure that it falls between the current date and 30 days prior to the current date, and the IF statement for the @InvoiceTotal parameter checks to be sure that it’s greater than zero. If not, a custom error is raised.

Instead of returning an error if the @CreditTotal or @PaymentTotal parameter contains a null value, this procedure sets the value of the parameter to zero. It also checks that the credit total isn’t greater than the invoice total, and it checks that the payment total isn’t greater than the invoice total minus the credit total.

The next IF statement checks the value of the @TermsID parameter to see if a row with this value exists in the Terms table. If not, this parameter is set to the value of the DefaultTermsId column for the vendor if the parameter contains a null value. If it contains any other value, though, a custom error is raised.

This procedure also sets the @InvoiceDueDate parameter if a due date isn’t passed to the procedure. To do that, it gets the value of the TermsDueDays column from the Terms table and stores it in a variable, and then uses the DATEADD function to add the value of that variable to the @InvoiceDate parameter. You might think that you could calculate the invoice due date using a SET statement like this:

```
SET @InvoiceDueDate = @InvoiceDate +
    (SELECT TermsDueDays FROM Terms WHERE TermsID = @TermsID);
```

You can’t do that, though, because as you learned in chapter 9, you can’t use the addition and subtraction operators with the date data type, and all three of the date parameters for this procedure are declared with that type.

If a due date is passed to the procedure, the procedure checks that the date is after the invoice date but isn’t more than 180 days after the invoice date. Finally, the procedure checks the @PaymentDate parameter to be sure that it’s not less than the invoice date or more than 14 days before the current date.

As you can see, some of the data validation performed by this procedure duplicates the constraints for the Invoices table. If you didn’t check the VendorID to be sure that it existed in the Vendors table, for example, the foreign key constraint would cause an error to occur when the row was inserted. By testing for the error before the insertion, however, the stored procedure can raise a custom error message that might be more easily handled by the calling program.

Some of the data validation performed by this procedure goes beyond the constraints for the columns in the table. For example, all three of the date values

A stored procedure that validates the data in a new invoice**Page 2**

```

IF NOT EXISTS (SELECT * FROM Vendors WHERE VendorID = @VendorID)
    THROW 50001, 'Invalid VendorID.', 1;
IF @InvoiceNumber IS NULL
    THROW 50001, 'Invalid InvoiceNumber.', 1;
IF @InvoiceDate IS NULL OR @InvoiceDate > GETDATE()
    OR DATEDIFF(dd, @InvoiceDate, GETDATE()) > 30
    THROW 50001, 'Invalid InvoiceDate.', 1;
IF @InvoiceTotal IS NULL OR @InvoiceTotal <= 0
    THROW 50001, 'Invalid InvoiceTotal.', 1;
IF @PaymentTotal IS NULL
    SET @PaymentTotal = 0;
IF @CreditTotal IS NULL
    SET @CreditTotal = 0;
IF @CreditTotal > @InvoiceTotal
    THROW 50001, 'Invalid CreditTotal.', 1;
IF @PaymentTotal > @InvoiceTotal - @CreditTotal
    THROW 50001, 'Invalid PaymentTotal.', 1;
IF NOT EXISTS (SELECT * FROM Terms WHERE TermsID = @TermsID)
    IF @TermsID IS NULL
        SELECT @TermsID = DefaultTermsID
        FROM Vendors
        WHERE VendorID = @VendorID;
    ELSE -- @TermsID IS NOT NULL
        THROW 50001, 'Invalid TermsID.', 1;
IF @InvoiceDueDate IS NULL
    BEGIN
        DECLARE @TermsDueDays int;
        SELECT @TermsDueDays = TermsDueDays
        FROM Terms
        WHERE TermsID = @TermsID;
        SET @InvoiceDueDate = DATEADD(day, @TermsDueDays, @InvoiceDate);
    END
ELSE -- @InvoiceDueDate IS NOT NULL
    IF @InvoiceDueDate < @InvoiceDate OR
        DATEDIFF(dd, @InvoiceDueDate, @InvoiceDate) > 180
        THROW 50001, 'Invalid InvoiceDueDate.', 1;
    IF @PaymentDate < @InvoiceDate OR
        DATEDIFF(dd, @PaymentDate, GETDATE()) > 14
        THROW 50001, 'Invalid PaymentDate.', 1;

```

Description

- A series of IF statements is used to validate the data in each column of the new invoice row. If the value in a column is invalid, a THROW statement is used to return a custom error to the calling program. This terminates the stored procedure.
- Some of the conditions tested by this code could be accomplished using constraints. However, testing these conditions before the INSERT statement is executed prevents system errors from occurring and allows you to return a user-friendly custom error message. Other conditions tested by this code can't be enforced using constraints.

Figure 15-8 A stored procedure that manages insert operations (part 2 of 3)

are tested to determine whether they fall within an appropriate range. This illustrates the flexibility provided by using stored procedures to validate data.

If the input parameters pass all of the validation tests, the INSERT statement is executed. Since the data has already been validated, the INSERT statement should execute successfully most of the time and insert the row. Then, the following statement uses the @@IDENTITY function to get the new invoice ID value that's generated when the row is inserted, and it returns that value to the calling program.

However, even though the data for the insert has been validated, it's still possible for an unexpected system error to occur. In that case, SQL Server will raise a system error and end the stored procedure.

In most cases, a stored procedure like this would be called from an application program. Since the details of doing that are beyond the scope of this book, however, this figure presents a SQL script that calls the procedure. This script includes processing similar to what might be used in an application program. In short, this script uses a TRY...CATCH statement to catch any errors that have been raised and to handle them appropriately. In many cases, handling an error is as simple as displaying a message that describes the error and indicates what can be done to fix the problem. In this figure, for example, the CATCH block uses three PRINT statements to indicate that an error occurred, to display the error number, and to display the error message. In other cases, though, error handling can include additional processing such as saving data or exiting the program as gracefully as possible.

A stored procedure that validates the data in a new invoice**Page 3**

```
INSERT Invoices
VALUES (@VendorID, @InvoiceNumber, @InvoiceDate, @InvoiceTotal,
        @PaymentTotal, @CreditTotal, @TermsID, @InvoiceDueDate,
        @PaymentDate);
RETURN @@IDENTITY;
```

A SQL script that calls the stored procedure

```
BEGIN TRY
    DECLARE @InvoiceID int;
    EXEC @InvoiceID = spInsertInvoice
        @VendorID = 799,
        @InvoiceNumber = 'RZ99381',
        @InvoiceDate = '2020-02-12',
        @InvoiceTotal = 1292.45;
    PRINT 'Row was inserted.';
    PRINT 'New InvoiceID: ' + CONVERT(varchar, @InvoiceID);
END TRY
BEGIN CATCH
    PRINT 'An error occurred. Row was not inserted.';
    PRINT 'Error number: ' + CONVERT(varchar, ERROR_NUMBER());
    PRINT 'Error message: ' + CONVERT(varchar, ERROR_MESSAGE());
END CATCH;
```

The response from the system for a successful insert

```
Row was inserted.
New InvoiceID: 115
```

The response from the system when a validation error occurs

```
An error occurred. Row was not inserted.
Error number: 50001
Error message: Invalid VendorID.
```

Description

- If the data in all of the columns of the new row is valid, the procedure executes an INSERT statement to insert the row. If the insert succeeds, this procedure gets the new InvoiceID value and returns it to the calling program, which ends the procedure. Otherwise, the database engine raises a system error, returns a value of zero, and ends the procedure.
- If this procedure was called by an application program, the program would need to handle any errors that occur. This includes custom errors raised by this stored procedure and unexpected system errors raised by SQL Server if it can't execute the INSERT statement.

How to pass a table as a parameter

So far, all of the stored procedures shown in this chapter have accepted scalar values as parameters. However, there are times when you may want to pass an entire table to a stored procedure. For example, you may want to pass multiple invoices or line items to a stored procedure for processing. With SQL Server 2008 and later, you can pass a table as a parameter as shown in figure 15-9.

Before you can pass a table as a parameter, you must define a data type for the table. In other words, you must create a *user-defined table type*. To do that, you can code a CREATE TYPE statement like the one shown in this figure. This statement defines a data type named LineItems for a table that contains columns similar to the columns of the InvoiceLineItems table, including the same definition for the primary key. However, foreign keys aren't allowed for user-defined table types. As a result, the column definitions in this figure don't include foreign keys.

Once you define a table type, you can create a stored procedure that accepts this data type as a parameter. In this figure, for example, the CREATE PROC statement creates a procedure that accepts a single parameter named @LineItems of the LineItems type. As a result, the body of the stored procedure can treat this parameter just as if it was a table variable. Here, an INSERT statement inserts all of the rows stored in the @LineItems parameter into the InvoiceLineItems table. This works because the LineItems type contains the same number and type of columns as the InvoiceLineItems table.

When you code a stored procedure that accepts a table as a parameter, you must use the READONLY keyword to identify this parameter as read-only. As a result, you can't modify the data that's stored in the parameter. However, outside of the stored procedure, you can modify the data that's stored in a variable of a user-defined table type. In this figure, for example, the code that passes the table to the stored procedure begins by declaring a variable of the LineItems type. Then, it uses three INSERT statements to insert three rows into the table. Finally, it uses an EXEC statement to pass this table to the procedure.

If you need to, you can also join a table that's passed as a parameter with another table. For example, suppose you want to update the InvoiceLineItems table with data from the @LineItems table. To do that, you could code a statement like this:

```
UPDATE InvoiceLineItems
SET InvoiceLineItemAmount = li.ItemAmount
FROM InvoiceLineItems i JOIN @LineItems li
ON i.InvoiceID = li.InvoiceID
AND i.InvoiceSequence = li.InvoiceSequence;
```

Here, the InvoiceLineItems table has a primary key that's defined by two columns. As a result, tables are joined based on both of these columns.

The syntax for creating a user-defined table type

```
CREATE TYPE TableTypeName AS
TABLE
table_definition
```

A statement that creates a user-defined table type

```
CREATE TYPE LineItems AS
TABLE
(InvoiceID      INT          NOT NULL,
InvoiceSequence SMALLINT     NOT NULL,
AccountNo       INT          NOT NULL,
ItemAmount      MONEY        NOT NULL,
ItemDescription VARCHAR(100) NOT NULL,
PRIMARY KEY (InvoiceID, InvoiceSequence));
```

A statement that creates a stored procedure that accepts a table as a parameter

```
CREATE PROC spInsertLineItems
    @LineItems LineItems READONLY
AS
    INSERT INTO InvoiceLineItems
    SELECT *
    FROM @LineItems;
```

Statements that pass a table to a stored procedure

```
DECLARE @LineItems LineItems;

INSERT INTO @LineItems VALUES (114, 1, 553, 127.75, 'Freight');
INSERT INTO @LineItems VALUES (114, 2, 553, 29.25, 'Freight');
INSERT INTO @LineItems VALUES (114, 3, 553, 48.50, 'Freight');

EXEC spInsertLineItems @LineItems;
```

The response from the system

```
(1 row affected)
(1 row affected)
(1 row affected)
(3 rows affected)
```

Description

- If you want to pass a table as a parameter to a stored procedure or a user-defined function, you must create a *user-defined table type* for the table.
- A user-defined table type can only be used as an input parameter, not as an output parameter.
- Creating a user-defined table type is similar to creating a regular table. However, you can't define foreign keys for the table.

How to delete or change a stored procedure

Figure 15-10 presents the syntax of the DROP PROC statement. You use this statement to delete one or more stored procedures from the database. As with the other statements you've learned that delete objects from the database, the deletion is permanent.

This figure also presents the syntax of the ALTER PROC statement. You use this statement to redefine an existing stored procedure. As you can see, the syntax is the same as the syntax of the CREATE PROC statement.

Like the ALTER VIEW statement, ALTER PROC completely replaces the previous definition for the stored procedure. Because of that, you'll usually change the definition of a stored procedure by deleting the procedure and then recreating it. If you've assigned security permissions to restrict the users who can call the procedure, however, those permissions are lost when you delete the procedure. If you want to retain the permissions, then, you should use the ALTER PROC statement instead.

The examples in this figure show how you might use the ALTER PROC and DROP PROC statements. The first example creates a stored procedure named spVendorState that selects vendors from the state specified by the @State parameter. Because the SELECT statement will fail if a state isn't specified, this parameter is required. In the second example, however, an ALTER PROC statement is used to modify this procedure so the state is optional. The last example deletes this procedure.

If you delete a table or view used by a stored procedure, you should be sure to delete the stored procedure as well. If you don't, the stored procedure can still be called by any user or program that has access to it. Then, an error will occur because the table or view has been deleted.

The syntax of the DROP PROC statement

```
DROP {PROC|PROCEDURE} procedure_name [, ...]
```

The syntax of the ALTER PROC statement

```
ALTER {PROC|PROCEDURE} procedure_name  
[parameter declarations]  
[WITH [RECOMPILE] [, ENCRYPTION] [, EXECUTE_AS_clause]]  
AS sql_statements
```

A statement that creates a procedure

```
CREATE PROC spVendorState  
    @State varchar(20)  
AS  
SELECT VendorName  
FROM Vendors  
WHERE VendorState = @State;
```

A statement that changes the parameter defined by the procedure

```
ALTER PROC spVendorState  
    @State varchar(20) = NULL  
AS  
IF @State IS NULL  
    SELECT VendorName  
    FROM Vendors;  
ELSE  
    SELECT VendorName  
    FROM Vendors  
    WHERE VendorState = @State;
```

A statement that deletes the procedure

```
DROP PROC spVendorState;
```

Description

- To delete a stored procedure from the database, use the DROP PROC statement.
- To modify the definition of a procedure, you can delete the procedure and then create it again, or you can use the ALTER PROC statement to specify the new definition.
- When you delete a procedure, any security permissions that are assigned to the procedure are also deleted. If that's not what you want, you can use the ALTER PROC statement to modify the procedure and preserve the permissions.

How to work with system stored procedures

SQL Server comes with hundreds of *system stored procedures* that you can use to manage and maintain your databases. These procedures are stored in the Master database, but you can call them from any database. Figure 15-11 presents a table of commonly used system stored procedures.

This figure also presents a script that calls the `sp_HelpText` system stored procedure. This procedure returns the SQL code that was specified by the `CREATE` statement for a view, stored procedure, user-defined function, or trigger. If the object was created with the `WITH ENCRYPTION` option, however, the SQL code can't be returned. Because the code for system stored procedures is never encrypted, you can examine the code in these procedures too.

You can use the system stored procedures to simplify your administrative tasks. However, you may want to avoid using these procedures in production programs. That's because each time a new version of SQL Server is released, some of these stored procedures may change. Then, you may have to rewrite the programs that use them.

In addition to the system stored procedures provided by SQL Server, you can also create your own system stored procedures. To do that, you create the procedure in the Master database, and you give the procedure a name that starts with `sp_`.

Commonly used system stored procedures

Procedure	Description
<code>sp_Help [name]</code>	Returns information about the specified database object or data type. Without a parameter, returns a summary of all objects in the current database.
<code>sp_HelpText name</code>	Returns the text of an unencrypted stored procedure, user-defined function, trigger, or view.
<code>sp_HelpDb [database_name]</code>	Returns information about the specified database or, if no parameter is specified, all databases.
<code>sp_Who [login_ID]</code>	Returns information about who is currently logged in and what processes are running. If no parameter is specified, information on all active users is returned.
<code>sp_Columns name</code>	Returns information about the columns defined in the specified table or view.

How to use the `sp_HelpText` system stored procedure

```
USE AP;
EXEC sp_HelpText spInvoiceReport;
```

The results returned by the procedure

Text
1 CREATE PROC spInvoiceReport
2 AS
3
4 SELECT VendorName, InvoiceNumber, InvoiceDate, I...
5 FROM Invoices JOIN Vendors
6 ON Invoices.VendorID = Vendors.VendorID
7 WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0
8 ORDER BY VendorName;

The results if WITH ENCRYPTION is included in the procedure definition

The text for object 'spInvoiceReport' is encrypted.

Description

- Microsoft SQL Server includes many *system stored procedures* that you can use to perform useful tasks on a database. These procedures are identified by the prefix `sp_`. You can use these procedures in the scripts and procedures you write.
- System stored procedures are stored in the Master database, but you can execute them on any database. These procedures operate within the current database context that you've set with the USE statement.
- SQL Server has hundreds of system stored procedures. To view the complete list, look up “system stored procedures” in the SQL Server documentation.
- You can also create your own system stored procedures. To do that, give the procedure a name that begins with `sp_` and create it in the Master database.

Figure 15-11 How to work with system stored procedures

How to code user-defined functions

In addition to the SQL Server functions you've learned about throughout this book, you can also create your own functions, called user-defined functions. To do that, you use code that's similar to the code you use to create a stored procedure. There are some distinct differences between stored procedures and user-defined functions, however. You'll learn about those differences in the topics that follow.

An introduction to user-defined functions

Figure 15-12 summarizes the three types of user-defined functions, also called *UDFs*, or just *functions*, that you can create using Transact-SQL.

Scalar-valued functions are like the functions you learned about in chapter 9 that return a single value. In addition to scalar-valued functions, however, you can also create *table-valued functions*. As its name implies, a table-valued function returns an entire table. A table-valued function that's based on a single SELECT statement is called a *simple table-valued function*. By contrast, a table-valued function that's based on multiple SQL statements is called a *multi-statement table-valued function*.

Like a stored procedure, a UDF can accept one or more input parameters. The function shown in this figure, for example, accepts a parameter named @VendorName. However, a UDF can't be defined with output parameters. Instead, the RETURN statement must be used to pass a value back to the calling program. The value that's returned must be compatible with the data type that's specified in the RETURNS clause. In this example, an integer that contains a VendorID value selected from the Vendors table is returned.

To call, or *invoke*, a scalar-valued function, you include it in an expression. Then, the value returned by the function is substituted for the function. The first SELECT statement in this figure, for example, uses the value returned by the fnVendorID function in its WHERE clause. Note that when you refer to a user-defined function, you must include the name of the schema. In this case, the schema is dbo.

To invoke a table-valued function, you refer to it anywhere you would normally code a table or view name. The second SELECT statement in this figure, for example, uses a function named fnTopVendorsDue in the FROM clause. You'll see the definition of this function later in this chapter.

Unlike a stored procedure, a UDF can't make permanent changes to the objects in a database. For example, it can't issue INSERT, UPDATE, and DELETE statements against tables or views in the database. However, within the code for a function, you can create a table, a temporary table, or a table variable. Then, the function can perform insert, update, and delete operations on that table.

The three types of user-defined functions

Function type	Description
Scalar-valued function	Returns a single value of any T-SQL data type.
Simple table-valued function	Returns a table that's based on a single SELECT statement.
Multi-statement table-valued function	Returns a table that's based on multiple statements.

A statement that creates a scalar-valued function

```
CREATE FUNCTION fnVendorID
    (@VendorName varchar(50))
    RETURNS int
BEGIN
    RETURN (SELECT VendorID FROM Vendors WHERE VendorName = @VendorName);
END;
```

A statement that invokes the scalar-valued function

```
SELECT InvoiceDate, InvoiceTotal
FROM Invoices
WHERE VendorID = dbo.fnVendorID('IBM');
```

A statement that invokes a table-valued function

```
SELECT * FROM dbo.fnTopVendorsDue(5000);
```

Description

- A user-defined function, also called a *UDF* or just a *function*, is an executable database object that contains SQL statements.
- The name of a function can be up to 128 characters and is typically prefixed with the letters *fn*.
- Functions always return a value. A *scalar-valued function* returns a single value of any T-SQL data type. A *table-valued function* returns an entire table.
- A table-valued function can be based on a single SELECT statement, in which case it's called a *simple table-valued function*, or it can be based on two or more statements, in which case it's called a *multi-statement table-valued function*.
- A function can't have a permanent effect on the database. In other words, it can't run an action query against the database.
- You can call, or *invoke*, a scalar-valued function from within any expression. You can invoke a table-valued function anywhere you'd refer to a table or a view.
- Unlike other database objects, you must specify the name of the schema when invoking a UDF.

Figure 15-12 An introduction to user-defined functions

How to create a scalar-valued function

Figure 15-13 presents the syntax of the CREATE FUNCTION statement you use to create a scalar-valued function. The CREATE FUNCTION clause names the function and declares the input parameters. If you don't specify the schema name as part of the name, the function is stored in the schema that's associated with the current user.

The syntax you use to declare parameters for a function is similar to the syntax you use to declare parameters for stored procedures. For a function, however, the declarations must be enclosed in parentheses. In addition, because a function can't have output parameters, the OUTPUT keyword isn't allowed.

To invoke a function that has parameters, you must pass the parameters by position. You can't pass them by name as you can when you call a stored procedure. For this reason, you should code required parameters first, followed by optional parameters. Furthermore, you can't simply omit optional parameters when invoking a function as you can with a stored procedure. Instead, you must use the DEFAULT keyword as a placeholder for the optional parameter. You'll see an example of that in figure 15-14.

The RETURNS clause specifies the data type of the value that's returned by the function. Because the value must be scalar, you can't specify the table data type. In addition, you can't specify text, ntext, image, or timestamp.

You code the statements for the function within a BEGIN...END block. Within that block, the RETURN statement specifies the value that's passed back to the invoking program. Since this statement causes the function to terminate, it's usually coded at the end of the function. Notice that unlike a RETURN statement you code within a stored procedure, a RETURN statement you code within a function can return a value with any data type. Within a specific function, however, it must return a value with a data type that's compatible with the data type specified by the RETURNS clause.

The scalar-valued function that's shown in this figure doesn't accept any input parameters and returns a value with the money data type. In this case, the code for the function consists of a single SELECT statement coded within the RETURN statement. However, a function can include as many statements as are necessary to calculate the return value.

If you find yourself repeatedly coding the same expression, you may want to create a scalar-valued function for the expression. Then, you can use that function in place of the expression, which can save you coding time and make your code easier to maintain. Most SQL programmers create a set of useful UDFs each time they work on a new database.

The syntax for creating a scalar-valued function

```
CREATE FUNCTION [schema_name.]function_name
    ([@parameter_name data_type [= default]] [, ...])
    RETURNS data_type
    [WITH [ENCRYPTION] [, SCHEMABINDING] [, EXECUTE_AS_clause]]
    [AS]
BEGIN
    [sql_statements]
    RETURN scalar_expression
END
```

A statement that creates a scalar-valued function that returns the total invoice amount due

```
CREATE FUNCTION fnBalanceDue()
    RETURNS money
BEGIN
    RETURN (SELECT SUM(InvoiceTotal - PaymentTotal - CreditTotal)
            FROM Invoices
            WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0);
END;
```

A script that invokes the function

```
PRINT 'Balance due: $' + CONVERT(varchar, dbo.fnBalanceDue(), 1);
```

The response from the system

```
Balance due: $32,020.42
```

Description

- Functions can be defined with from zero to 1024 input parameters. You specify these parameters in parentheses after the name of the function in the CREATE FUNCTION statement. Each parameter can be assigned an optional default value.
- A function can't contain output parameters. Instead, you specify the data type of the data to be returned by the function in the RETURNS clause.
- You code the statements that define the function within a BEGIN...END block. This block includes a RETURN statement that specifies the value to be returned.
- When you invoke a function, you list the parameters within parentheses after the name of the function. To use the default value of a parameter, code the DEFAULT keyword in place of the parameter value. You can't pass function parameters by name.
- When you create a function, it's stored in the schema associated with the current user if you don't specify a schema name. When you invoke the function, however, you must specify the schema name.
- The SCHEMABINDING option binds the function to the database schema. This prevents you from dropping or altering tables or views that are used by the function. This option is more commonly used for table-valued functions.
- The ENCRYPTION option prevents users from viewing the code in the function.
- The EXECUTE_AS_clause option specifies the security context under which the function is executed.

Figure 15-13 How to create and use a scalar-valued function

How to create a simple table-valued function

Figure 15-14 presents the syntax for creating a simple table-valued function, also called an *inline table-valued function*. You use this syntax if the result set can be returned from a single SELECT statement. Otherwise, you'll need to use the syntax that's presented in the next figure.

To declare the function as table-valued, you code the table data type in the RETURNS clause. Then, you code the SELECT statement that defines the table in parentheses in the RETURN statement. Note that because a table can't have any unnamed columns, you must assign a name to every calculated column in the result set.

The function shown in this figure returns a table that contains the vendor name and total balance due for each vendor with a balance due. The one input parameter, @CutOff, is an optional parameter because it's assigned a default value of 0. This parameter is used in the HAVING clause to return only those vendors with total invoices that are greater than or equal to the specified amount. The first SELECT statement shown in this figure, for example, returns vendors with total invoices greater than or equal to \$5,000.

The second SELECT statement shows how you can join the result of a table-valued function with another table. Notice that to avoid having to code the function in both the FROM and ON clauses, the function is assigned a correlation name. Also notice in this example that a value isn't specified for the optional parameter. Instead, the DEFAULT keyword is specified so the default value of the parameter will be used.

A table-valued function like the one shown here acts like a dynamic view. Because a function can accept parameters, the result set it creates can be modified. This is a powerful extension to standard SQL functionality.

The syntax for creating a simple table-valued function

```
CREATE FUNCTION [schema_name.]function_name
    ([@parameter_name data_type [= default]] [, ...])
    RETURNS TABLE
    [WITH [ENCRYPTION] [, SCHEMABINDING]]
    [AS]
    RETURN [() select_statement ()]
```

A statement that creates a simple table-valued function

```
CREATE FUNCTION fnTopVendorsDue
    (@CutOff money = 0)
    RETURNS table
RETURN
    (SELECT VendorName, SUM(InvoiceTotal) AS TotalDue
    FROM Vendors JOIN Invoices ON Vendors.VendorID = Invoices.VendorID
    WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0
    GROUP BY VendorName
    HAVING SUM(InvoiceTotal) >= @CutOff);
```

A SELECT statement that invokes the function

```
SELECT * FROM dbo.fnTopVendorsDue(5000);
```

The result set

	VendorName	TotalDue
1	Malloy Lithographing Inc	31527.24

A SELECT statement that uses the function in a join operation

```
SELECT Vendors.VendorName, VendorCity, TotalDue
FROM Vendors JOIN dbo.fnTopVendorsDue(DEFAULT) AS TopVendors
    ON Vendors.VendorName = TopVendors.VendorName;
```

The result set

	VendorName	VendorCity	TotalDue
1	Blue Cross	Oxnard	224.00
2	Cardinal Business Media, Inc.	Philadelphia	90.36
3	Data Reproductions Corp	Auburn Hills	85.31
4	Federal Express Corporation	Memphis	210.89
5	Ford Motor Credit Company	Los Angeles	503.20
6	Ingram	Dallas	579.42
7	Malloy Lithographing Inc	Ann Arbor	31527.24

Description

- You create a simple table-valued function, also called an *inline table-valued function*, by coding the table data type in the RETURNS clause of the CREATE FUNCTION statement. Then, you code a SELECT statement that defines the table in the RETURN statement.
- To use a simple table-valued function, code the function name in place of a table name or a view name. If you use a table-valued function in a join operation, you'll want to assign a correlation name to it as shown above.

Figure 15-14 How to create and use a simple table-valued function

How to create a multi-statement table-valued function

Figure 15-15 presents the syntax for creating a multi-statement table-valued function. Although you should know about this syntax, you'll probably never need to use it. That's because a single SELECT statement with joins and subqueries can fulfill almost every query need.

Since a multi-statement table-valued function creates a new table, you must define the structure of that table. To do that, you declare a table variable in the RETURNS clause and then define the columns for the new table. The syntax you use to define the columns is similar to the syntax you use to define the columns of a table variable.

You code the SQL statements that create the table within a BEGIN...END block. This block ends with a RETURN keyword with no argument. This terminates the function and returns the table variable to the invoking program.

The function shown in this figure returns a table with one row for each invoice with a balance due. This function calculates the credit adjustment that would be necessary to reduce the total balance due to the threshold amount that's passed to the function. This function is similar to the script you saw in figure 14-9 in chapter 14 that uses a temporary table.

This function starts by using an INSERT statement to copy all the rows in the Invoices table with a balance due to the @OutTable table variable. Then, a WHILE statement is used to increment the CreditTotal column of each row in this table by one cent until the total amount due for all invoices falls below the threshold. The SELECT statement that uses this function summarizes the CreditTotal column by vendor.

The syntax for creating a multi-statement table-valued function

```
CREATE FUNCTION [schema_name.]function_name
    ([@parameter_name data_type [= default]] [, ...])
    RETURNS @return_variable TABLE
        (column_name_1 data_type [column_attributes]
        [, column_name_2 data_type [column_attributes]]...)
        [WITH [ENCRYPTION] [, SCHEMABINDING] [, EXECUTE_AS_clause]]
        [AS]
BEGIN
    sql_statements
    RETURN
END
```

A statement that creates a multi-statement table-valued function

```
CREATE FUNCTION fnCreditAdj (@HowMuch money)
    RETURNS @OutTable table
        (InvoiceID int, VendorID int, InvoiceNumber varchar(50),
        InvoiceDate date, InvoiceTotal money,
        PaymentTotal money, CreditTotal money)
BEGIN
    INSERT @OutTable
        SELECT InvoiceID, VendorID, InvoiceNumber, InvoiceDate,
            InvoiceTotal, PaymentTotal, CreditTotal
        FROM Invoices
        WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0;
    WHILE (SELECT SUM(InvoiceTotal - CreditTotal - PaymentTotal)
        FROM @OutTable) >= @HowMuch
        UPDATE @OutTable
        SET CreditTotal = CreditTotal + .01
        WHERE InvoiceTotal - CreditTotal - PaymentTotal > 0;
    RETURN;
END;
```

A SELECT statement that uses the function

```
SELECT VendorName, SUM(CreditTotal) AS CreditRequest
FROM Vendors JOIN dbo.fnCreditAdj(25000) AS CreditTable
    ON Vendors.VendorID = CreditTable.VendorID
GROUP BY VendorName;
```

The response from the system

	VendorName	CreditRequest
1	Blue Cross	224.00
2	Cardinal Business Media, Inc.	90.36
3	Data Reproductions Corp	85.31
4	Federal Express Corporation	210.89
5	Ford Motor Credit Company	503.20
6	Ingram	579.42
7	Malloy Lithographing Inc	6527.26

Warning

- Because this code must loop through the Invoices table thousands of times, this function can take several seconds or more to execute.

Figure 15-15 How to create and use a multi-statement table-valued function

How to delete or change a function

Figure 15-16 presents the syntax of the DROP FUNCTION and ALTER FUNCTION statements. The DROP FUNCTION statement permanently deletes one or more user-defined functions from the database. In addition, it drops any security permissions defined for the function along with any dependencies between the function and the tables and views it uses.

The ALTER FUNCTION statement modifies the definition of a user-defined function. You should use this statement if you need to preserve permissions and dependencies that would be lost if you dropped the function and then recreated it. Just like the CREATE FUNCTION statement, the ALTER FUNCTION statement has three syntax variations for the three types of functions you can create.

The syntax of the DROP FUNCTION statement

```
DROP FUNCTION [schema_name.]function_name [, ...]
```

The syntax of the ALTER FUNCTION statement for a scalar-valued function

```
ALTER FUNCTION [schema_name.]function_name
    ([@parameter_name data_type [= default]] [, ...])
    RETURNS data_type
    [WITH [ENCRYPTION] [, SCHEMABINDING] [, EXECUTE_AS_clause]]
BEGIN
    [sql_statements]
    RETURN scalar_expression
END
```

The syntax for altering a simple table-valued function

```
ALTER FUNCTION [schema_name.]function_name
    ([@parameter_name data_type [= default]] [, ...])
    RETURNS TABLE
    [WITH [ENCRYPTION] [, SCHEMABINDING]]
    RETURN [(] select_statement [)]
```

The syntax for altering a multi-statement table-valued function

```
ALTER FUNCTION [schema_name.]function_name
    ([@parameter_name data_type [= default]] [, ...])
    RETURNS @return_variable TABLE
    (column_name_1 data_type [column_attributes]
    [, column_name_2 data_type [column_attributes]]...)
    [WITH [ENCRYPTION] [, SCHEMABINDING] [, EXECUTE_AS_clause]]
BEGIN
    sql_statements
    RETURN
END
```

Description

- To delete a user-defined function from the database, use the DROP FUNCTION statement.
- To modify the definition of a function, you can delete the function and then create it again, or you can use the ALTER FUNCTION statement to specify the new definition.
- When you delete a function, any security permissions that are assigned to the function and any dependencies between the function and the tables and views it uses are also deleted. If that's not what you want, you can use the ALTER FUNCTION statement to modify the function and preserve the permissions and dependencies.

Figure 15-16 How to delete or change a function

How to code triggers

A trigger is a special type of procedure that's invoked, or *fired*, automatically when an action query is executed on a table or view. Triggers provide a powerful way to control how action queries modify the data in your database. If necessary, you can use triggers to enforce design rules, implement business logic, and prevent data inconsistency. However, triggers can cause locking and performance problems. In addition, there's often a better way to accomplish a task than by using a trigger. As a result, I recommend you use them sparingly.

How to create a trigger

Figure 15-17 presents the syntax of the CREATE TRIGGER statement you use to create a trigger. Notice in this syntax that a trigger can't use parameters. In addition, a trigger can't return a value.

The CREATE TRIGGER statement provides for two types of triggers: AFTER triggers and INSTEAD OF triggers. Both types of triggers can be defined to fire for an insert, update, or delete operation or any combination of these operations. If an action query has an AFTER trigger, the trigger fires after the action query. If an action query has an INSTEAD OF trigger, the trigger is fired instead of the action query. In other words, the action query is never executed.

In addition to AFTER and INSTEAD OF, you can code the FOR keyword in the CREATE TRIGGER statement. A FOR trigger is identical to an AFTER trigger. FOR is an ANSI-standard keyword and is more commonly used than AFTER. However, this book uses AFTER since it more clearly describes when the trigger fires.

Each trigger is associated with the table or view named in the ON clause. Although each trigger is associated with a single table or view, a single table can have any number of AFTER triggers. Since two or more triggers for the same table can be confusing to manage and debug, however, I recommend you have no more than one trigger for each action. Each table or view can also have one INSTEAD OF trigger for each action. A view can't have AFTER triggers.

The CREATE TRIGGER statement in this figure defines an AFTER trigger for the Vendors table. Notice that the name of this trigger reflects the table it's associated with and the operations that will cause it to fire. This is a common naming convention. In this case, the trigger fires after an insert or update operation is performed on the table. As you can see, the trigger updates the VendorState column so state codes are in uppercase letters.

Notice that the WHERE clause in the trigger uses a subquery that's based on a table named Inserted. This is a special table that's created by SQL Server during an insert operation. It contains the rows that are being inserted into the table. Since this table only exists while the trigger is executing, you can only refer to it in the trigger code.

The syntax of the CREATE TRIGGER statement

```
CREATE TRIGGER trigger_name
    ON {table_name|view_name}
    [WITH [ENCRYPTION] [,] [EXECUTE_AS_clause]]
    {FOR|AFTER|INSTEAD OF} [INSERT] [,] [UPDATE] [,] [DELETE]
    AS sql_statements
```

A CREATE TRIGGER statement that corrects mixed-case state names

```
CREATE TRIGGER Vendors_INSERT_UPDATE
    ON Vendors
    AFTER INSERT,UPDATE
AS
    UPDATE Vendors
    SET VendorState = UPPER(VendorState)
    WHERE VendorID IN (SELECT VendorID FROM Inserted);
```

An INSERT statement that fires the trigger

```
INSERT Vendors
VALUES ('Peerless Uniforms, Inc.', '785 S Pixley Rd', NULL,
        'Piqua', 'OH', '45356', '(937) 555-8845', NULL, NULL, 4, 550);
```

The new row that's inserted into the Vendors table

	VendorID	VendorName	VendorAddress1	VendorAddress2	VendorCity	VendorState	VendorZip
1	125	Peerless Uniforms, Inc.	785 S Pixley Rd	NULL	Piqua	OH	45356

Description

- A trigger is a special kind of procedure that executes, or *fires*, in response to an action query. Unlike a stored procedure, you can't invoke a trigger directly, you can't pass parameters to a trigger, and a trigger can't pass back a return value.
- A trigger is associated with a single table or view, which you identify in the ON clause. The trigger can be set to fire on INSERT, UPDATE, or DELETE statements or on a combination of these statements.
- A trigger can be set to fire after the action query (AFTER) or instead of the action query (INSTEAD OF). A FOR trigger is the same as an AFTER trigger.
- A table can have multiple AFTER triggers, even for the same action. A view can't have an AFTER trigger. A table or view can have only one INSTEAD OF trigger for each action.
- To hide the code for the trigger from the user, include the ENCRYPTION option.
- To execute a trigger under a specific security context, include the EXECUTE_AS_clause option.
- It's a common programming practice to name triggers based on the table or view and the actions that will cause the trigger to fire.
- Within a trigger, you can refer to two tables that are created by the system: Inserted and Deleted. The Inserted table contains the new rows for insert and update operations. The Deleted table contains the original rows for update and delete operations.

Figure 15-17 How to create a trigger

Similarly, a table named Deleted that contains the rows that are being deleted is created by SQL Server during a delete operation. For an update operation, SQL Server creates both tables. In that case, the Inserted table contains the rows with the updated data, and the Deleted table contains the original data from the rows that are being updated.

How to use AFTER triggers

Figure 15-18 shows another example of an AFTER trigger. This trigger archives all rows deleted from the Invoices table by inserting the deleted rows into another table named InvoiceArchive.

The CREATE TRIGGER statement shown in this figure begins by specifying a name of Invoices_DELETE. Then, it specifies that this trigger should execute after a DELETE statement is executed against the Invoices table. The body of this trigger uses an INSERT statement to insert the rows that have been deleted from the Invoices table into the InvoiceArchive table. To accomplish this, this trigger uses a SELECT statement to retrieve columns from the Deleted table.

The DELETE statement shown in this figure deletes three rows from Invoices table. This causes the Invoices_DELETE trigger to fire. As a result, the trigger inserts these three rows into the InvoiceArchive table after the DELETE statement executes.

Because an AFTER trigger fires after the action query is executed, the trigger doesn't fire if the action query causes an error. Usually, that's what you want. In this figure, for example, you wouldn't want to archive deleted rows if the DELETE statement caused an error and didn't execute successfully.

An AFTER trigger that archives deleted data

```
CREATE TRIGGER Invoices_DELETE
    ON Invoices
    AFTER DELETE
AS
    INSERT INTO InvoiceArchive
        (InvoiceID, VendorID, InvoiceNumber, InvoiceDate, InvoiceTotal,
         PaymentTotal, CreditTotal, TermsID, InvoiceDueDate, PaymentDate)
    SELECT InvoiceID, VendorID, InvoiceNumber, InvoiceDate, InvoiceTotal,
          PaymentTotal, CreditTotal, TermsID, InvoiceDueDate, PaymentDate
    FROM Deleted
```

A DELETE statement that causes the AFTER trigger to fire

```
DELETE Invoices
WHERE VendorID = 37
```

The rows inserted into the InvoiceArchive table

	InvoiceID	VendorID	InvoiceNumber	InvoiceDate	InvoiceTotal	PaymentTotal	CreditTotal	Term
1	113	37	547480102	2016-04-01	224.00	0.00	0.00	3
2	50	37	547479217	2016-02-07	116.00	116.00	0.00	3
3	46	37	547481328	2016-02-03	224.00	224.00	0.00	3

Description

- An AFTER trigger fires after the action query is executed. If the action query causes an error, the AFTER trigger never fires.
- AFTER triggers can be used to archive deleted data.

Figure 15-18 How to use AFTER triggers

How to use INSTEAD OF triggers

An INSTEAD OF trigger can be associated with either a table or a view. However, INSTEAD OF triggers are used most often to provide better control of updatable views.

Figure 15-19 presents an INSTEAD OF trigger that's used to control an insert operation through a view named IBM_Invoices. This view selects the InvoiceNumber, InvoiceDate, and InvoiceTotal from the Invoices table for the vendor named "IBM." If you look at the design of the Invoices table you'll see that there are six additional columns that don't allow null values. The problem is, how do you insert values into these six additional columns when the view isn't aware of their existence?

Three of the six columns (InvoiceID, PaymentTotal, and CreditTotal) don't need to be inserted through the view. That's because the design of the Invoices table automatically handles null values for these columns. InvoiceID is an identity column, so the database automatically inserts the value for that column. The other two columns have a default value of 0, so the database uses this default value.

This leaves three columns (VendorID, TermsID, and InvoiceDueDate) that must be updated when you insert data through the view. If you attempt to insert a row through the view, you can't specify these required columns, and the insert operation fails.

This trigger accommodates these missing columns by calculating their values based on three logical assumptions. First, the VendorID can be assumed because this view is explicitly for invoices for vendor "IBM." Second, the terms for the invoice can be assumed to be the default terms for the vendor. Third, the due date for the invoice can be calculated based on the invoice date and the terms.

After it declares the variables it uses, the trigger queries the Inserted table to get a count of the number of rows that are being inserted. Since this trigger will work only if a single row is being inserted, an error is raised if the row count is greater than one. Otherwise, the trigger queries the Inserted table to get the values of the three columns that were specified in an INSERT statement like the one shown in this figure. The SELECT statement assigns these values to three of the variables. Then, if all three variables contain values other than null, the trigger calculates the values of the missing columns.

Since an INSTEAD OF trigger is executed instead of the action query that caused it to fire, the action will never occur unless you code it as part of the trigger. For this reason, the last statement in this trigger is an INSERT statement that inserts the new row into the Invoices table. Without this statement, the row would never be inserted.

An INSTEAD OF INSERT trigger for a view

```

CREATE TRIGGER IBM_Invoices_INSERT
    ON IBM_Invoices
    INSTEAD OF INSERT
AS
DECLARE @InvoiceDate date, @InvoiceNumber varchar(50),
        @InvoiceTotal money, @VendorID int,
        @InvoiceDueDate date, @TermsID int,
        @DefaultTerms smallint, @TestRowCount int;
SELECT @TestRowCount = COUNT(*) FROM Inserted;
IF @TestRowCount = 1
    BEGIN
        SELECT @InvoiceNumber = InvoiceNumber, @InvoiceDate = InvoiceDate,
               @InvoiceTotal = InvoiceTotal
        FROM Inserted;
        IF (@InvoiceDate IS NOT NULL AND @InvoiceNumber IS NOT NULL AND
            @InvoiceTotal IS NOT NULL)
            BEGIN
                SELECT @VendorID = VendorID, @TermsID = DefaultTermsID
                FROM Vendors
                WHERE VendorName = 'IBM';

                SELECT @DefaultTerms = TermsDueDays
                FROM Terms
                WHERE TermsID = @TermsID;

                SET @InvoiceDueDate =
                    DATEADD(day, @DefaultTerms, @InvoiceDate);

                INSERT Invoices
                    (VendorID, InvoiceNumber, InvoiceDate, InvoiceTotal,
                     TermsID, InvoiceDueDate, PaymentDate)
                VALUES (@VendorID, @InvoiceNumber, @InvoiceDate,
                        @InvoiceTotal, @TermsID, @InvoiceDueDate, NULL);
            END;
    END;
ELSE
    THROW 50027, 'Limit INSERT to a single row.', 1;

```

An INSERT statement that succeeds due to the trigger

```

INSERT IBM_Invoices
VALUES ('RA23988', '2020-03-09', 417.34);

```

Description

- An INSTEAD OF trigger is executed instead of the action query that causes it to fire. Because the action query is never executed, the trigger typically contains code that performs the operation.
- INSTEAD OF triggers are typically used to provide for updatable views. They can also be used to prevent errors, such as constraint violations, before they occur.
- Each table or view can have only one INSTEAD OF trigger for each type of action. However, if a table is defined with a foreign key constraint that specifies the CASCADE UPDATE or CASCADE DELETE option, INSTEAD OF UPDATE and INSTEAD OF DELETE triggers can't be defined for the table.

Figure 15-19 How to use INSTEAD OF triggers

How to use triggers to enforce data consistency

Triggers can also be used to enforce data consistency. For example, the sum of line item amounts in the InvoiceLineItems table should always be equal to the invoice total for the invoice in the Invoices table. Unfortunately, you can't enforce this rule using a constraint on either the Invoices table or the InvoiceLineItems table. However, you can use a trigger like the one in figure 15-20 to enforce this rule when a payment amount is updated.

The trigger shown here fires after an update operation on the Invoices table. Since you can assume that posting a payment is likely to be the last action taken on an invoice, firing a trigger on this action is a good way to verify that all of the data is valid. If an update operation changes the PaymentTotal column, the rest of the trigger verifies that the sum of the line items is equal to the invoice total. If the data isn't valid, the trigger raises an error and uses the ROLLBACK TRAN statement to roll back the update. You'll learn more about using this statement in the next chapter.

Notice the two IF statements shown in this figure. They use the EXISTS operator to test for the existence of the data specified by the subqueries that follow. In chapter 6, you saw how to use the EXISTS operator in the WHERE clause. Because this keyword returns a Boolean value, however, you can use it in an IF statement as well.

You can use triggers like the one shown here to enforce business rules or verify data consistency. Since you can program a trigger to accommodate virtually any situation, triggers are more flexible than constraints. As a result, some programmers prefer to use triggers rather than constraints to enforce data consistency and sometimes even check constraints and defaults.

A trigger that validates line item amounts when posting a payment

```

CREATE TRIGGER Invoices_UPDATE
    ON Invoices
    AFTER UPDATE
AS
IF EXISTS          --Test whether PaymentTotal was changed
    (SELECT *
     FROM Deleted JOIN Invoices
     ON Deleted.InvoiceID = Invoices.InvoiceID
     WHERE Deleted.PaymentTotal <> Invoices.PaymentTotal)
BEGIN
    IF EXISTS      --Test whether line items total and InvoiceTotal match
        (SELECT *
         FROM Invoices JOIN
              (SELECT InvoiceID, SUM(InvoiceLineItemAmount) AS SumOfInvoices
               FROM InvoiceLineItems
               GROUP BY InvoiceID) AS LineItems
         ON Invoices.InvoiceID = LineItems.InvoiceID
         WHERE (Invoices.InvoiceTotal <> LineItems.SumOfInvoices) AND
               (LineItems.InvoiceID IN (SELECT InvoiceID FROM Deleted)))
    BEGIN
        ;
        THROW 50113, 'Correct line item amounts before posting payment.', 1;
        ROLLBACK TRAN;
    END;
END;

```

An UPDATE statement that fires the trigger

```

UPDATE Invoices
SET PaymentTotal = 662, PaymentDate = '2020-03-09'
WHERE InvoiceID = 98;

```

The response from the system

```

Msg 50113, Level 16, State 1, Procedure Invoices_UPDATE, Line 23
Correct line item amounts before posting payment.

```

Description

- Triggers can be used to enforce database rules for data consistency that can't be enforced by constraints.
- Triggers can also be used to enforce the same rules as constraints, but with more flexibility.

How to use triggers to work with DDL statements

So far, this chapter has only shown you how to create triggers for DML statements such as the INSERT, UPDATE, and DELETE statements since that's typically how triggers are used. However, you can also create triggers for DDL statements such as the CREATE TABLE statement. For example, figure 15-21 shows a trigger that is executed when any CREATE TABLE or DROP TABLE statement is executed on the current database.

Although the syntax for this trigger is similar to a trigger for a DML statement, the ON clause works a little differently. To start, you can code the DATABASE keyword after the ON keyword to fire the trigger when the specified DDL actions are executed on the current database. Or, you can code the ALL SERVER keywords after the ON keyword to fire the trigger when the specified DDL actions are executed on any database on the current server. However, only a handful of DDL actions apply to a server-level trigger. As a result, a server-level trigger can only be fired by certain actions that apply to the server such as the CREATE DATABASE statement.

To specify the DDL actions for the trigger, you begin by coding the AFTER or INSTEAD OF keywords just as you do for any type of trigger. Then, you specify one or more DDL statements, separated by commas. In this figure, for example, the statement uses CREATE_TABLE and DROP_TABLE to specify that the trigger fires after a CREATE TABLE or DROP TABLE statement is executed.

Next, two variables are declared, @EventData with an xml data type and @EventType with a varchar data type. Then, the @EventData variable is filled using a special function called EVENTDATA. This function is available from within the body of a trigger for a DDL statement, and it returns an XML document that contains data about the event that caused the trigger to fire.

After that, the @EventType variable is filled by parsing the character data that's stored in the @EventData variable. This returns information about what type of DDL statement was executed. For now, don't worry if you don't understand the code that parses the @EventData variable. It should make more sense after you've read chapter 18. Finally, the code inserts the data stored in the @EventData and @EventType variables into the corresponding columns in the AuditDDL table.

This figure shows the data in the AuditDDL table after a user has created a table named VendorsTest. If you are using Management Studio and you click on the XML in the EventData column, Management Studio displays the XML for the event. This XML should be similar to the XML shown in this figure.

If you want to make it easier to access any of this data, you can store it in the columns of the AuditDDL table. To do that, create the necessary columns in the AuditDDL table. Then, use the techniques described in chapter 18 to parse the @EventData variable and store each piece of data in the appropriate column. For example, you might want to store the PostTime and UserName values in their own columns. That way, you could easily query the AuditDDL table to determine when the DDL statement was executed and who executed it.

A trigger that works with DDL statements

```
CREATE TRIGGER Database_CreateTable_DropTable
    ON DATABASE
    AFTER CREATE_TABLE, DROP_TABLE
AS
    DECLARE @EventData xml;
    DECLARE @EventType varchar(20);

    SELECT @EventData = EVENTDATA();
    SET @EventType =
        @EventData.value('(/EVENT_INSTANCE/EventType)[1]', 'varchar(100)');

    INSERT INTO AuditDDL (EventType, EventData)
        VALUES(@EventType, @EventData);
```

A CREATE TABLE statement that fires the trigger

```
CREATE TABLE VendorsTest (VendorID int, VendorName varchar(50));
```

The row inserted into the AuditDDL table

	EventType	EventData
1	CREATE_TABLE	<EVENT_INSTANCE><EventType>CREATE TABLE</EventType>

The XML saved in the EventData column

```
<EVENT_INSTANCE>
    <EventType>CREATE_TABLE</EventType>
    <PostTime>2020-02-05T12:38:23.147</PostTime>
    <SPID>54</SPID>
    <ServerName>MMA17\SQLEXPRESS</ServerName>
    <LoginName>murach\anne</LoginName>
    <UserName>dbo</UserName>
    <DatabaseName>AP</DatabaseName>
    <SchemaName>dbo</SchemaName>
    <ObjectName>VendorsTest</ObjectName>
    <ObjectType>TABLE</ObjectType>
    <TSQLCommand>
        <SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON" ANSI_PADDING="ON"
            QUOTED_IDENTIFIER="ON" ENCRYPTED="FALSE" />
        <CommandText>
            CREATE TABLE VendorsTest
                (VendorID int, VendorName varchar(50));
        </CommandText>
    </TSQLCommand>
</EVENT_INSTANCE>
```

Description

- In the ON clause, you can specify the DATABASE keyword to fire the trigger only for the current database, or you can specify the ALL SERVER keywords to fire the trigger for any database on the current server.
- To specify a DDL statement for a trigger, you can code the name of the DDL statement, replacing any spaces with underscores.
- The EVENTDATA function returns an XML document of the xml data type. If necessary, you can use the skills presented in chapter 18 to parse this document.

Figure 15-21 A trigger that works with DDL statements

How to delete or change a trigger

Figure 15-22 presents the syntax of the DROP TRIGGER and ALTER TRIGGER statements. DROP TRIGGER permanently deletes one or more triggers along with any security permissions associated with the trigger.

If you want to change the definition of a trigger without affecting permissions, you can use the ALTER TRIGGER statement. The statement shown in this figure, for example, modifies the trigger you saw in figure 15-17. This trigger now removes spaces from the beginning and end of the address columns in addition to converting the state code to upper case.

The syntax of the DROP TRIGGER statement

```
DROP TRIGGER trigger_name [, ...]
```

The syntax of the ALTER TRIGGER statement

```
ALTER TRIGGER trigger_name  
ON {table_name|view_name}  
[WITH [ENCRYPTION] [,] [EXECUTE AS clause]]  
{FOR|AFTER|INSTEAD OF} [INSERT] [,] [UPDATE] [,] [DELETE]  
AS sql_statements
```

A statement that modifies the trigger in figure 15-17

```
ALTER TRIGGER Vendors_INSERT_UPDATE  
ON Vendors  
AFTER INSERT, UPDATE  
AS  
    UPDATE Vendors  
    SET VendorState = UPPER(VendorState),  
        VendorAddress1 = TRIM(VendorAddress1),  
        VendorAddress2 = TRIM(VendorAddress2)  
    WHERE VendorID IN (SELECT VendorID FROM Inserted);
```

A statement that deletes the trigger

```
DROP TRIGGER Vendors_INSERT_UPDATE;
```

Description

- To delete a trigger from the database, use the DROP TRIGGER statement.
- To modify the definition of a trigger, you can locate the trigger in the Object Explorer of SQL Server Management Studio by expanding the table the trigger is associated with and then expanding the Triggers folder. Finally, you can right-click on the trigger and choose modify.
- To modify the definition of a DDL trigger, you can locate the trigger in the Object Explorer by expanding the Programmability folder and then expanding the Database Triggers folder. Finally, you can right-click on the trigger and choose modify.
- When you delete a trigger, any security permissions that are assigned to the trigger are also deleted. If that's not what you want, you can use the ALTER TRIGGER statement to modify the trigger and preserve the permissions.
- Unless a trigger and its corresponding table or view belong to the default schema, you must include the schema name on the DROP TRIGGER and ALTER TRIGGER statements.

Figure 15-22 How to delete or change a trigger

Perspective

In this chapter, you've learned how to create the three types of executable database objects supported by SQL Server using SQL statements. Stored procedures are the most flexible of the three because you can use them in so many different ways. You can code procedures to simultaneously simplify and restrict a user's access to the database, to verify data integrity, and to ease your own administrative tasks.

Although they're generally less flexible than stored procedures, functions and triggers are powerful objects. You can use them to solve problems that otherwise would be difficult or impossible to solve. In particular, table-valued functions are one of the most useful extensions provided by Transact-SQL because they behave like views but can accept parameters that can change the result set.

In addition to using SQL statements to work with stored procedures, functions, and triggers, you can use the Management Studio. You'll find all three of these object types in folders within the Programmability folder for a database. Then, you can add, modify, and delete objects using the menus that appear when you right-click on a folder or object. You can use this same technique to work with the user-defined table types you use with stored procedures.

Terms

stored procedure	required parameter
user-defined function (UDF)	optional parameter
trigger	passing parameters by position
sproc	passing parameters by name
call a procedure	return value
precompiled	data validation
execution plan	user-defined table type
recursive call	system stored procedure
recursion	scalar-valued function
temporary stored procedure	table-valued function
local procedure	simple table-valued function
global procedure	multi-statement table-valued function
parameter	invoke a function
input parameter	inline table-valued function
output parameter	fire a trigger

Exercises

1. Create a stored procedure named `spBalanceRange` that accepts three optional parameters. The procedure should return a result set consisting of `VendorName`, `InvoiceNumber`, and `Balance` for each invoice with a balance due, sorted with largest balance due first. The parameter `@VendorVar` is a mask that's used with a `LIKE` operator to filter by vendor name, as shown in figure 15-5. `@BalanceMin` and `@BalanceMax` are parameters used to specify the requested range of balances due. If called with no parameters or with a maximum value of 0, the procedure should return all invoices with a balance due.
 2. Code three calls to the procedure created in exercise 1:
 - (a) passed by position with `@VendorVar='M%`' and no balance range
 - (b) passed by name with `@VendorVar` omitted and a balance range from \$200 to \$1000
 - (c) passed by position with a balance due that's less than \$200 filtering for vendors whose names begin with C or F
 3. Create a stored procedure named `spDateRange` that accepts two parameters, `@DateMin` and `@DateMax`, with data type `varchar` and default value `null`. If called with no parameters or with null values, raise an error that describes the problem. If called with non-null values, validate the parameters. Test that the literal strings are valid dates and test that `@DateMin` is earlier than `@DateMax`. If the parameters are valid, return a result set that includes the `InvoiceNumber`, `InvoiceDate`, `InvoiceTotal`, and `Balance` for each invoice for which the `InvoiceDate` is within the date range, sorted with earliest invoice first.
 4. Code a call to the stored procedure created in exercise 3 that returns invoices with an `InvoiceDate` between October 10 and October 20, 2019. This call should also catch any errors that are raised by the procedure and print the error number and description.
 5. Create a scalar-valued function named `fnUnpaidInvoiceID` that returns the `InvoiceID` of the earliest invoice with an unpaid balance. Test the function in the following `SELECT` statement:

```
SELECT VendorName, InvoiceNumber, InvoiceDueDate,
       InvoiceTotal - CreditTotal - PaymentTotal AS Balance
  FROM Vendors JOIN Invoices
    ON Vendors.VendorID = Invoices.VendorID
   WHERE InvoiceID = dbo.fnUnpaidInvoiceID();
```
 6. Create a table-valued function named `fnDateRange`, similar to the stored procedure of exercise 3. The function requires two parameters of data type `date`. Don't validate the parameters. Return a result set that includes the `InvoiceNumber`, `InvoiceDate`, `InvoiceTotal`, and `Balance` for each invoice for which the `InvoiceDate` is within the date range. Invoke the function from within a `SELECT` statement to return those invoices with `InvoiceDate` between October 10 and October 20, 2019.

7. Use the function you created in exercise 6 in a SELECT statement that returns five columns: VendorName and the four columns returned by the function.
8. Create a trigger for the Invoices table that automatically inserts the vendor name and address for a paid invoice into a table named ShippingLabels. The trigger should fire any time the PaymentTotal column of the Invoices table is updated. The structure of the ShippingLabels table is as follows:

```
CREATE TABLE ShippingLabels
(VendorName      varchar(50),
 VendorAddress1  varchar(50),
 VendorAddress2  varchar(50),
 VendorCity      varchar(50),
 VendorState     char(2),
 VendorZipCode   varchar(20));
```

Use this UPDATE statement to test the trigger:

```
UPDATE Invoices
SET PaymentTotal = 67.92, PaymentDate = '2020-02-23'
WHERE InvoiceID = 100;
```

9. Write a trigger that prohibits duplicate values *except for nulls* in the NoDupName column of the following table:

```
CREATE TABLE TestUniqueNulls
(RowID      int IDENTITY NOT NULL,
 NoDupName  varchar(20)  NULL);
```

(Note that you can't do this by using a unique constraint because the constraint wouldn't allow duplicate null values.) If an INSERT or UPDATE statement creates a duplicate value, roll back the statement and return an error message.

Write a series of INSERT statements that tests that duplicate null values are allowed but duplicates of other values are not.

How to manage transactions and locking

If you've been working with a stand-alone copy of SQL Server, you've been the only user of your database. In the real world, though, a database is typically used by many users working simultaneously. Then, what happens when two users try to update the same data at the same time?

In this chapter, you'll learn how SQL Server manages concurrent changes. But first, you'll learn how to combine related SQL statements into a single unit, called a transaction. By learning these skills, you'll be able to write code that anticipates these conflicts.

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
Perspective	534

How to work with transactions

A *transaction* is a group of database operations that you combine into a single logical unit. By combining operations in this way, you can prevent certain kinds of database errors. In the topics that follow, you'll learn the SQL statements for managing transactions.

How transactions maintain data integrity

Figure 16-1 presents an example of three INSERT statements that are good candidates for a transaction. As you can see, the first INSERT statement adds a new invoice to the Invoices table. Next, a SET statement assigns the identity value for the newly inserted invoice to the @InvoiceID variable. Then, the last two INSERT statements insert rows into the InvoiceLineItems table that represent the two line items associated with the invoice.

What would happen if one or more of these INSERT statements failed? If the first statement failed, @InvoiceID wouldn't be assigned a valid value, so the last two insertions would also fail. However, if the first statement succeeded and one or both of the other INSERT statements failed, the Invoices and InvoiceLineItems tables wouldn't match. Specifically, the total of the InvoiceLineItemAmount columns in the InvoiceLineItems table wouldn't equal the InvoiceTotal column in the Invoices table, so the data would be invalid.

Now, suppose that these three INSERT statements were executed as part of the same transaction as illustrated in the second example in this figure. Here, you can see that a BEGIN TRAN statement is executed before the first INSERT statement. Then, after all three INSERT statements are executed, a COMMIT TRAN statement *commits* the changes to the database making them permanent. Because these statements are coded within a TRY block, however, the COMMIT TRAN statement is never executed if any of the INSERT statements fail. Instead, execution jumps into the CATCH block. This block executes a ROLLBACK TRAN statement to undo, or *rollback*, all of the changes made since the beginning of the transaction.

By grouping these SQL statements together in a single transaction, you can control whether and how changes are made to the database. Since all three INSERT statements must succeed for the transaction to be committed, a failure of any of the statements will cause the entire transaction to be rolled back. Note, however, that once you commit the transaction, you can't roll it back. Likewise, once you roll a transaction back, you can't commit it.

In this particular example, an error in one of the INSERT statements wouldn't be catastrophic. If a statement failed because you coded it incorrectly, you could easily correct the error by resubmitting the failed INSERT statement. If the failure was due to a system error such as a server crash, however, you wouldn't discover the error unless you looked for it after the server was restored.

Three INSERT statements that work with related data

```
DECLARE @InvoiceID int;
INSERT Invoices
    VALUES (34,'ZXA-080','2020-03-01',14092.59,0,0,3,'2020-03-31',NULL);
SET @InvoiceID = @@IDENTITY;
INSERT InvoiceLineItems VALUES (@InvoiceID,1,160,4447.23,'HW upgrade');
INSERT InvoiceLineItems VALUES (@InvoiceID,2,167,9645.36,'OS upgrade');
```

The same statements coded as a transaction

```
DECLARE @InvoiceID int;
BEGIN TRY
    BEGIN TRAN;
    INSERT Invoices
        VALUES (34,'ZXA-080','2020-03-01',14092.59,0,0,3,'2020-03-31',NULL);
    SET @InvoiceID = @@IDENTITY;
    INSERT InvoiceLineItems VALUES (@InvoiceID,1,160,4447.23,'HW upgrade');
    INSERT InvoiceLineItems VALUES (@InvoiceID,2,167,9645.36,'OS upgrade');
    COMMIT TRAN;
END TRY
BEGIN CATCH
    ROLLBACK TRAN;
END CATCH;
```

When to use explicit transactions

- When you code two or more action queries that affect related data
- When you update foreign key references
- When you move rows from one table to another table
- When you code a SELECT query followed by an action query and the values inserted in the action query are based on the results of the SELECT query
- When a failure of any set of SQL statements would violate data integrity

Description

- A *transaction* is a group of database operations that are combined into a logical unit. By default, each SQL statement is treated as a separate transaction. However, you can combine any number of SQL statements into a single transaction as shown above.
- When you *commit* a transaction, the operations performed by the SQL statements become a permanent part of the database. Until it's committed, you can undo all of the changes made to the database since the beginning of the transaction by *rolling back* the transaction.
- A transaction is either committed or rolled back in its entirety. Once you commit a transaction, it can't be rolled back.

For some systems, however, a violation of data integrity such as this one is critical. For instance, consider the classic example of a transfer between two accounts in a banking system. In that case, one update reduces the balance in the first account and another update increases the balance in the second account. If one of these updates fails, either the bank or the customer gets an unexpected windfall. Because an error like this could cause problems even during the short period of time it may take to fix it, these two updates should be coded as a transaction.

SQL statements for handling transactions

Figure 16-2 summarizes the SQL statements used to process transactions. As you can see, you can code either the TRAN or the TRANSACTION keyword in each of these statements, although TRAN is used more commonly. You can also omit this keyword entirely from the COMMIT and ROLLBACK statements. However, it's customary to include this keyword since it makes your code easier to read.

The BEGIN TRAN statement explicitly marks the starting point of a transaction. If you don't code this statement, SQL Server implicitly starts a new transaction for each SQL statement you code. If the statement succeeds, the implicit transaction is committed automatically. For this reason, this mode is called *autocommit* mode. Note that you can't use the COMMIT TRAN statement to commit an implicit transaction.

However, you can code a ROLLBACK TRAN statement to roll back an implicit transaction. You saw an example of that in the trigger in figure 15-20. That trigger rolled back an UPDATE statement if it caused the data in the Invoices and InvoiceLineItems tables to be inconsistent.

You can also use the SAVE TRAN statement to declare one or more *save points* within a transaction. Then, you can roll back part of a transaction by coding the save point name in the ROLLBACK TRAN statement. You'll learn more about how that works in a moment.

This figure presents another script that uses a transaction. This script deletes the invoices for a particular vendor and then deletes the vendor. Notice that the script tests the value of the @@ROWCOUNT system function after rows are deleted from the Invoices table to see if more than one invoice was deleted. If so, the transaction is rolled back, so the deletion from the Invoices table is undone. If only one invoice was deleted, however, the transaction is committed, so the deletion is made permanent.

Before I go on, you should realize that you can also name a transaction in the BEGIN TRAN statement, and you can refer to that name in the COMMIT TRAN and ROLLBACK TRAN statements. Since there's usually no reason to do that, however, I've omitted that option from the syntax shown in this figure and from the examples shown in this chapter.

Summary of the SQL statements for processing transactions

Statement	Description
<code>BEGIN {TRAN TRANSACTION}</code>	Marks the starting point of a transaction.
<code>SAVE {TRAN TRANSACTION} save_point</code>	Sets a new save point within a transaction.
<code>COMMIT [TRAN TRANSACTION]</code>	Marks the end of a transaction and makes the changes within the transaction a permanent part of the database.
<code>ROLLBACK [[TRAN TRANSACTION] [save_point]]</code>	Rolls back a transaction to the starting point or to the specified save point.

A script that performs a test before committing the transaction

```

BEGIN TRAN;

DELETE Invoices
WHERE VendorID = 34;

IF @@ROWCOUNT > 1
    BEGIN
        ROLLBACK TRAN;
        PRINT 'More invoices than expected. Deletions rolled back.';
    END;
ELSE
    BEGIN
        COMMIT TRAN;
        PRINT 'Deletions committed to the database.';
    END;

```

The response from the system

```

(3 rows affected)
More invoices than expected. Deletions rolled back.

```

Description

- Although you can omit the TRAN keyword from the COMMIT and ROLLBACK statements, it's generally included for readability.
- By default, SQL Server is in *autocommit mode*. Then, unless you explicitly start a transaction using the BEGIN TRAN statement, each statement is automatically treated as a separate transaction. If the statement causes an error, it's automatically rolled back. Otherwise, it's automatically committed.
- Even if you don't explicitly start a transaction, you can roll it back using the ROLLBACK TRAN statement. However, you can't explicitly commit an implicit transaction.
- When you use *save points*, you can roll a transaction back to the beginning or to a particular save point. See figure 16-4 for details on using save points.
- Although you can name a transaction in the BEGIN TRAN statement and you can refer to that name in the COMMIT TRAN and ROLLBACK TRAN statements, you're not likely to do that.

Figure 16-2 SQL statements for handling transactions

How to work with nested transactions

A *nested transaction* is a transaction that's coded within another transaction. In other words, a BEGIN TRAN statement is coded after another BEGIN TRAN statement but before the COMMIT TRAN or ROLLBACK TRAN statement that ends the first transaction. Since there are few problems that can only be solved using nested transactions, it's unlikely that you'll ever need to code them. However, you should understand how the COMMIT TRAN statement behaves when you code it within a nested transaction. Figure 16-3 presents a script that illustrates how this works.

This example uses the @@TRANCOUNT system function, which returns the number of explicit transactions that are active on the current connection. If you haven't coded a BEGIN TRAN statement, @@TRANCOUNT returns zero. Then, each BEGIN TRAN statement increments @@TRANCOUNT by one, so its value indicates how deeply you've nested the transactions.

If the current value of @@TRANCOUNT is one, the COMMIT TRAN statement closes the current transaction and commits the changes to the database as you've seen in the last two figures. But if @@TRANCOUNT is greater than one, COMMIT TRAN simply decrements @@TRANCOUNT by 1. In other words, within a nested transaction, the COMMIT TRAN statement doesn't commit a transaction.

This counterintuitive behavior is illustrated by the script in this figure. Here, the COMMIT TRAN statement that follows the DELETE statement that deletes all the rows in the Vendors table decrements @@TRANCOUNT, but doesn't commit the deletion. That's because this COMMIT TRAN statement is coded within a nested transaction.

On the other hand, the ROLLBACK TRAN statement always rolls back all of the uncommitted statements, whether or not they're coded within a nested transaction. In this script, for example, the ROLLBACK TRAN statement rolls back both DELETE statements. As you can see from the results of the last five statements in this script, neither DELETE statement was committed.

A script with nested transactions

```

BEGIN TRAN;
PRINT 'First Tran  @@TRANCOUNT: ' + CONVERT(varchar,@@TRANCOUNT);
DELETE Invoices;
BEGIN TRAN;
PRINT 'Second Tran @@TRANCOUNT: ' + CONVERT(varchar,@@TRANCOUNT);
DELETE Vendors;
COMMIT TRAN;           -- This COMMIT decrements @@TRANCOUNT.
                      -- It doesn't commit 'DELETE Vendors'.
PRINT 'COMMIT          @@TRANCOUNT: ' + CONVERT(varchar,@@TRANCOUNT);
ROLLBACK TRAN;
PRINT 'ROLLBACK        @@TRANCOUNT: ' + CONVERT(varchar,@@TRANCOUNT);

PRINT '';
DECLARE @VendorsCount int, @InvoicesCount int;
SELECT @VendorsCount = COUNT (*) FROM Vendors;
SELECT @InvoicesCount = COUNT (*) FROM Invoices;
PRINT 'Vendors Count: ' + CONVERT (varchar , @VendorsCount);
PRINT 'Invoices Count: ' + CONVERT (varchar , @InvoicesCount);

```

The response from the system

```

First Tran @@TRANCOUNT: 1

(114 rows affected)
Second Tran @@TRANCOUNT: 2

(122 rows affected)
COMMIT      @@TRANCOUNT: 1
ROLLBACK    @@TRANCOUNT: 0

Vendors count: 122
Invoices count: 114

```

Description

- You can *nest* transactions by coding nested BEGIN TRAN statements. Each time this statement is executed, it increments the @@TRANCOUNT system function by 1. Then, you can query this function to determine how many levels deep the transactions are nested.
- If you execute a COMMIT TRAN statement when @@TRANCOUNT is equal to 1, all of the changes made to the database during the transaction are committed and @@TRANCOUNT is set to zero. If @@TRANCOUNT is greater than 1, however, the changes aren't committed. Instead, @@TRANCOUNT is simply decremented by 1.
- The ROLLBACK TRAN statement rolls back all active transactions regardless of the nesting level where it's coded. It also sets the value of @@TRANCOUNT back to 0.
- Since there are few programming problems that you can only solve using nested transactions, you probably won't use them often.

Figure 16-3 How to work with nested transactions

How to work with save points

You can create save points within a transaction by coding the `SAVE TRAN` statement. In that case, you can roll back the transaction to that particular point by coding the save point name in the `ROLLBACK TRAN` statement. Figure 16-4 presents a script that shows how this works.

First, this script creates a temporary table named `#VendorCopy` that contains a copy of the vendor IDs and names for the vendors in the `Vendors` table with vendor IDs 1, 2, 3, and 4. After beginning a transaction, the script deletes a row and then sets a save point named `Vendor1`. Then the script deletes a second row, sets another save point named `Vendor2`, and deletes a third row. The result of the first `SELECT` statement that follows illustrates that only one row is left in the `#VendorCopy` table.

Next, a `ROLLBACK TRAN` statement rolls back the transaction to the `Vendor2` save point. This rolls back the third delete, as illustrated by the second `SELECT` statement in this figure. The next `ROLLBACK TRAN` statement rolls the transaction back to the `Vendor1` save point, which rolls back the second delete. At that point, the `#VendorCopy` table contains three rows, as illustrated by the third `SELECT` statement. Finally, a `COMMIT TRAN` statement commits the transaction. Since the only statement that hasn't already been rolled back is the statement that deleted the first row, this row is deleted permanently. The last `SELECT` statement illustrates the final result of this code.

You should note that if you don't code a save point name in the `ROLLBACK TRAN` statement, it ignores any save points and rolls back the entire transaction. In addition, you should notice that you can't code a save point name in a `COMMIT TRAN` statement. This means that you can't partially commit a transaction. Instead, a `COMMIT TRAN` statement ignores save points completely and commits the entire transaction.

As with nested transactions, you'll probably never need to use save points since there are few problems that can be solved by using them. Unlike the way that nested transactions work, however, save points work in an intuitive way, so coding them is less confusing.

A transaction with two save points

```

IF OBJECT_ID('tempdb..#VendorCopy') IS NOT NULL
    DROP TABLE tempdb.. #VendorCopy;
SELECT VendorID, VendorName
INTO #VendorCopy
FROM Vendors
WHERE VendorID < 5;
BEGIN TRAN;
    DELETE #VendorCopy WHERE VendorID = 1;
    SAVE TRAN Vendor1;
    DELETE #VendorCopy WHERE VendorID = 2;
    SAVE TRAN Vendor2;
    DELETE #VendorCopy WHERE VendorID = 3;
    SELECT * FROM #VendorCopy;
    ROLLBACK TRAN Vendor2;
    SELECT * FROM #VendorCopy;
    ROLLBACK TRAN Vendor1;
    SELECT * FROM #VendorCopy;
COMMIT TRAN;
SELECT * FROM #VendorCopy;

```

The response from the system

VendorID	VendorName
1	4 Jobtrak

VendorID	VendorName
1	3 Register of Copyrights
2	4 Jobtrak

VendorID	VendorName
1	2 National Information Data Ctr
2	3 Register of Copyrights
3	4 Jobtrak

VendorID	VendorName
1	2 National Information Data Ctr
2	3 Register of Copyrights
3	4 Jobtrak

Description

- You can partially roll back a transaction if you use save points. If you code a save point name in the ROLLBACK TRAN statement, the system rolls back all of the statements to that save point.
- If you don't code a save point name, the ROLLBACK TRAN statement rolls back the entire transaction.
- Since you can't code a save point name in a COMMIT TRAN statement, the system always commits the entire transaction.
- As with nested transactions, there are few practical programming problems that you can solve using save points.

Figure 16-4 How to work with save points

An introduction to concurrency and locking

When two or more users have access to the same database, it's possible for them to be working with the same data at the same time. This is called *concurrency*. Concurrency isn't a problem when two users retrieve the same data at the same time. If they then try to update that data, however, that can be a problem. In the topics that follow, you'll learn more about concurrency and how SQL Server uses locking to prevent concurrency problems. You'll also learn how you can control the types of problems that are allowed.

How concurrency and locking are related

Figure 16-5 presents two transactions that select and then update data from the same row in the same table. If these two transactions are submitted at the same time, the one that executes first will be overwritten by the one that executes second. Since this means that one of the two updates is lost, this is known as a *lost update*.

This figure shows the result if the update operation in transaction A is executed first, in which case its update is lost when the update in transaction B is executed. Because transaction A is unaware that its update has been lost, however, this can leave the data in an unpredictable state that affects the integrity of the data. For the AP database, it's unlikely that a lost update will adversely affect the system. For some database systems, however, this sort of unpredictability can be disastrous.

If your database has a relatively small number of users, the likelihood of concurrency problems is low. However, the larger the system, the greater the number of users and transactions. For a large system, then, you should expect concurrency, and therefore concurrency problems, to occur more frequently.

One way to avoid concurrency problems is to use *locking*. By holding a lock on the data, the transaction prevents others from using that data. Then, after the transaction releases the lock, the next transaction can work with that data.

Since SQL Server automatically enables and manages locking, it may prevent most of the concurrency problems on your system. If the number of users of your system grows, however, you may find that the default locking mechanism is insufficient. In that case, you may need to override the default locking behavior. You'll learn how to do that in a moment. But first, you need to understand the four concurrency problems that locks can prevent.

Two transactions that retrieve and then modify the data in the same row

Transaction A

```
BEGIN TRAN;
DECLARE @InvoiceTotal money, @PaymentTotal money, @CreditTotal money;
SELECT @InvoiceTotal = InvoiceTotal, @CreditTotal = CreditTotal,
       @PaymentTotal = PaymentTotal FROM Invoices WHERE InvoiceID = 112;
UPDATE Invoices
    SET InvoiceTotal = @InvoiceTotal, CreditTotal = @CreditTotal + 317.40,
        PaymentTotal = @PaymentTotal WHERE InvoiceID = 112;
COMMIT TRAN;
```

Transaction B

```
BEGIN TRAN;
DECLARE @InvoiceTotal money, @PaymentTotal money, @CreditTotal money;
SELECT @InvoiceTotal = InvoiceTotal, @CreditTotal = CreditTotal,
       @PaymentTotal = PaymentTotal FROM Invoices WHERE InvoiceID = 112;
UPDATE Invoices
    SET InvoiceTotal = @InvoiceTotal, CreditTotal = @CreditTotal,
        PaymentTotal = @InvoiceTotal - @CreditTotal,
        PaymentDate = GetDate() WHERE InvoiceID = 112;
COMMIT TRAN;
```

The initial values for the row

	InvoiceTotal	CreditTotal	PaymentTotal	PaymentDate
1	10976.06	0.00	0.00	NULL

The values after transaction A executes

	InvoiceTotal	CreditTotal	PaymentTotal	PaymentDate
1	10976.06	317.40	0.00	NULL

The values after transaction B executes, losing transaction A's updates

	InvoiceTotal	CreditTotal	PaymentTotal	PaymentDate
1	10976.06	317.40	10658.66	2020-02-05

Description

- Concurrency is the ability of a system to support two or more transactions working with the same data at the same time.
- Because small systems have few users, concurrency isn't generally a problem on these systems. On large systems with many users and many transactions, however, you may need to account for concurrency in your SQL code.
- Concurrency is a problem only when the data is being modified. When two or more transactions simply read the same data, the transactions don't affect each other.
- You can avoid some database concurrency problems by using *locks*, which delay the execution of a transaction if it conflicts with a transaction that's already running. Then, the second transaction can't use the data until the first transaction releases the lock.
- Although SQL Server automatically enforces locking, you can write more efficient code by understanding and customizing locking in your programs.

Figure 16-5 An introduction to concurrency and locking

The four concurrency problems that locks can prevent

Figure 16-6 describes the four types of concurrency problems. You've already learned about the first problem: lost updates. In a moment, you'll see how locking can be used to prevent all four of these problems.

Like lost updates, the other three problems may not adversely affect a database. That depends on the nature of the data. In fact, for many systems, these problems happen infrequently. Then, when they do occur, they can be corrected by simply resubmitting the query that caused the problem. On some database systems, however, these problems can affect data integrity in a serious way.

Although locks can prevent the problems listed in this figure, SQL Server's default locking behavior won't. If your transaction could adversely affect data integrity on your system, then, you should consider changing the default locking behavior by setting the transaction isolation level.

The four types of concurrency problems

Problem	Description
Lost updates	Occur when two transactions select the same row and then update the row based on the values originally selected. Since each transaction is unaware of the other, the later update overwrites the earlier update.
Dirty reads (uncommitted dependencies)	Occur when a transaction selects data that isn't committed by another transaction. For example, transaction A changes a row. Transaction B then selects the changed row before transaction A commits the change. If transaction A then rolls back the change, transaction B has selected a row that doesn't exist in the database.
Nonrepeatable reads (inconsistent analysis)	Occur when two SELECT statements of the same data result in different values because another transaction has updated the data in the time between the two statements. For example, transaction A selects a row. Transaction B then updates the row. When transaction A selects the same row again, the data is different.
Phantom reads	Occur when you perform an update or delete on a set of rows when another transaction is performing an insert or delete that affects one or more rows in that same set of rows. For example, transaction A updates the payment total for each invoice that has a balance due. Transaction B inserts a new, unpaid invoice while transaction A is still running. After transaction A finishes, there is still an invoice with a balance due.

Description

- In a large system with many users, you should expect for these kinds of problems to occur. In general, you don't need to take any action except to anticipate the problem. In many cases, if the query is resubmitted, the problem goes away.
- On some systems, if two transactions overwrite each other, the validity of the database is compromised and resubmitting one of the transactions will not eliminate the problem. If you're working on such a system, you must anticipate these concurrency problems and account for them in your code.
- You should consider these locking problems as you write your code. If one of these problems would affect data integrity, you can change the default locking behavior by setting the transaction isolation level as shown in the next figure.

Figure 16-6 The four concurrency problems that locks can prevent

How to set the transaction isolation level

The simplest way to prevent concurrency problems is to reduce concurrency. To do that, you need to change SQL Server's default locking behavior. Figure 16-7 shows you how.

To change the default locking behavior, you use the `SET TRANSACTION ISOLATION LEVEL` statement to set the *transaction isolation level* for the current session. As you can see, this statement accepts one of five options. The table in this figure lists which of the four concurrency problems each option will prevent or allow. For example, if you code the `SERIALIZABLE` option, all four concurrency problems will be prevented.

When you set the isolation level to `SERIALIZABLE`, each transaction is completely isolated from every other transaction and concurrency is severely restricted. The server does this by locking each resource, preventing other transactions from accessing it. Since each transaction must wait for the previous transaction to commit, the transactions are executed serially, one after another.

Since the `SERIALIZABLE` isolation level eliminates all possible concurrency problems, you may think that this is the best option. However, this option requires more server overhead to manage all of the locks. In addition, access time for each transaction is increased, since only one transaction can work with the data at a time. For most systems, this will actually eliminate few concurrency problems but will cause severe performance problems.

The lowest isolation level is `READ UNCOMMITTED`, which allows all four of the concurrency problems to occur. It does this by performing `SELECT` queries without setting any locks and without honoring any existing locks. Since this means that your `SELECT` statements will always execute immediately, this setting provides the best performance. Since other transactions can retrieve and modify the same data, however, this setting can't prevent concurrency problems.

The default isolation level, `READ COMMITTED`, is acceptable for most applications. However, the only concurrency problem it prevents is dirty reads. Although it can prevent some lost updates, it doesn't prevent them all.

The `REPEATABLE READ` level allows more concurrency than the `SERIALIZABLE` level but less than the `READ COMMITTED` level. As you might expect, then, it results in faster performance than `SERIALIZABLE` and permits fewer concurrency problems than `READ COMMITTED`.

The `SNAPSHOT` isolation level uses a feature called *row versioning*. With row versioning, any data that's retrieved by a transaction that uses `SNAPSHOT` isolation is consistent with the data that existed at the start of the transaction. To accomplish that, SQL Server stores the original version of a row in the `tempdb` database each time it's modified.

You can also use row versioning with the `READ COMMITTED` isolation level. Then, each statement within a transaction works with a snapshot of the data as it existed at the start of the statement.

When you use row versioning, locks are not required for read operations, which improves concurrency. However, the need to maintain row versions requires additional resources and can degrade performance. In most cases, then, you'll use row versioning only when data consistency is imperative.

The syntax of the SET TRANSACTION ISOLATION LEVEL statement

```
SET TRANSACTION ISOLATION LEVEL
    {READ UNCOMMITTED|READ COMMITTED|REPEATABLE READ|SNAPSHOT|SERIALIZABLE}
```

The concurrency problems prevented by each transaction isolation level

Isolation level	Dirty reads	Lost updates	Nonrepeatable reads	Phantom reads
READ UNCOMMITTED	Allows	Allows	Allows	Allows
READ COMMITTED	Prevents	Allows	Allows	Allows
REPEATABLE READ	Prevents	Prevents	Prevents	Allows
SNAPSHOT	Prevents	Prevents	Prevents	Prevents
SERIALIZABLE	Prevents	Prevents	Prevents	Prevents

Description

- Since SQL Server manages locking automatically, you can't control every aspect of locking for your transactions. However, you can set the isolation level in your code.
- The *transaction isolation level* controls the degree to which transactions are isolated from one another. The server isolates transactions by using more restrictive locking behavior. If you isolate your transactions from other transactions, concurrency problems are reduced or eliminated.
- You specify the transaction isolation level by changing the ISOLATION LEVEL session setting. The default transaction isolation level is READ COMMITTED. At this level, some lost updates can occur, but this is acceptable for most transactions.
- The READ UNCOMMITTED isolation level doesn't set any locks and ignores locks that are already held. Setting this level results in the highest possible performance for your query, but at the risk of every kind of concurrency problem. For this reason, you should only use this level for data that is rarely updated.
- The REPEATABLE READ level places locks on all data that's used in a transaction, preventing other users from updating that data. However, this isolation level still allows inserts, so phantom reads can occur.
- The SNAPSHOT level uses *row versioning* rather than locks to provide read consistency. To use this level, you use the ALTER DATABASE statement to set the ALLOW_SNAPSHOT_ISOLATION option on.
- The SERIALIZABLE level places a lock on all data that's used in a transaction. Since each transaction must wait for the previous transaction to commit, the transactions are handled in sequence. This is the most restrictive of the five isolation levels.
- You can also use row versioning with the READ COMMITTED isolation level. To do that, you must set the READ_COMMITTED_SNAPSHOT database option on.
- With row versioning, each time a transaction modifies a row, SQL Server stores an image of the row as it existed before the modification. That way, read operations that use row versioning retrieve the row as it existed at the start of the transaction (SNAPSHOT) or statement (READ COMMITTED).

Figure 16-7 How to set the transaction isolation level

How SQL Server manages locking

SQL Server automatically manages locking by setting a lock on the data used by each transaction. By understanding how this process works, you'll be able to write better SQL code.

Lockable resources and lock escalation

A transaction like the one shown in figure 16-5 affects only one row in one table. By contrast, a transaction that uses DDL statements to change the design of a database can affect every object in the database. To accommodate these differences, SQL Server can lock data resources at ten different levels. These levels are presented in figure 16-8.

A resource's *granularity* refers to the relative amount of data it includes. For example, a row is a *fine-grain resource* and has higher granularity than a database, which is a *coarse-grain resource*. As you can see, the resources listed in this figure are listed in order of increasing granularity.

The SQL Server *lock manager* automatically assigns locks for each transaction. Since a coarse-grain lock will lock out more transactions than a fine-grain lock, the lock manager always tries to lock resources at the highest possible granularity. However, it takes greater server resources to maintain several fine-grain locks compared to one coarse-grain lock. For this reason, the lock manager detects when several fine-grain locks apply to a single coarse-grain resource. Then it converts, or *escalates*, the fine-grain locks to a single coarse-grain lock.

The ten levels of lockable resources

Granularity	Resource	Description
Coarse	Database	Locks an entire database.
	Allocation unit	Locks a collection of pages that contains a particular type of data.
	Metadata	Locks the data in the system catalog.
	File	Locks an entire database file.
	Table	Locks an entire table, including indexes.
	Heap or B-tree	Locks the index pages (B-tree) for a table with a clustered index or the data pages (heap) for a table with no clustered index.
	Extent	Locks a contiguous group of eight pages.
	Page	Locks one page (8 KB) of data.
	Key	Locks a key or range of keys in an index.
Fine	Row	Locks a single row within a table.

Description

- SQL Server can lock data at various levels, known as *lockable resources*. The ten levels form a hierarchy based on *granularity*, which refers to the amount of data the resource encompasses. A resource that encompasses more data than another resource is said to be less granular, or *coarser*, than the other resource.
- A *coarse-grain lock* affects more data than a *fine-grain lock*. For this reason, more transactions are locked out when the lock is less granular. Since this slows database performance, the server assigns locks of the finest possible granularity.
- Locking is automatically enabled and controlled by a SQL Server application called the *lock manager*. This program generates locking events and handles the setting and releasing of locks.
- Maintaining several fine-grain locks requires greater server resources than maintaining one coarse-grain lock. For this reason, the lock manager will automatically convert multiple fine-grain locks on the same resource into a single coarse-grain lock. This is known as *lock escalation*.

Figure 16-8 Lockable resources and lock escalation

Lock modes and lock promotion

In addition to assigning a resource level, the lock manager also assigns a *lock mode* to your transaction. Figure 16-9 presents the most common lock modes used by SQL Server. Although nine different lock modes are listed, each mode can be categorized as either a shared lock or an exclusive lock.

A *shared lock* doesn't prevent other shared locks from being granted on the same resource. For example, if you submit the query

```
SELECT * FROM Invoices;
```

the lock manager grants your transaction a Shared (S) lock on the Invoices table. If, while your query is executing, another user submits a query on the same table, your lock doesn't prevent the lock manager from granting a second S lock on the same table.

An *exclusive lock* on a resource, however, is granted exclusively to a single transaction. If another transaction requests a lock on the same resource, it must wait until the transaction that holds the exclusive lock has finished and its lock is released. If you submit an INSERT statement against the Invoices table, for example, the lock manager requests an Exclusive (X) lock on the Invoices table. If no other transaction has an exclusive lock on that table, the lock is granted.

While that transaction holds that lock, no other transaction can be granted a lock.

A single transaction can include various SQL statements that each require a different lock mode. In that case, a shared lock may need to be *promoted* to an exclusive lock. If, while the transaction is still executing, an exclusive lock can't be acquired, the transaction must wait until the lock is available. If the lock never becomes available, the transaction can never commit.

To prevent this problem, an Update (U) lock is assigned for some transactions. For example, consider the locks needed for an UPDATE query. First, the query must determine which row or rows are being updated based on the WHERE clause. For this part of the query, only a shared lock is needed. Then, when the actual update takes place, the lock must be promoted to an exclusive lock. Since this kind of lock promotion occurs with virtually every action query, the lock manager first assigns a U lock, which prevents another transaction from gaining a shared lock.

The Schema lock modes place a lock on a table's design. For this reason, they can't be placed at resource levels other than the table level. Interestingly, these lock modes represent both the least restrictive and the most restrictive mode. A Schema Stability (Sch-S) lock is placed when a query is compiling to prevent changes to the table's design. A Schema Modification (Sch-M) lock is placed when a query includes DDL statements that modify a table's design.

If another transaction requests a lock on the same resource but at a lower granularity, your finer-grain lock must still be honored. In other words, if your transaction holds an X lock on a page of data, you wouldn't want the lock manager to grant another transaction an X lock on the entire table. To manage this, the three intent lock modes are used as placeholders for locks on finer-grained resources.

Common SQL Server lock modes

Category	Lock mode	What the lock owner can do
Shared	Schema Stability (Sch-S)	Compile a query
	Intent Shared (IS)	Read but not change data
	Shared (S)	Read but not change data
	Update (U)	Read but not change data until promoted to an Exclusive (X) lock
Exclusive	Shared with Intent Exclusive (SIX)	Read and change data
	Intent Exclusive (IX)	Read and change data
	Exclusive (X)	Read and change data
	Bulk Update (BU)	Bulk-copy data into a table
	Schema Modification (Sch-M)	Modify the database schema

Description

- SQL Server automatically determines the appropriate *lock mode* for your transaction. In general, retrieval operations acquire *shared locks*, and update operations acquire *exclusive locks*. As a single transaction is being processed, its lock may have to be converted, or *promoted*, from one lock mode to a more exclusive lock mode.
- An Update (U) lock is acquired during the first part of an update, when the data is being read. Later, if the data is changed, the Update lock is promoted to an Exclusive (X) lock. This can prevent a common locking problem called a deadlock.
- An *intent lock* indicates that SQL Server intends to acquire a shared lock or an exclusive lock on a finer-grain resource. For example, an Intent Shared (IS) lock acquired at the table level means that the transaction intends to acquire shared locks on pages or rows within that table. This prevents another transaction from acquiring an exclusive lock on the table containing that page or row.
- *Schema locks* are placed on a table's design. Schema Modification (Sch-M) locks are acquired when the design is being changed with a DDL statement. Schema Stability (Sch-S) locks are acquired when compiling a query to prevent a schema change while the query is compiling.
- The Bulk Update (BU) lock mode is acquired for the BULK INSERT statement and by the bulk copy program (bcp). Since these operations are typically done by DBAs, neither is presented in this book.

Figure 16-9 Lock modes and lock promotion

The three *intent locks* differ based on the portion of the resource and the type of lock that the transaction intends to acquire. An Intent Shared (IS) lock indicates that the transaction intends to acquire a shared lock on some, but not all, of the finer-grained resource. Likewise, an Intent Exclusive (IX) lock indicates an intent to acquire an exclusive lock on some, but not all, of the resource. Finally, a Shared with Intent Exclusive (SIX) lock indicates an intent to acquire both an exclusive lock on some of the resource and a shared lock on the entire resource.

The Bulk Update (BU) lock mode is used exclusively for copying large amounts of data in bulk into a database using either the BULK INSERT statement or the bulk copy program. Since bulk copies are usually done by DBAs to create databases based on other sources, they're not presented in this book.

Lock mode compatibility

Figure 16-10 presents a table that shows the compatibility between the different lock modes. When a transaction tries to acquire a lock on a resource, the lock manager must first determine whether another transaction already holds a lock on that resource. If a lock is already in place, the lock manager will grant the new lock only if it's compatible with the current lock. Otherwise, the transaction will have to wait.

For example, if a transaction currently holds a U lock on a table and another transaction requests a U lock on the same table, the lock manager doesn't grant the second transaction's request. Instead, the second transaction must wait until the first transaction commits and releases its lock.

As you can see, Sch-S lock mode is compatible with every other lock mode except Sch-M. For this reason, the only lock that can delay the compilation of a query is the lock placed by a DDL statement that's changing the table's design.

Notice that the IS and S locks are compatible. This means that any number of SELECT queries can execute concurrently. All of the other locks, however, are incompatible to some extent. That's because each of these other modes indicates that data is already being modified by a current transaction.

Although the intent locks are similar to the standard shared and exclusive locks, they result in improved performance. That's because when the lock manager grants an intent lock, it locks a resource at a higher level than it would if it granted shared or exclusive locks. In other words, it grants a more coarse-grained lock. Then, to determine if a resource is already locked, the lock manager needs to look only at the coarse-grained resource rather than every fine-grained resource it contains.

Compatibility between lock modes

Current lock mode	Sch-S	Requested lock mode							
		IS	S	U	SIX	IX	X	BU	Sch-M
Schema Stability	Sch-S	✓	✓	✓	✓	✓	✓	✓	
Intent Shared	IS	✓	✓	✓	✓	✓	✓		
Shared	S	✓	✓	✓	✓				
Update	U	✓	✓	✓					
Shared w/Intent Exclusive	SIX	✓	✓						
Intent Exclusive	IX	✓	✓				✓		
Exclusive	X	✓							
Bulk Update	BU	✓							
Schema Modification	Sch-M								

Description

- If a resource is already locked by a transaction, a request by another transaction to acquire a lock on the same resource will be granted or denied depending on the compatibility of the two lock modes.
- For example, if a transaction has a Shared (S) lock on a table and another transaction requests an Exclusive (X) lock on the same table, the lock isn't granted. The second transaction must wait until the first transaction releases its lock.
- Intent locks can help improve performance since the server only needs to examine the high-level locks rather than examining every low-level lock.

Figure 16-10 Lock mode compatibility

How to prevent deadlocks

A *deadlock* occurs when two transactions are simultaneously holding and requesting a lock on each other's resource. Since deadlocks can occur more frequently at higher isolation levels, you need to understand how they come about and how you can prevent them.

Two transactions that deadlock

Figure 16-11 presents two transactions that are executed simultaneously. As you can see, transaction A queries the InvoiceLineItems table to determine the sum of all line item amounts for a specific invoice. Then, the WAITFOR DELAY statement causes the transaction to wait five seconds before continuing. (This statement is included only so you can actually cause the deadlock to occur.) Next, an UPDATE statement updates the InvoiceTotal column in the Invoices table with the value retrieved by the SELECT statement.

When this transaction executes, it requests several different locks. In particular, when the SELECT statement is executed, it requests an S lock on one page of the InvoiceLineItems table. And when the UPDATE statement is executed, it requests an X lock on a page of the Invoices table.

Now take a look at transaction B, which queries the Invoices table and then updates the InvoiceLineItems table for the same invoice as transaction A. In this case, the transaction requests an S lock on the Invoices table when the SELECT statement is executed, and it requests an X lock on the InvoiceLineItems table when the UPDATE statement is executed. Because transaction A has an S lock on the InvoiceLineItems table, however, the X lock isn't granted. Similarly, the X lock on the Invoices table isn't granted to transaction A because transaction B has an S lock on it. Because neither UPDATE can execute, neither transaction can commit and neither can release the resource needed by the other. In other words, the two transactions are deadlocked.

SQL Server automatically detects deadlocks and keeps them from tying up the system. It does this by selecting one of the transactions as the *deadlock victim*, which is rolled back and receives an error message. The other transaction runs to completion and commits. In the example in this figure, transaction B is the deadlock victim, as you can see by the system response.

Note that these two transactions will deadlock only if you set the transaction isolation level to REPEATABLE READ or SERIALIZABLE. Otherwise, the S lock acquired by each transaction is released after the SELECT statement completes. Since this doesn't prevent the other transaction from acquiring an X lock, each transaction can commit but causes the other to suffer from a dirty read.

Two transactions that deadlock

```
A SET TRANSACTION ISOLATION LEVEL
    REPEATABLE READ;

DECLARE @InvoiceTotal money;

BEGIN TRAN;
    SELECT @InvoiceTotal =
        SUM(InvoiceLineItemAmount)
    FROM InvoiceLineItems
    WHERE InvoiceID = 101;

WAITFOR DELAY '00:00:05';

UPDATE Invoices
SET InvoiceTotal =
    @InvoiceTotal
WHERE InvoiceID = 101;

COMMIT TRAN;
```

```
B SET TRANSACTION ISOLATION LEVEL
    REPEATABLE READ;

DECLARE @InvoiceTotal money;

BEGIN TRAN;
    SELECT @InvoiceTotal =
        InvoiceTotal
    FROM Invoices
    WHERE InvoiceID = 101;

UPDATE InvoiceLineItems
SET InvoiceLineItemAmount =
    @InvoiceTotal
WHERE InvoiceID = 101 AND
    InvoiceSequence = 1;

COMMIT TRAN;
```

The response from the system

```
(1 row(s) affected)
```

```
Msg 1205, Level 13, State 51, Line
11
Transaction (Process ID 53) was
deadlocked on lock resources with
another process and has been chosen
as the deadlock victim. Rerun the
transaction.
```

How the deadlock occurs

1. Transaction A requests and acquires a shared lock on the InvoiceLineItems table.
2. Transaction B requests and acquires a shared lock on the Invoices table.
3. Transaction A tries to acquire an exclusive lock on the Invoices table to perform the update. Since transaction B already holds a shared lock on this table, transaction A must wait for the exclusive lock.
4. Transaction B tries to acquire an exclusive lock on the InvoiceLineItems table, but must wait because transaction A holds a shared lock on that table.

Description

- A *deadlock* occurs when neither of two transactions can be committed because they each have a lock on a resource needed by the other.
- SQL Server automatically detects deadlocks and allows one of the transactions to commit. The other transaction is rolled back and raises error number 1205. This transaction is known as the *deadlock victim*.

Note

- To test this example, you must execute transaction A first and then execute transaction B within five seconds.

Figure 16-11 Two transactions that deadlock

Coding techniques that prevent deadlocks

Deadlocks slow system performance and cause transactions to become deadlock victims. For these reasons, you should try to avoid deadlocks as much as possible. Figure 16-12 presents a summary of the techniques you can use to do that.

First, you shouldn't leave transactions open any longer than is necessary. That's because the longer a transaction remains open and uncommitted, the more likely it is that another transaction will need to work with that same resource. Second, you shouldn't use a higher isolation level than you need. That's because the higher you set the isolation level, the more likely it is that two transactions will be unable to work concurrently on the same resource. Third, you should schedule transactions that modify a large number of rows to run when no other transactions, or only a small number of other transactions, will be running. That way, it's less likely that the transactions will try to change the same rows at the same time.

Finally, you should consider how a program you code could cause a deadlock. To illustrate, consider the UPDATE statements shown in this figure that transfer money between two accounts. The first example transfers money from a savings to a checking account. Notice that the savings account is updated first. The second example transfers money from a checking to a savings account. In this example, the checking account is updated first, which could cause a deadlock if the first transaction already has an X lock on the data. To prevent this situation, you should always update the same account first, regardless of which is being debited and which is being credited. This is illustrated by the third example in this figure.

Don't allow transactions to remain open for very long

- Keep transactions short.
- Keep SELECT statements outside of the transaction except when absolutely necessary.
- Never code requests for user input during an open transaction.

Use the lowest possible transaction isolation level

- The default level of READ COMMITTED is almost always sufficient.
- Reserve the use of higher levels for short transactions that make changes to data where integrity is vital.

Make large changes when you can be assured of nearly exclusive access

- If you need to change millions of rows in an active table, don't do so during hours of peak usage.
- If possible, give yourself exclusive access to the database before making large changes.

Consider locking when coding your transactions

- If you need to code two or more transactions that update the same resources, code the updates in the same order in each transaction.

UPDATE statements that transfer money between two accounts

From savings to checking

```
UPDATE Savings SET Balance = Balance - @TransferAmt;  
UPDATE Checking SET Balance = Balance + @TransferAmt;
```

From checking to savings

```
UPDATE Checking SET Balance = Balance - @TransferAmt;  
UPDATE Savings SET Balance = Balance + @TransferAmt;
```

From checking to savings in reverse order to prevent deadlocks

```
UPDATE Savings SET Balance = Balance + @TransferAmt;  
UPDATE Checking SET Balance = Balance - @TransferAmt;
```

Figure 16-12 Coding techniques that prevent deadlocks

Perspective

In this chapter, you've learned the ways that SQL Server protects your data from the problems that can occur on a real-world system. Since the failure of one or more related SQL statements can violate data integrity, you learned how to prevent these problems by grouping the statements into transactions. Since multiple transactions can simultaneously modify the same data, you learned how to prevent concurrency problems by setting the transaction isolation level to change the default locking behavior. And since changing the isolation level can increase the chances of deadlocks, you learned defensive programming techniques to prevent deadlocks.

Terms

transaction	row versioning
commit a transaction	granularity
roll back a transaction	fine-grain lock
autocommit mode	coarse-grain lock
nested transactions	lock manager
save point	lock escalation
concurrency	lock mode
locking	shared lock
lost update	exclusive lock
dirty read	intent lock
nonrepeatable read	schema lock
phantom read	lock promotion
transaction isolation level	deadlock
lockable resource	deadlock victim

Exercises

1. Write a set of action queries coded as a transaction to reflect the following change: United Parcel Service has been purchased by Federal Express Corporation and the new company is named FedUP. Rename one of the vendors and delete the other after updating the VendorID column in the Invoices table.
2. Write a set of action queries coded as a transaction to move rows from the Invoices table to the InvoiceArchive table. Insert all paid invoices from Invoices into InvoiceArchive, but only if the invoice doesn't already exist in the InvoiceArchive table. Then, delete all paid invoices from the Invoices table, but only if the invoice exists in the InvoiceArchive table.

How to manage database security

If you've been using a stand-alone copy of SQL Server installed on your own computer, the security of the system hasn't been of concern. When you install SQL Server for use in a production environment, however, you must configure security to prevent misuse of your data. In this chapter, you'll learn how to do that using either the Management Studio or Transact-SQL.

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
Perspective	584

How to work with SQL Server login IDs

Before a user can work with the data in a database, he must have a valid *login ID* so he can log on to SQL Server. Then, he must have access to the database itself. In the topics that follow, you'll learn how to work with login IDs and how to give a user access to a database. But first, I'll present an overview of how SQL Server manages database security.

An introduction to SQL Server security

Figure 17-1 illustrates how a user gains access to a SQL Server database. First, the user must connect and log on to the server using either an application program or the Management Studio. As you can see, the login ID can be authenticated in one of two ways, which I'll discuss in a moment.

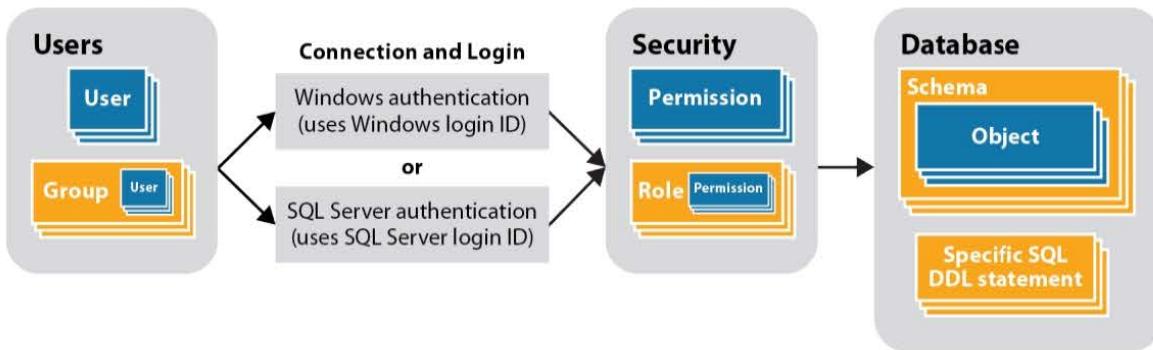
Once the user is logged on to SQL Server, the data he has access to and the operations he can perform depend on the *permissions* that have been granted to him. You can grant *object permissions* so the user can perform specific actions on a specific database object, you can grant *schema permissions* so the user can perform actions on every object in the schema, you can grant *database permissions* so the user can perform specific database operations, and you can grant *server permissions* so the user can perform specific actions at the server level. In addition, you can define a collection of permissions called a *role*. Then, you can assign users to that role to grant them all of the permissions associated with that role. This reduces the number of permissions you must grant each user and makes it easier to manage security. For this reason, roles are used on most systems.

This figure also summarizes the two ways you can manage SQL Server security. First, you can do that by executing SQL statements and system stored procedures from the Query Editor or using the SQLCMD utility. Second, you can use the graphical interface of the Management Studio. You'll learn how to use both of these techniques in this chapter. The technique you use is mostly a matter of preference. However, even if you intend to use the Management Studio, you should still read the topics on using the SQL statements and stored procedures. These topics will help you understand the underlying structure of SQL Server security, which will help you use the Management Studio better.

Although the Management Studio's graphical interface makes it easier to work with security, it can also slow you down. For example, if you need to set up a new database with hundreds of users, you'll have to create those users one at a time using the Management Studio. On the other hand, if you've read chapter 14 and know how to code dynamic SQL, you can code a script that will manage the entire process. For this reason, many experienced system administrators prefer to manage security using Transact-SQL.

Before I go on, you should know about two terms that are used frequently when talking about security. The first term, *principal*, refers to a user, group, login, or role that has access to a database. The second term, *securable*, refers to a SQL Server entity that can be secured. That includes tables, schemas, databases, and the server itself. You'll learn more about principals and securables as you progress through this chapter.

How users gain access to a SQL Server database



Two ways to configure SQL Server security

Method	Description
Transact-SQL	Use Transact-SQL statements to manage login IDs, database users, permissions, and roles.
Management Studio	Use the Management Studio to configure all aspects of system security.

Description

- Typically, a network user must log on to the network at a PC using a *login ID* and password. If the client PC uses Windows, SQL Server can use the Windows login ID defined for the user. Otherwise, you can create a separate SQL Server login ID.
- Once a user is logged on to SQL Server, the security configuration determines which database objects the user can work with and which SQL statements the user can execute.
- Permissions* determine the actions a user can take on a database object, such as a table, view, or stored procedure, on the objects in a schema, on a database, and on a server.
- A *role* is a collection of permissions that you can assign to a user by assigning the user to that role.
- You can create a collection of users in Windows called a *group*. Then, you can assign permissions and roles either to individual users or to a group of users.
- The users, groups, logins, and roles that have access to a server are called *principals*. The entities that can be secured on a server, including the server itself, are called *securables*.
- If you need to set up a new system with many users, it's often easier to code SQL scripts using the SQL security statements. The Management Studio is better for making changes to an existing system or for setting up a small system.
- Even if you use the Management Studio to manage security, you should know how to manage security with Transact-SQL statements. That will help you understand the underlying structure of SQL Server security.

Figure 17-1 An introduction to SQL Server security

How to change the authentication mode

As you learned in chapter 2, you can log on to SQL Server using one of two types of login authentication: *Windows authentication* or *SQL Server authentication*. To accommodate these two types of authentication, a server can be configured to run in one of two *authentication modes*: Windows Authentication mode or Mixed mode. Figure 17-2 summarizes these authentication modes and shows you how you can change from one to the other.

When you install SQL Server, Windows Authentication mode is the default. Then, when a user logs on to SQL Server, authentication is handled by the security that's integrated into Windows. In other words, the login ID and password that the user enters to log on to Windows are also used to log on to SQL Server.

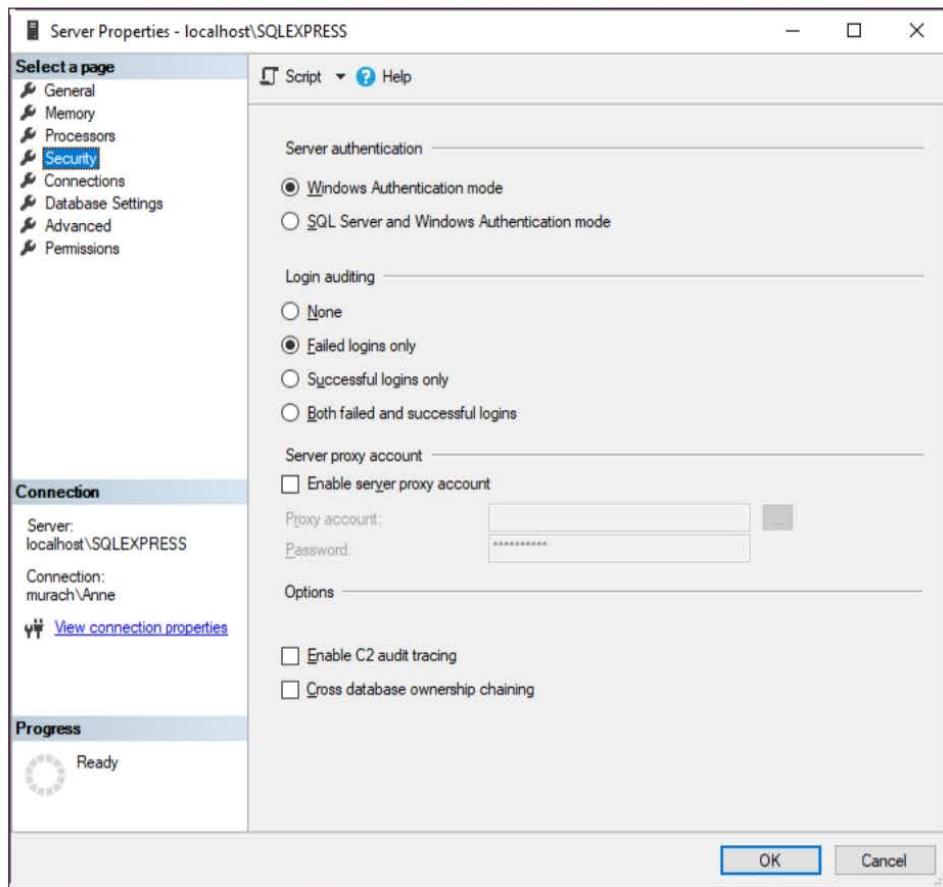
If you use Mixed mode authentication, users can log on using either Windows authentication or SQL Server authentication. When SQL Server authentication is used, the user must enter a SQL Server user ID and password to log on to SQL Server. This user ID and password are separate from the Windows user ID and password, which means that the user must enter two IDs and passwords to access SQL Server. Since non-Windows clients can't use Windows authentication, it's likely that the only time you'll use SQL Server authentication is to support access by non-Windows clients.

To change the authentication mode, you use the Security page of the Server Properties dialog box shown in this figure. The two available options are listed under the Server Authentication heading. The SQL Server and Windows Authentication Mode option corresponds to Mixed mode.

If you change the authentication mode, the Management Studio warns you that the change won't take effect until you stop and restart SQL Server. Before you stop SQL Server, though, you'll want to be sure that there aren't any transactions currently executing. If there are and you stop SQL Server, those transactions won't be committed. If you're working on a desktop server or on a new server with no users, this shouldn't be a problem. If you're working on an active server, however, you shouldn't restart the server until you're sure that no users are connected.

If you install SQL Server with Windows Authentication mode, you should know that the default system administrator login ID, sa, is disabled. That's because this login can only connect using SQL Server authentication. If you later change to Mixed mode, however, the sa login ID isn't automatically enabled. So you'll have to enable it manually. You'll learn how to enable and disable login IDs later in this chapter.

The Security tab of the SQL Server Properties dialog box



The two SQL Server authentication modes

Mode	Description
Windows Authentication mode	Only Windows authentication is allowed. This is the default.
Mixed mode	Both Windows authentication and SQL Server authentication are allowed. To use this mode, select the SQL Server and Windows Authentication Mode option. If your database needs to be accessed by non-Windows clients, you must use Mixed mode.

Description

- To change the SQL Server *authentication mode*, right-click on the server in the Object Explorer of the Management Studio, select the Properties command to display the Server Properties dialog box, then display the Security page.
- When you use *Windows authentication*, access to SQL Server is controlled via the security integrated into Windows. This simplifies login because Windows users only have to log on once.
- When you use *SQL Server authentication*, access to SQL Server is controlled via the separate security built into SQL Server. The user has a login ID and password that are distinct from their Windows login ID and password, so they have to log on twice.

Figure 17-2 How to change the authentication mode

How to create login IDs

When you install SQL Server, it's configured with some built-in IDs. Then, to add additional login IDs, you can use the CREATE LOGIN statement shown in figure 17-3. As you can see, the syntax you use depends on whether you're creating a login for Windows authentication or SQL Server authentication.

To create a new Windows login ID, you can simply specify the login name and the FROM WINDOWS keywords. Then, a login ID with the same name as the Windows login ID is created. Note that the name you specify must include the Windows domain name along with the user or group name, and the name must be enclosed in square brackets. The first CREATE LOGIN statement in this figure, for example, creates a login ID for a Windows user named SusanRoberts in the Windows domain named Accounting.

If you're working with SQL Server Express on your own system, the domain name is just the name of your computer. In that case, though, you probably won't need to set up additional login IDs. The exception is if you have more than one user account defined on your system, in which case you can set up a separate login for each account.

In addition to the login name, you can specify the default database and language. If you set the default database, the user won't have to execute a USE statement to work with that database. If you don't specify a default database, the system database named master is the default.

To create a new SQL Server login ID, you must specify a login name and password. The second statement in this figure, for example, creates a login ID for user JohnDoe with the password "pt8806FG\$B". It also sets the default database to AP.

The last two options determine how password policies are enforced. If the CHECK_EXPIRATION option is on, users are reminded to change passwords, and SQL Server disables IDs that have expired passwords. If the CHECK_POLICY option is on, password policies are enforced, and the CHECK_EXPIRATION option is also on unless it's explicitly turned off.

In most cases, you'll leave these options at their defaults so the password policies specified for the server are enforced. Among other things, the default password policies for SQL Server 2012 and later require that SQL Server logins use *strong passwords*, which are difficult for someone to guess. This figure lists the guidelines for coding strong passwords, and the example in this figure that creates a SQL Server login ID illustrates a strong password.

Another option you can use with the CREATE LOGIN statement is MUST_CHANGE. If you include this option, the user will be prompted for a new password the first time the new login is used. That way, users can set their own passwords. If you specify the MUST_CHANGE option, the CHECK_EXPIRATION and CHECK_POLICY options must also be on.

By the way, you should know that the CREATE LOGIN statement, as well as many of the other statements presented in this chapter, were introduced with SQL Server 2005. These statements replace stored procedures that were used in previous versions of SQL Server. Because these stored procedures may

The syntax of the CREATE LOGIN statement

For Windows authentication

```
CREATE LOGIN login_name FROM WINDOWS  
    [WITH [DEFAULT_DATABASE = database]  
        [, DEFAULT_LANGUAGE = language]]
```

For SQL Server authentication

```
CREATE LOGIN login_name WITH PASSWORD = 'password' [MUST_CHANGE]  
    [, DEFAULT_DATABASE = database]  
    [, DEFAULT_LANGUAGE = language]  
    [, CHECK_EXPIRATION = {ON|OFF}]  
    [, CHECK_POLICY = {ON|OFF}]
```

A statement that creates a new login ID from a Windows account

```
CREATE LOGIN [Accounting\SusanRoberts] FROM WINDOWS;
```

A statement that creates a new SQL Server login ID

```
CREATE LOGIN JohnDoe WITH PASSWORD = 'pt8806FG$B',  
    DEFAULT_DATABASE = AP;
```

Guidelines for strong passwords

- Cannot be blank or null or the values “Password”, “Admin”, “Administrator”, “sa”, or “sysadmin”
- Cannot be the name of the current user or the machine name
- Must contain more than 8 characters
- Must contain at least three of the following: uppercase letters, lowercase letters, numbers, and non-alphanumeric characters (#, %, &, etc.)

Description

- You use the CREATE LOGIN statement to create a new SQL Server login ID or to create a new login ID from a Windows account.
- If you don't specify a default database when you create a login, the default is set to master. If you don't specify a default language, the default is set to the default language of the server. Unless it's been changed, the server language default is English.
- The password you specify for a SQL Server login ID should be a *strong password*. A strong password is not easy to guess and cannot easily be hacked. A password can have up to 128 characters.
- If you include the MUST_CHANGE option, SQL Server will prompt the user for a new password the first time the login ID is used.
- The CHECK_EXPIRATION option determines whether SQL Server enforces password expiration policy. The CHECK_POLICY option determines if password policies, such as strong passwords, are enforced. These options are enforced only if SQL Server is running on Windows Server.

Figure 17-3 How to create login IDs

be dropped in a future release of SQL Server, you should use the statements presented in this chapter instead.

How to delete or change login IDs or passwords

To change an existing login ID, you use the ALTER LOGIN statement shown in figure 17-4. A common task you can perform with this statement is to change the password for a SQL Server Login ID. This is illustrated in the first example in this figure. Notice in this example that you can change the password without specifying the old password. You can also force the user to enter a new password the next time the login is used by coding the MUST_CHANGE option.

Another common task that you can perform with the ALTER LOGIN statement is to disable or enable a login ID as illustrated in the second example. Here, the login ID for a Windows account is being disabled. That means that the user will no longer have access to SQL Server.

The third example in this figure shows how you can use the ALTER LOGIN statement to change a login name. This is useful if a user's name actually changes. It's also useful if one employee replaces another. Then, you can give the new employee the same permissions as the old employee simply by changing the login name.

To delete a login ID, you use the DROP LOGIN statement. This is illustrated in the fourth example in this figure.

The syntax of the DROP LOGIN statement

```
DROP LOGIN login_name
```

The syntax of the ALTER LOGIN statement

For Windows authentication

```
ALTER LOGIN login_name {{ENABLE|DISABLE}|WITH  
    [NAME = login_name]  
    [, DEFAULT_DATABASE = database]  
    [, DEFAULT_LANGUAGE = language]}
```

For SQL Server authentication

```
ALTER LOGIN login_name {{ENABLE|DISABLE}|WITH  
    [PASSWORD = 'password' [OLD_PASSWORD = 'oldpassword']  
    [MUST_CHANGE]]  
    [, NAME = login_name]  
    [, DEFAULT_DATABASE = database]  
    [, DEFAULT_LANGUAGE = language]  
    [, CHECK_EXPIRATION = {ON|OFF}]  
    [, CHECK_POLICY = {ON|OFF}]}}
```

Statements that use the ALTER LOGIN and DROP LOGIN statements

A statement that changes the password for a SQL Server login ID

```
ALTER LOGIN JohnDoe WITH PASSWORD = '1g22A%G45x';
```

A statement that disables a Windows login ID

```
ALTER LOGIN [Accounting\SusanRoberts] DISABLE;
```

A statement that changes a login name

```
ALTER LOGIN JohnDoe WITH NAME = JackWilliams;
```

A statement that deletes a SQL Server login ID

```
DROP LOGIN JackWilliams;
```

Description

- You use the ALTER LOGIN statement to enable or disable a login ID, change the name for a login ID, or change the default database or language. For a SQL Server login ID, you can also change the password and the password options.
- You use the DROP LOGIN statement to drop a login ID.

How to work with database users

Each database maintains a list of the users that are authorized to access that database. This list is distinct from the list of login IDs that's maintained by the server. Figure 17-5 presents the SQL statements you use to maintain the list of users for a database.

You use the `CREATE USER` statement to create a database user. On this statement, you code the name of the user, which is usually the same as the login name. In that case, you don't need to specify the login name. This is illustrated in the first example in this figure. If you want to use a user name that's different from the login name, however, you can include the `FOR LOGIN` clause to specify the login name that the user name is mapped to.

In most cases, it's not a good idea to use two different names for the same user. For this reason, the `FOR LOGIN` clause is generally omitted. However, since login IDs generated from Windows user names include the domain name, those login IDs can be quite long. If all of your users are on the same Windows domain, then, you may want to use just the user names for the database users. This is illustrated in the second example in this figure, which creates a database user named SusanRoberts for the login ID Accounting\SusanRoberts.

You can also specify a default schema for a database user as illustrated in the third example in this figure. Then, when SQL Server searches for an object for that user, it will look for the object in the user's default schema before it looks in the `dbo` schema.

After you create a database user, the user can set the database as the current database using the `USE` statement but can't perform any operations on the database or the objects it contains. To do that, the user must be granted object and database permissions. You'll learn how to grant these permissions in a moment.

If you need to change a database user, you can use the `ALTER USER` statement. This statement lets you change the user name or the default schema for the user. The fourth statement in this figure, for example, changes the name of a database user from SusanRoberts to SusanStanley, and the fifth statement changes the default schema for a user to Marketing.

Finally, if you need to delete a database user, you use the `DROP USER` statement as illustrated in the last example. The only information you specify on this statement is the user name.

Note that all three of these statements work with the current database. For this reason, you must be sure to change the database context to the database you want to work with before you execute any of these statements. If you don't, you may inadvertently create, change, or delete a user in the wrong database.

The syntax of the CREATE USER statement

```
CREATE USER user_name  
    [{FOR|FROM} LOGIN login_name]  
    [WITH DEFAULT_SCHEMA = schema_name]
```

The syntax of the ALTER USER statement

```
ALTER USER user_name WITH  
    [NAME = new_user_name]  
    [, DEFAULT_SCHEMA = schema_name]
```

The syntax of the DROP USER statement

```
DROP USER user_name
```

Statements that work with database users

A statement that creates a database user with the same name as a login ID

```
CREATE USER JohnDoe;
```

A statement that creates a database user for a Windows user account

```
CREATE USER SusanRoberts FOR LOGIN [Accounting\SusanRoberts];
```

A statement that creates a database user and assigns a default schema

```
CREATE USER SusanRoberts FOR LOGIN [Accounting\SusanRoberts]  
    WITH DEFAULT_SCHEMA = Accounting;
```

A statement that changes a user name

```
ALTER USER SusanRoberts WITH NAME = SusanStanley;
```

A statement that assigns a default schema to a user

```
ALTER USER JohnDoe WITH DEFAULT_SCHEMA = Marketing;
```

A statement that deletes a database user

```
DROP USER JohnDoe;
```

Description

- You use the CREATE USER statement to create a user for a login ID for the current database. If the login name is the same as the user name, you can omit the FOR LOGIN clause.
- When you create a database user, you can specify a default schema. Then, SQL Server will look in this schema when it searches for objects for the database user before it looks in the default schema (dbo).
- The ALTER USER statement lets you change the name of an existing database user or change the default schema for a user.
- You use the DROP USER statement to delete a user from the current database.
- Since all three of these statements work on the current database, you must change the database context using the USE statement before executing any of these statements.

How to work with schemas

As you know, the tables, views, functions, and procedures of a database are stored in schemas. If you don't specify a schema when you create these objects, they're stored in the default schema.

One advantage of using schemas is that you can grant permissions to all the objects in a schema by granting permissions to the schema. Another advantage is that users don't own database objects, they own schemas. Because of that, if you need to delete a user, you can just transfer the schemas that user owns to another user rather than having to transfer the ownership of each individual object.

Figure 17-6 presents the SQL statements for working with schemas. To create a schema, you use the CREATE SCHEMA statement. The only information you must include on this statement is the schema name. This is illustrated in the first CREATE SCHEMA statement in this figure, which creates a schema named Accounting.

When you create a schema, you can also create tables and views within that schema, and you can grant, revoke, and deny permissions to those tables and views. For example, the second CREATE SCHEMA statement shown here creates a schema named Marketing. In addition, it creates a table named Contacts within the Marketing schema. Notice that because the CREATE TABLE statement is coded within the CREATE SCHEMA statement, it isn't necessary to specify the schema on the CREATE TABLE statement. For this to work, the CREATE SCHEMA statement must be coded as a separate batch.

By default, a schema is owned by the owner of the database. In most cases, that's what you want. If you want to assign a different owner to a schema, however, you can include the AUTHORIZATION clause with the name of the user or role you want to own the schema on the CREATE SCHEMA statement.

The ALTER SCHEMA statement lets you transfer a securable from one schema to another. For example, the ALTER SCHEMA statement in this figure transfers the Contacts table in the Marketing schema that was created by the second statement to the Accounting schema that was created by the first statement. Note that you can't transfer an object to a different schema if any views or functions are schema-bound to the object. Because of that, you'll want to make sure an object is in the correct schema before you create any views or functions that are bound to it.

To delete a schema from a database, you use the DROP SCHEMA statement. The last statement in this figure, for example, deletes the Marketing schema. Keep in mind that before you delete a schema, you must delete any objects it contains or transfer them to another schema.

The syntax of the CREATE SCHEMA statement

```
CREATE SCHEMA schema_name [AUTHORIZATION owner_name]
    [table_definition]...
    [view_definition]...
    [grant_statement]...
    [revoke_statement]...
    [deny_statement]...
```

The syntax of the ALTER SCHEMA statement

```
ALTER SCHEMA schema_name TRANSFER securable_name
```

The syntax of the DROP SCHEMA statement

```
DROP SCHEMA schema_name
```

Statements that work with schemas

A statement that creates a schema

```
CREATE SCHEMA Accounting;
```

A statement that creates a schema and a table within that schema

```
CREATE SCHEMA Marketing
    CREATE TABLE Contacts
        (ContactID INT NOT NULL IDENTITY PRIMARY KEY,
        ContactName VARCHAR(50) NOT NULL,
        ContactPhone VARCHAR(50) NULL,
        ContactEmail VARCHAR(50) NULL);
```

A statement that transfers a table from one schema to another

```
ALTER SCHEMA Accounting TRANSFER Marketing.Contacts;
```

A statement that deletes a schema

```
DROP SCHEMA Marketing;
```

Description

- You use the CREATE SCHEMA statement to create a schema in the current database. You can also create tables and views within the new schema, and you can grant, revoke, or deny permissions for those tables and views.
- After you create a schema, you can create any object within that schema by qualifying the object name with the schema name.
- You use the ALTER SCHEMA statement to transfer an object from one schema to another.
- You can't transfer an object from one schema to another if any views or functions are bound to it.
- When you transfer an object from one schema to another, all the permissions that were associated with that object are dropped.
- You use the DROP SCHEMA statement to delete a schema. The schema you delete must not contain any objects.

How to work with permissions

Now that you understand how to create login IDs and database users, you need to learn how to grant users permission to work with a database and the objects it contains. That's what you'll learn in the topics that follow. In addition, you'll learn how to grant login IDs permission to work with the server.

How to grant or revoke object permissions

Figure 17-7 presents the GRANT and REVOKE statements you use to grant or revoke permissions to use an object in the current database. In the GRANT clause, you list the permissions you want to grant. You'll see a list of the standard permissions in the next figure.

You code the name of the object for which this permission is granted in the ON clause. This object can be a table, a view, a stored procedure, or a user-defined function. If the object is contained in a schema other than the default schema, you must specify the schema name along with the object name. Note that you can only grant permissions for a single object with the GRANT statement.

In the TO clause, you code one or more database principal names to which you're granting the permission. Typically, this is the database user name. The statement in this figure, for example, grants permission for the user SusanRoberts to select data from the Invoices table. You can also use this statement to assign permissions to a database role. You'll learn more about database roles later in this chapter.

If you code the GRANT statement with the optional WITH GRANT OPTION clause, you delegate to this user the permission to GRANT this same permission to others. Since it's simpler to have a single person or group managing the security for a database, I don't recommend that you use this option. If you do, however, you should keep good records so you can later revoke this permission if you begin to have security problems.

The syntax of the REVOKE statement is similar to the syntax of the GRANT statement. You code a list of the permissions you're revoking in the REVOKE clause, the object name in the ON clause, and one or more database principal names in the FROM clause. The statement in this figure, for example, revokes the SELECT permission for the Invoices table that was granted to user SusanRoberts by the GRANT statement.

You can code two optional clauses in the REVOKE statement. These clauses are related to the WITH GRANT OPTION clause you can code in the GRANT statement. The GRANT OPTION FOR clause revokes the user's permission to grant this permission to others. The CASCADE clause revokes this permission from all of the users to whom this user has granted permission. If you avoid using the WITH GRANT OPTION clause, you won't have to use these clauses.

How to grant object permissions

The syntax of the GRANT statement for object permissions

```
GRANT permission [, ...]
ON [schema_name.]object_name [(column [, ...])]
TO database_principal [, ...]
[WITH GRANT OPTION]
```

A GRANT statement that grants SELECT permission for the Invoices table

```
GRANT SELECT
ON Invoices
TO SusanRoberts;
```

How to revoke object permissions

The syntax of the REVOKE statement for object permissions

```
REVOKE [GRANT OPTION FOR] permission [, ...]
ON [schema_name.]object_name [(column [, ...])]
FROM database_principal [, ...]
[CASCADE]
```

A REVOKE statement that revokes SELECT permission

```
REVOKE SELECT
ON Invoices
FROM SusanRoberts;
```

Description

- You use this GRANT statement format to give a user permission to work with a database object. This format of the REVOKE statement takes object permissions away. See figure 17-8 for a list of the standard permissions that can be granted for objects.
- The object_name argument specifies the object for which the permission is being granted or revoked and can specify a table, a view, a stored procedure, or a user-defined function. If you specify a table, a view, or a table-valued function, you can also list the columns for which SELECT, UPDATE, or REFERENCES permissions are granted or revoked.
- The database_principal argument in the TO and FROM clauses can be the name of a database user or a user-defined role.
- The WITH GRANT OPTION clause gives a user permission to grant this permission to other users.
- The REVOKE statement includes two clauses that undo WITH GRANT OPTION. GRANT OPTION FOR revokes the user's permission to grant the permission to others. CASCADE revokes the permission from any other users who were given the permission by this user.
- Since both the GRANT and REVOKE statements work on the current database, you must first change the database context using the USE statement.

The SQL Server object permissions

Figure 17-8 lists the specific object permissions that you can code in either the GRANT or REVOKE statement. The first four permissions let the user execute the corresponding SQL statement: SELECT, UPDATE, INSERT, or DELETE. The fifth permission, EXECUTE, lets the user run an executable database object.

Each permission can be granted only for certain types of objects. For example, you can grant SELECT permission only to an object from which you can select data, such as a table or view. Likewise, you can grant EXECUTE permission only to an object that you can execute, such as a stored procedure or scalar function.

The REFERENCES permission lets a user refer to an object, even if the user doesn't have permission to use that object directly. For example, to create a FOREIGN KEY constraint that refers to another table, the user would need to have REFERENCES permission on that other table. Of course, he'd also need permission to create a table. You'll see how to grant permissions like this in a moment.

You also need to assign the REFERENCES permission to objects that are referenced by a function or view that's created with the WITH SCHEMABINDING clause. Since this permission is only needed for users who'll be creating database objects, you'll probably never assign it individually. Instead, you'll include it with other permissions in a database role as you'll learn later in this chapter.

The last permission, ALTER, lets the user change the definition of an object. This permission is typically given to users who are responsible for designing a database. That includes database administrators and, in many cases, programmers.

In addition to the permissions shown here, SQL Server 2019 continues to support the deprecated ALL permission. Because this permission may not work in future versions of SQL Server, you should avoid using it. If you do use it, though, you should know that despite its name, this permission doesn't always grant all permissions that are applicable to the object. For example, when working with database permissions as shown in figure 17-10, the ALL permission doesn't include the CREATE SCHEMA permission.

The standard permissions for SQL Server objects

Permission	Description	Applies to
SELECT	Lets the user select the data.	Tables, views, and table-valued functions
UPDATE	Lets the user update existing data.	Tables, views, and table-valued functions
INSERT	Lets the user insert new data.	Tables, views, and table-valued functions
DELETE	Lets the user delete existing data.	Tables, views, and table-valued functions
EXECUTE	Lets the user execute a procedure or function.	Stored procedures and scalar and aggregate functions
REFERENCES	Lets the user create objects that refer to the object.	Tables, views, and functions
ALTER	Lets the user modify an object.	Tables, procedures, functions, and sequences

A GRANT statement that grants permission to run action queries

```
GRANT INSERT, UPDATE, DELETE
ON Invoices
TO SusanRoberts;
```

A REVOKE statement that revokes the DELETE permission

```
REVOKE DELETE
ON Invoices
FROM SusanRoberts;
```

A GRANT statement that grants permission to execute a stored procedure

```
GRANT EXECUTE
ON spInvoiceReport
TO [Payroll\MarkThomas], JohnDoe, TomAaron;
```

A GRANT statement that grants SELECT permission to specific columns

```
GRANT SELECT
ON Vendors (VendorName, VendorAddress1, VendorCity, VendorState, VendorZipCode)
TO TomAaron, [Payroll\MarkThomas];
```

A GRANT statement that grants REFERENCES permission to the Contacts table in the Accounting schema

```
GRANT REFERENCES
ON Accounting.Contacts
To JohnDoe;
```

A GRANT statement that grants permission to alter a table

```
GRANT ALTER
ON Vendors
To JoelMurach;
```

Description

- You can only grant permissions that are appropriate for the object or schema.
- You can grant or revoke SELECT, UPDATE, or REFERENCES permission to specific columns in a table, view, or table-valued function. However, a view is typically a better way to limit access to specific columns.

Figure 17-8 The SQL Server object permissions

How to grant or revoke schema permissions

In addition to granting or revoking permissions to individual objects in a database, you can grant or revoke permissions to all the objects in a schema. To do that, you use the formats of the GRANT and REVOKE statements shown in figure 17-9.

The main difference between the formats shown here and the formats shown in figure 17-7 is that the ON clause uses a class name and a *scope qualifier* (::). In this case, the class name is SCHEMA. You can also use a class name and scope qualifier when you grant or revoke object permissions. For example, I could have coded the GRANT statement in figure 17-7 like this:

```
GRANT SELECT  
ON OBJECT :: Invoices  
TO SusanRoberts;
```

However, because the class name and scope qualifier aren't required when you grant or revoke object permissions, I've omitted them from the syntax of the GRANT and REVOKE statements shown in figure 17-7.

Before you go on, you should realize that to delete an object from a schema, a user must have ALTER permission to the schema. A user must also have ALTER permission to a schema to create an object in the schema. In addition, the user must have permission to create the object in the database. The exception is that you can create a sequence simply by granting the CREATE SEQUENCE permission to the schema. To alter an object in the schema, the user only needs to have ALTER permission on the object.

How to grant schema permissions

The syntax of the GRANT statement for schema permissions

```
GRANT permission [, ...]
ON SCHEMA :: schema_name
TO database_principal [, ...]
[WITH GRANT OPTION]
```

A GRANT statement that grants UPDATE permission for the Accounting schema

```
GRANT UPDATE
ON SCHEMA :: Accounting
TO JohnDoe;
```

A GRANT statement that grants ALTER permission to a schema

```
GRANT ALTER
ON SCHEMA :: Marketing
TO JudyTaylor;
```

How to revoke schema permissions

The syntax of the REVOKE statement for schema permissions

```
REVOKE [GRANT OPTION FOR] permission [, ...]
ON SCHEMA :: schema_name
FROM database_principal [, ...]
[CASCADE]
```

A REVOKE statement that revokes UPDATE permission

```
REVOKE UPDATE
ON SCHEMA :: Accounting
FROM JohnDoe;
```

Description

- You use this format of the GRANT statement to give a user permission to work with all the objects in a database schema. This format of the REVOKE statement takes schema permissions away. These statements work just as they do for object permissions.
- To create an object in a schema or to delete an object from a schema, the user must have ALTER permission to the schema. To create an object, the user must also have CREATE permission to the database that contains the object. See figure 17-10 for information on database permissions.
- You can also grant or revoke the CREATE SEQUENCE permission on a schema.
- The ON clause in the GRANT and REVOKE statements includes a class name (SCHEMA) and a *scope qualifier* (::). Although class names and scope qualifiers can be used with other Transact-SQL statements, they're not usually required.

How to grant or revoke database permissions

Figure 17-10 presents the syntax of the GRANT and REVOKE statements you use to work with database permissions. In the GRANT clause, you list the statements you want to grant a user permission to execute. Some of the statements you can include in this list are shown in this figure. Then, in the TO clause, you list the users you want to have these permissions. The GRANT statement in this figure, for example, grants two users permission to create views. Remember that to create a view, though, the user must also have permission to alter the schema that contains it.

The syntax of the REVOKE statement is identical except that you code the user names in the FROM clause. The REVOKE statement shown in this figure, for example, revokes permission for the specified user to create databases or tables.

How to grant database permissions

The syntax of the GRANT statement for database permissions

```
GRANT permission [, ...]
TO database_principal [, ...]
[WITH GRANT OPTION]
```

A GRANT statement that gives permission to create views

```
GRANT CREATE VIEW
TO JohnDoe, SusanRoberts;
```

How to revoke database permissions

The syntax of the REVOKE statement for database permissions

```
REVOKE permission [, ...]
FROM database_principal [, ...]
[CASCADE]
```

A REVOKE statement that revokes permission to create databases and tables

```
REVOKE CREATE DATABASE, CREATE TABLE
FROM SylviaJones;
```

Some of the permissions that can be explicitly permitted

```
CREATE DATABASE
CREATE TABLE
CREATE VIEW
CREATE PROCEDURE
CREATE FUNCTION
CREATE SCHEMA
```

Description

- In addition to granting or revoking permissions for objects and schemas, you can grant or revoke permissions for databases.
- The list of SQL statements shown above only includes those discussed in this book. For a complete list, refer to the “GRANT Database Permissions (Transact-SQL)” topic in the SQL Server documentation.

How to grant or revoke server permissions

The highest level at which you can grant or revoke permissions is the server level. Server permissions are typically reserved for system and database administrators. Figure 17-11 lists some of the permissions that are available and shows how to grant and revoke them.

Because the GRANT and REVOKE statements for server permissions are similar to the GRANT and REVOKE statements for other types of permissions, I'll just point out the main difference here. That is, because the permission is for the server and not the current database, the TO and FROM clauses must name a server principal instead of a database principal. A server principal can be either a login ID or a user-defined server role. In this figure, for example, both statements refer to a login ID. You'll learn more about server roles in just a minute.

How to grant server permissions

The syntax of the GRANT statement for server permissions

```
GRANT permission [, ...]  
TO server_principal [, ...]  
[WITH GRANT OPTION]
```

A GRANT statement that gives permission to create, alter, and drop databases

```
GRANT ALTER ANY DATABASE  
TO JoelMurach;
```

How to revoke server permissions

The syntax of the REVOKE statement for server permissions

```
REVOKE permission [, ...]  
FROM server_principal [, ...]  
[CASCADE]
```

A REVOKE statement that revokes permission to create server roles

```
REVOKE CREATE SERVER ROLE  
FROM [Administration\SylviaJones];
```

Some of the permissions that can be explicitly permitted

Permission	Description
CONTROL SERVER	Can perform any activity on the server.
CREATE ANY DATABASE	Can create databases.
CREATE SERVER ROLE	Can create server roles.
ALTER ANY DATABASE	Can create, alter, and drop databases.
ALTER ANY LOGIN	Can create, alter, and drop logins.
ALTER ANY SERVER ROLE	Can create, alter, and drop server roles.
VIEW ANY DATABASE	Can view database properties.

Description

- You can also use the GRANT and REVOKE statements to grant and revoke permissions for the entire server. When you do that, you specify a login or server role on the TO or FROM clause.
- For a complete list of server permissions, refer to the “GRANT Server Permissions (Transact-SQL)” topic in the SQL Server documentation.

Figure 17-11 How to grant or revoke server permissions

How to work with roles

Now that you've learned how to grant object and database permissions to a user, you can set up security on your database. If a system has many users, however, granting and revoking all of these permissions one by one would require a lot of coding. To help reduce the amount of coding and to help you keep your database security organized, you can use roles.

As you know, a role is a collection of permissions. When you assign a user to a particular role, you grant them all of the permissions associated with that role. SQL Server supports two different types of roles: fixed roles and user-defined roles. You'll learn how to work with both of these types of roles in the topics that follow.

How to work with the fixed server roles

Fixed roles are roles that are built into SQL Server. These roles can't be deleted and the permissions associated with them can't be modified. SQL Server provides two types of fixed roles: *fixed server roles* and *fixed database roles*. Figure 17-12 shows you how to work with the fixed server roles. You'll learn how to work with the fixed database roles later in this chapter.

The fixed server roles typically include users who manage the server. For example, the sysadmin role is intended for system administrators. For this reason, it grants permission to perform any task on the server. If you're a member of the Windows BUILTIN\Administrators group, you're a member of this role by default.

The securityadmin role is intended for those users who need to be able to manage security. The members of this role are allowed to work with login IDs and passwords. The dbcreator role is intended for those users who need to be able to work with database objects. The members of this role can create, alter, and drop databases. Although SQL Server provides other server roles, these are the ones you'll use most often.

To assign a user to a server role or to remove a user from a server role, you use the ALTER SERVER ROLE statement. On this statement, you specify the name of the server role. Then, to assign a user to the role, you specify the login ID of the user on the ADD MEMBER clause. To remove a user from the role, you specify the login ID on the DROP MEMBER clause. The first statement in this figure, for example, adds user JohnDoe to the sysadmin server role. The second statement drops this user from that role.

You can also use the ALTER SERVER ROLE statement to rename a server role. However, you can't rename fixed server roles, only user-defined server roles. You'll see an example of that in the next figure.

One additional fixed server role you should know about is the public server role. Each login that's created on the server is automatically assigned to this role and can't be removed. Then, the default permission for this role, VIEW ANY DATABASE, lets any login view the properties of any database on the server. If that's not what you want, you can revoke this permission from the role. To do that, you code the role name on the TO clause of the REVOKE statement. In most cases, though, it's not necessary to change the permissions for this role.

The syntax of the ALTER SERVER ROLE statement

```
ALTER SERVER ROLE role_name
{
    ADD MEMBER server_principal |
    DROP MEMBER server_principal |
    WITH NAME = new_role_name
}
```

A statement that assigns a user to a server role

```
ALTER SERVER ROLE sysadmin ADD MEMBER JohnDoe;
```

A statement that removes a user from a server role

```
ALTER SERVER ROLE sysadmin DROP MEMBER JohnDoe;
```

Some of the SQL Server fixed server roles

Role	Description
sysadmin	Can perform any activity on the server. By default, all members of the Windows BUILTIN\Administrators group are members of this role.
securityadmin	Can manage login IDs and passwords for the server and can grant, deny, and revoke database permissions.
dbcreator	Can create, alter, drop, and restore databases.

Description

- A role is a collection of permissions you can assign to a user or group of users. By assigning a user to a role, you grant that user all of the permissions of the role. You can use roles to simplify user and security administration.
- SQL Server has built-in, or *fixed*, roles defined at the server level and at the database level. In addition, you can create user-defined roles for your server or database.
- Each role is assigned a set of permissions. For example, the dbcreator role can execute CREATE DATABASE, ALTER DATABASE, DROP DATABASE, and RESTORE DATABASE statements. This role can also add new members to the role.
- You use the ALTER SERVER ROLE statement to add a user to or remove a user from a server role. You can also use this statement to rename a user-defined server role.
- The fixed server roles are intended for users who are involved in the administration of the server. For a complete list of the fixed server roles, see the “Server-level Roles” topic in the SQL Server documentation.
- The ALTER SERVER ROLE statement was introduced with SQL Server 2012. In previous versions of SQL Server, you used the sp_AddSrvRoleMember system stored procedure to add a user to a server role, and you used the sp_DropSrvRoleMember procedure to remove a user from a server role.

Figure 17-12 How to work with the fixed server roles

How to work with user-defined server roles

In addition to fixed server roles, SQL Server 2012 and later let you create *user-defined server roles*. Like the fixed server roles, a user-defined server role consists of a set of permissions that you can grant to a user by giving them membership in that role. Unlike the fixed server roles, you can create your own user-defined server roles, you can modify the permissions associated with those roles, and you can delete the roles when necessary. Figure 17-13 presents the two SQL statements you use to create and delete user-defined server roles.

The CREATE SERVER ROLE statement creates a new role on the server. The role name you specify on this statement must be unique: It can't be the same as another user-defined server role, fixed server role, or login name. The first statement in this figure, for example, creates a new server role named Consultant.

By default, a user-defined server role is owned by the login that creates it. If that's not what you want, you can include the AUTHORIZATION clause on the CREATE SERVER ROLE statement. This clause names the login or fixed server role that will own the new role.

Once a role is defined, you can use the GRANT statement to grant permissions to that role. To do that, you simply code the role name in the TO clause instead of a user name. The GRANT statement in this figure, for example, grants the Consultant role ALTER ANY LOGIN permission. Note that because these permissions can only be assigned at the server level, the current database must be set to master when they're executed.

The next statement in this figure assigns a user to the new role. That means that this user now has the ALTER ANY LOGIN permission that was assigned to the role by the previous GRANT statement. In addition, the next statement assigns the new role as a member of the dbcreator role. If you look back to figure 17-12, you'll see that this role grants any member of the role permission to create, alter, drop, and restore databases. Since the member itself is a role, any member of that role now has permission to create, alter, drop, and restore databases.

The next statement shows how to change the name of a user-defined server role. To do that, you use the WITH NAME clause of the ALTER SERVER ROLE statement. The statement shown here, for example, renames the Consultant role to DBConsultant.

To drop a user-defined server role, you use the DROP SERVER ROLE statement. Before you do that, however, you must delete all of the members of the role. The last two statements in this figure, for example, drop the member of the DBConsultant role and then drop the role.

Because the fixed server roles that SQL Server provides are adequate for most systems, you won't usually need to create user-defined server roles. That's because a limited number of users are typically given permissions at the server level. However, many users can be given permissions at the database level. Because of that, you're more likely to create user-defined database roles. You'll learn how to do that in a minute. But first, you should know how to display information about server roles and how to work with fixed database roles.

The syntax of the CREATE SERVER ROLE statement

```
CREATE SERVER ROLE role_name [AUTHORIZATION server_principal]
```

The syntax of the DROP SERVER ROLE statement

```
DROP SERVER ROLE role_name
```

Statements that work with a user-defined server role

A statement that creates a new server role

```
CREATE SERVER ROLE Consultant;
```

A statement that grants permissions to the new role

```
GRANT ALTER ANY LOGIN  
TO Consultant;
```

A statement that assigns a user to the new role

```
ALTER SERVER ROLE Consultant ADD MEMBER JohnDoe;
```

A statement that assigns the new role to a fixed server role

```
ALTER SERVER ROLE dbcreator ADD MEMBER Consultant;
```

A statement that changes the name of the new role

```
ALTER SERVER ROLE Consultant WITH NAME = DBConsultant;
```

Statements that delete the new role

```
ALTER SERVER ROLE DBConsultant DROP MEMBER JohnDoe;  
DROP SERVER ROLE DBConsultant;
```

Description

- You use the CREATE SERVER ROLE statement to create a *user-defined server role*. Role names can be up to 128 characters in length and can include letters, symbols, and numbers, but not the backslash (\) character.
- The AUTHORIZATION clause lets you specify a login or fixed server role that owns the user-defined role. If you omit this clause, the role will be owned by the login that executes the statement.
- Once you create a server role, you can grant permissions to or revoke permissions from the role. To do that, the current database must be master.
- To add members to a user-defined server role or delete members from the role, you use the ALTER SERVER ROLE statement shown in figure 17-12.
- You use the DROP SERVER ROLE statement to delete a user-defined server role. You can't delete a fixed server role or the public server role.
- Before you can delete a server role, you must delete all of its members. To find out how to list the members of a role, see figure 17-14.
- The CREATE SERVER ROLE and DROP SERVER ROLE statements were introduced with SQL Server 2012.

Figure 17-13 How to work with user-defined server roles

How to display information about server roles and role members

After you set the role membership for the fixed and user-defined server roles, you may want to review the role and membership information. One way to do that is to use the Management Studio. However, SQL Server also provides two catalog views that contain information about server roles and members. Figure 17-14 illustrates how you can use these catalog views.

To start, you can use the `sys.server_principals` catalog view to get a list of the server roles. In the first example in this figure, for instance, I used a `SELECT` statement to retrieve the name, `principal_id`, and `is_fixed_role` columns from this table. I also restricted the rows that were retrieved to roles (`type = 'R'`), since logins and Windows groups are also server principals. In the result set, you can see that all of the roles except for the first one and the last one are fixed roles. The first role is the public server role, and the last role is the user-defined Consultant role that I created in figure 17-13.

To list the members of a role, you have to join the `sys.server_role_members` catalog view with the `sys.server_principals` catalog view. The `sys.server_role_members` view contains every combination of server role principal ID and member principal ID. Then, you can join the `member_principal_id` column in that view with the `principal_id` column in the `sys.server_principals` view to get results like those shown in the second example in this figure.

If you only need to get information about fixed server roles and members, you can use two system stored procedures instead of the catalog views. The third example in this figure shows how to use the `sp_HelpSrvRole` procedure to get information about the server roles. That information includes the name and description for each server role. Although you can code a role name as a parameter so information for a single role is displayed, you're more likely to omit this parameter.

To get information about the members of a fixed server role, you can use the `sp_HelpSrvRoleMember` stored procedure. In this figure, for example, the stored procedure will return information about the members of the `sysadmin` role. If you omit the role name, this procedure will return information about the members in all of the fixed server roles that have at least one member.

How to display information for any server role

```
SELECT name, principal_id, is_fixed_role
FROM sys.server_principals
WHERE type = 'R';
```

	name	principal_id	is_fixed_role
1	public	2	0
2	sysadmin	3	1
3	securityadmin	4	1
4	serveradmin	5	1
5	setupadmin	6	1
6	processadmin	7	1
7	diskadmin	8	1
8	dbcreator	9	1
9	bulkadmin	10	1
10	Consultant	271	0

How to display member information for any server role

```
SELECT member_principal_id, name
FROM sys.server_role_members AS srm
JOIN sys.server_principals AS sp
    ON srm.member_principal_id = sp.principal_id
WHERE srm.role_principal_id = 271;
```

	member_principal_id	name
1	268	JohnDoe

How to display information for fixed server roles

The syntax for sp_HelpSrvRole

```
sp_HelpSrvRole [[@srvrolename = ] 'server_role_name']
```

A statement that lists the fixed server roles

```
EXEC sp_HelpSrvRole;
```

How to display member information for fixed server roles

The syntax for sp_HelpSrvRoleMember

```
sp_HelpSrvRoleMember [[@srvrolename = ] 'server_role_name']
```

A statement that lists the members of the sysadmin role

```
EXEC sp_HelpSrvRoleMember sysadmin;
```

Description

- The sys.server_principals catalog view contains information about each principal defined on the server. To display information about just the server roles, use a SELECT statement with a WHERE clause that checks for a type of 'R'.
- The sys.server_role_members catalog view contains a list of each role principal/member principal combination. To display information about the members of a server role, join this view with the sys.server_role_member view and restrict the results to the principal ID of the role whose members you want to display.
- To display information about just the fixed server roles and members, you can use the sp_HelpSrvRole and sp_HelpSrvRoleMember system stored procedures.

Figure 17-14 How to display information about server roles and role members

How to work with the fixed database roles

Figure 17-15 lists the fixed database roles and shows you how to work with them. These roles are added automatically to each new database you create. In addition, when you create a database, you're automatically added to the db_owner database role.

To add a member to a database role or to delete a member from a database role, you use the ALTER ROLE statement. This statement is similar to the ALTER SERVER ROLE statement you saw earlier in this chapter. The main difference is that you specify a database principal on the ADD MEMBER and DROP MEMBER clauses. A database principal can be a database user, a Windows user or group name, or a user-defined database role. You'll see how to create user-defined database roles in a moment. For now, just realize that the ability to assign user-defined database roles as members of fixed database roles makes assigning permissions flexible and convenient.

Like the ALTER SERVER ROLE statement, you can also use the ALTER ROLE statement to rename a user-defined database role. You'll see an example of that in the next figure.

In addition to the fixed database roles listed in this figure, SQL Server includes a special fixed database role named public. This role is included in every database, and every database user is automatically a member of this role. This role has no permissions by default, however, so you don't need to be concerned about security violations due to the existence of this role. You can't add or drop members from this role, nor can you delete the role from a database. If you want all users to have some basic permissions on a database, however, you can assign those permissions to this role.

The syntax of the ALTER ROLE statement

```
ALTER ROLE role_name
{
    ADD MEMBER database_principal |
    DROP MEMBER database_principal |
    WITH NAME = new_name
}
```

A statement that assigns a user to a database role

```
ALTER ROLE db_owner ADD MEMBER JohnDoe;
```

A statement that removes a user from a database role

```
ALTER ROLE db_owner DROP MEMBER JohnDoe;
```

The SQL Server fixed database roles

Role	Description
db_owner	Has all permissions for the database.
db_accessadmin	Can add or remove login IDs for the database.
db_securityadmin	Can manage object permissions, database permissions, roles, and role memberships.
db_ddladmin	Can issue all DDL statements except GRANT, REVOKE, and DENY.
db_datawriter	Can insert, delete, or update data from any user table in the database.
db_datareader	Can select data from any user table in the database.
db_denydatawriter	Can't insert, delete, or update data from any user table in the database.
db_denydatareader	Can't select data from any user table in the database.
db_backupoperator	Can back up the database and run consistency checks on the database.

Description

- The *fixed database roles* are added to each database you create. You can add and delete members from these roles, but you can't delete the roles.
- You use the ALTER ROLE statement to assign a user to or remove a user from a database role in the current database. You can also use this statement to change the name of a user-defined database role.
- The database_principal parameter can be the name of a database user, a user-defined database role, or a Windows login or group. If you specify a Windows login or group that doesn't have a corresponding database user, a database user is created.
- The users you specify are assigned to or removed from the role you name in the current database. Because of that, you should be sure to change the database context before executing one of these stored procedures.
- SQL Server also provides a public database role. Any user that's not given specific permissions on a securable is given the permissions assigned to the public role.
- Prior to SQL Server 2012, you could use the ALTER ROLE statement only to change the name of a user-defined database role. To add or drop a role, you used the sp_AddRoleMember or sp_DropRoleMember system stored procedure.

Figure 17-15 How to work with the fixed database roles

How to work with user-defined database roles

Figure 17-16 presents the two SQL statements you use to create and delete user-defined database roles. If you compare these statements with the statements for creating and deleting user-defined server roles, you'll see that they're almost identical. The only difference is that you can specify a user name or role on the AUTHORIZATION clause of the CREATE ROLE statement to indicate the owner of the role. If you omit this clause, the role is owned by the user who creates it.

The CREATE ROLE statement creates a new role in the current database. The role name you specify on this statement must be unique: It can't be the same as another user-defined database role, fixed database role, or database user name. The first statement in this figure, for example, creates a new role named InvoiceEntry.

To grant permissions to a user-defined database role, you use the GRANT statement. The two GRANT statements in this figure, for example, grant the InvoiceEntry role INSERT and UPDATE permissions to the Invoices and InvoiceLineItems tables in the AP database. (You can assume that AP is the current database for these examples.)

The next two statements in this figure assign two users to the new role. That means that these two users now have the INSERT and UPDATE permissions that were assigned to the role. Then, the next statement assigns the new role as a member of the db_datareader role. This role grants any member of the role permission to select data from any user table in the database. Since the member itself is a role, any member of that role now has permission to select data from the database.

The next statement in this figure uses the ALTER ROLE statement to rename the InvoiceEntry role to InvEntry. Then, the last group of statements deletes the two members from this role and then deletes the role.

As you can see, using roles can significantly simplify security management. If you assign roles as members of other roles, however, managing the various roles and permissions can quickly get out of hand. For example, suppose you added a new table to the AP database. Then, the two users that are members of the InvoiceEntry role would automatically be able to select data from that table because the InvoiceEntry role is a member of the db_datareader role. If that's not what you want, you'd need to remove InvoiceEntry from the db_datareader role and then grant SELECT permission to that role for each of the tables in the database that you want the users to have access to. If you plan to assign roles to other roles, then, you'll want to plan it out carefully to avoid having to redesign the security in the future.

The syntax of the CREATE ROLE statement

```
CREATE ROLE role_name [AUTHORIZATION owner_name]
```

The syntax of the DROP ROLE statement

```
DROP ROLE role_name
```

Statements that work with user-defined database roles

A statement that creates a new user-defined database role

```
CREATE ROLE InvoiceEntry;
```

Statements that grant permissions to the new role

```
GRANT INSERT, UPDATE
```

```
ON Invoices
```

```
TO InvoiceEntry;
```

```
GRANT INSERT, UPDATE
```

```
ON InvoiceLineItems
```

```
TO InvoiceEntry;
```

Statements that assign users to the new role

```
ALTER ROLE InvoiceEntry ADD MEMBER JohnDoe;
```

```
ALTER ROLE InvoiceEntry ADD MEMBER SusanRoberts;
```

A statement that assigns the new role to a fixed database role

```
ALTER ROLE db_datareader ADD MEMBER InvoiceEntry;
```

A statement that changes the name of the new role

```
ALTER ROLE InvoiceEntry WITH NAME = InvEntry;
```

Statements that delete the new role

```
ALTER ROLE InvEntry DROP MEMBER JohnDoe;
```

```
ALTER ROLE InvEntry DROP MEMBER SusanRoberts;
```

```
DROP ROLE InvEntry;
```

Description

- You use the CREATE ROLE statement to create a *user-defined database role*. Role names can be up to 128 characters in length and can include letters, symbols, and numbers, but not the backslash (\) character.
- Once you create a database role, you can grant permissions to or revoke permissions from the role. Then, you grant or revoke permissions for every member of the role.
- To add members to a user-defined database role, delete members from the role, or rename the role, you use the ALTER ROLE statement shown in figure 17-15.
- You use the DROP ROLE statement to delete user-defined database roles. You can't delete a fixed database role or the public database role.
- Before you can delete a database role, you must delete all of its members. To find out how to list the members of a role, see figure 17-17.

How to display information about database roles and role members

As you might expect, most systems have many database users and many database roles. Some users belong to several roles, and some roles belong to other roles. For this reason, keeping track of security permissions can be a complex task. Since the Management Studio provides an easy way to examine current role settings, most security managers use this tool rather than using Transact-SQL. However, SQL Server provides some system stored procedures that can be helpful for managing database roles. Figure 17-17 presents two of these procedures.

The `sp_HelpRole` procedure returns information about the database roles defined for the current database. If you code a valid role name as a parameter, this procedure returns information about that one role. Otherwise it returns information about all the roles in the database. That includes both user-defined database roles and fixed database roles. In most cases, you'll use this function just to list the roles in a database, so you'll omit the role name.

The information that's returned by this procedure includes the role name, the role ID, and an indication of whether or not the role is an application role. The role ID is the internal object identification number that's assigned to the role. An application role is a special kind of role that's typically used to provide secure access to an application program rather than a user. You'll learn more about application roles later in this chapter.

The `sp_HelpRoleMember` stored procedure returns information about the current members of a database role. If you include a role name as a parameter, it returns information about the members of that role. Otherwise, it returns information about the members in all the roles in the current database that have at least one member.

In most cases, the information provided by the `sp_HelpRole` and `sp_HelpRoleMember` stored procedures is all you need. If you need additional information, though, you can find it in the `sys.database_principals` and `sys.database_role_members` catalog views. These catalog views are similar to the catalog views that you use to get information about server roles and role members as shown in figure 17-14. To learn more about these catalog views, see the SQL Server documentation.

How to display database role information

The syntax for sp_HelpRole

```
sp_HelpRole [[@rolename = ] 'database_role_name']
```

A statement that lists the roles for the current database

```
EXEC sp_HelpRole;
```

The response from the system

	RoleName	RoleId	IsAppRole
1	public	0	0
2	InvoiceEntry	6	0
3	AppInvoiceQuery	8	1
4	db_owner	16384	0
5	db_accessadmin	16385	0
6	db_securityadmin	16386	0
7	db_ddladmin	16387	0
8	db_backupoperator	16389	0
9	db_datareader	16390	0
10	db_datawriter	16391	0
11	db_denydatareader	16392	0
12	db_denydatawriter	16393	0

How to display database role member information

The syntax for sp_HelpRoleMember

```
sp_HelpRoleMember [[@rolename = ] database_role_name']
```

A statement that lists the members of the InvoiceEntry role

```
EXEC sp_HelpRoleMember InvoiceEntry;
```

The response from the system

	DbRole	MemberName	MemberSID
1	InvoiceEntry	JohnDoe	0xE3BB643D227244AA02DA4213C0C9978
2	InvoiceEntry	MartinRey	0x4333E01D011965459F45A85D3F956780

Description

- To display information about the roles defined in the current database, use the `sp_HelpRole` system stored procedure. If you don't specify a role name on this procedure, information about all of the roles in the database is returned.
- To display information about the members of a database role, use the `sp_HelpRoleMember` system stored procedure. If you don't specify the role name on this procedure, information about the members of all of the roles in the database that have at least one member is returned.
- You can display additional information about database roles using the `sys.database_principals` and `sys.database_role_members` catalog views. These views are similar to the `sys.server_principals` and `sys.server_role_members` views described in figure 17-14.

Figure 17-17 How to display information about database roles and role members

How to deny permissions granted by role membership

A user's permissions include those that are granted explicitly to that user plus permissions that are granted by that user's membership in one or more roles. That means that if you revoke a permission from the user but the same permission is granted by a role to which the user belongs, the user still has that permission. Since this might not be what you want, SQL Server provides a DENY statement that you can use to deny a user permission that's granted by the user's membership in a role. This statement is presented in figure 17-18.

The syntax of the DENY statement is similar to the syntax of the REVOKE statement. To deny object permissions, you specify the permissions you want to deny, the object to which you want to deny permissions, and the users and roles whose permissions you want to deny. To deny permissions to all of the objects in the schema, you specify the permissions you want to deny, the schema that contains the objects to which you want to deny permissions, and the users and roles whose permissions you want to deny. To deny database permissions, you specify the permissions you want to deny and the users and roles whose permissions you want to deny. And to deny server permissions, you specify the permissions you want to deny and the login IDs and roles whose permissions you want to deny.

The two examples in this figure illustrate how this works. The script in the first example adds the user named MartinRey to the InvoiceEntry role. Since InvoiceEntry is a member of the db_datareaders fixed database role, MartinRey can retrieve data from any table in the database. This is illustrated by the successful completion of the SELECT statement that follows, which retrieves data from the GLAccounts table.

The script in the second example uses a DENY statement to deny the user named MartinRey SELECT permission on the GLAccounts table. As a result, when this user executes a SELECT statement against the GLAccounts table, the system responds with an error. That's because the DENY statement specifically denied this user permission to retrieve data from this table even though that permission is granted by the db_datareaders role.

The syntax of the DENY statement for object permissions

```
DENY permission [, ...]
ON [schema_name.]object_name [(column [, ...])]
TO database_principal [, ...]
[CASCADE]
```

The syntax of the DENY statement for schema permissions

```
DENY permission [, ...]
ON SCHEMA :: schema_name
TO database_principal [, ...]
[CASCADE]
```

The syntax of the DENY statement for database permissions

```
DENY permission [, ...]
TO database_principal [, ...]
[CASCADE]
```

The syntax of the DENY statement for server permissions

```
DENY permission [, ...]
TO server_principal [, ...]
[CASCADE]
```

A script that assigns membership to the InvoiceEntry role

```
ALTER ROLE InvoiceEntry ADD MEMBER MartinRey;
```

A SELECT statement entered by the user

```
SELECT * FROM GLAccounts;
```

The response from the system

	AccountNo	AccountDescription
1	100	Cash
2	110	Accounts Receivable
3	120	Book Inventory

A script that denies SELECT permission to GLAccounts

```
DENY SELECT
ON GLAccounts
TO MartinRey;
```

A SELECT statement entered by the user

```
SELECT * FROM GLAccounts;
```

The response from the system

```
Server: Msg 229, Level 14, State 5, Line 1
SELECT permission was denied on object 'GLAccounts', database 'AP', schema 'dbo'.
```

Description

- The permissions granted to individual users are granted by two sources: permissions granted to their user names or login IDs and permissions granted through any roles to which they are members.
- The DENY statement differs from the REVOKE statement in that DENY prevents the permission from being granted by role membership. A denied permission can't be granted by role membership, but a revoked permission can.

Figure 17-18 How to deny permissions granted by role membership

How to work with application roles

An *application role* is a special kind of user-defined database role. Unlike other roles, you can't assign members to an application role. Instead, you activate the role for a connection. Then, the normal security for the login ID that was used to open the connection is replaced by the security that's specified by the application role.

Figure 17-19 presents SQL statements and system stored procedures for working with application roles. To create a new application role, you use the CREATE APPLICATION ROLE statement. This statement requires a role name and a password. The first statement in this figure, for example, creates an application role named AppInvoiceQuery that has a password of "appqrypw". You can also specify a default schema for an application role.

After you create an application role, you can use it in GRANT, REVOKE, or DENY statements just as you would any other role. The second statement in this figure, for example, grants the application role permission to retrieve data from the Invoices table.

To activate an application role, you execute the sp_SetAppRole procedure. Once activated, the connection is granted the permissions associated with the application role instead of the permissions associated with the login ID. Since the connection takes on an entirely new set of permissions, it's almost as if the user logged off and then logged back on under a different login ID.

You can see how this works in the script in this figure. First, assume that the login ID that was used to log on to the server doesn't have SELECT permission for the Invoices table. For this reason, the first SELECT statement in this script fails and returns an error message. Next, the script activates the AppInvoiceQuery application role. Because this role has permission to select data from the Invoices table, the SELECT statement that follows now succeeds.

Notice that you can also create a cookie when you activate an application role. This cookie is stored in an OUTPUT parameter that must be defined as VARBINARY(8000). If you create a cookie, you can use the sp_UnsetAppRole stored procedure to deactivate the application role. Otherwise, the role remains in effect until the connection is closed.

Application roles are intended for use by application programs that manage their own security. Typically, an application like this will open a limited number of connections to a database and then share those connections among many application users. Then, the application role controls the application's access to the database, and the application controls the users that are allowed to use the connections it establishes.

You can also use application roles to provide for more flexible security. For example, suppose a user needs to access a database both through the Management Studio and through an application. Also suppose that the user needs broader permissions to use the application than you want to give him through the Management Studio. To do that, you could assign the user standard permissions through his login ID and role memberships, and you could give the application enhanced permissions through an application role.

SQL statements for working with application roles

The syntax of the CREATE APPLICATION ROLE statement

```
CREATE APPLICATION ROLE role_name WITH PASSWORD = 'password'
[, DEFAULT_SCHEMA = schema_name]
```

The syntax of the DROP APPLICATION ROLE statement

```
DROP APPLICATION ROLE role_name
```

System stored procedures for working with application roles

The syntax for sp_SetAppRole

```
sp_SetAppRole [@rolename = ] 'role_name',
[@password = ] 'password'
[, [ @fCreateCookie = ] {True|False}]
[, [ @cookie = ] @cookie OUTPUT]
```

The syntax for sp_UnsetAppRole

```
sp_UnsetAppRole @cookie
```

Statements that create an application role and give it permissions

```
CREATE APPLICATION ROLE AppInvoiceQuery WITH PASSWORD = 'AppQry2720';

GRANT SELECT
ON Invoices
TO AppInvoiceQuery;
```

A script that tests the application role

```
SELECT * FROM Invoices;
EXEC sp_SetAppRole AppInvoiceQuery, AppQry2720;
SELECT * FROM Invoices;
```

The response from the system

```
Server: Msg 229, Level 14, State 5, Line 1
SELECT permission was denied on object 'Invoices', database 'AP', schema 'dbo'.
```

InvoicelD	VendorID	InvoiceNumber	InvoiceDate	InvoiceTotal	PaymentTotal	CreditTotal
1	1	989319-457	2019-10-08	3813.33	3813.33	0.00
2	2	263253241	2019-10-10	40.20	40.20	0.00

Description

- An *application role* is a special type of user-defined database role. It can't contain any members, but it's activated when a connection executes the `sp_SetAppRole` system stored procedure.
- Once the connection activates an application role, the normal security for the login ID set by the permissions for the ID and its roles is ignored. Instead, the connection assumes a new security profile as defined by the permissions for the application role.
- Once a connection activates an application role, the application role remains in effect until the connection is closed or until the `sp_UnsetAppRole` procedure is executed. To use `sp_UnsetAppRole`, you must create a cookie when you execute `sp_SetAppRole`.
- Application roles are typically used by application programs that manage their own security. Then, those programs can control the users that can log on to the server.

Figure 17-19 How to work with application roles

How to manage security using the Management Studio

Now that you understand how SQL Server security works, you're ready to learn how to manage security using the Management Studio. The topics that follow present the basic skills for doing that. If you want to learn additional skills, you shouldn't have any trouble doing that on your own.

How to work with login IDs

Figure 17-20 presents the General page of the Login - New dialog box. You use this page to specify the basic settings for a new login. To start, you select which type of authentication you want to use. If you select Windows authentication, you can click the Search button to the right of the Login Name box to select a domain and a user or group. Alternatively, you can enter a domain name and a user or group name in the Login Name box.

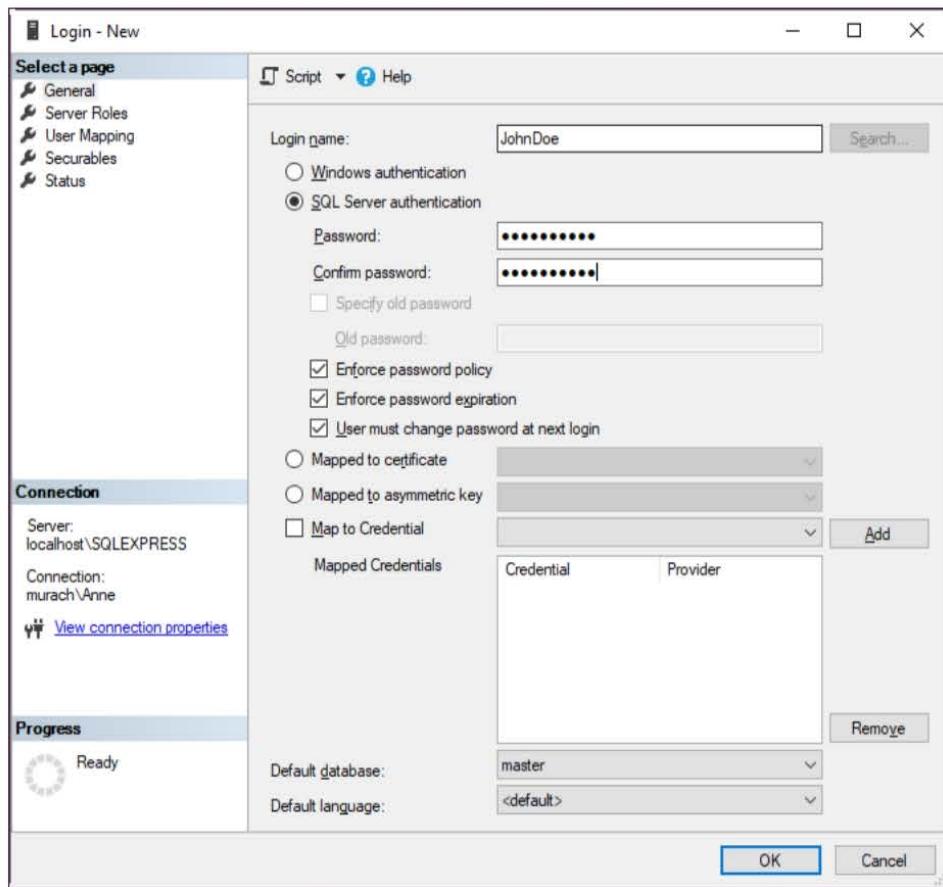
If you select SQL Server authentication, you must enter the new login name in the Login Name box. In addition, you must enter a password for the user in both the Password and Confirm Password boxes. You can also set the three password options shown here. These options are equivalent to the CHECK_POLICY, CHECK_EXPIRATION, and MUST_CHANGE options that you can include on the CREATE LOGIN statement.

In addition to setting the authentication mode and password options, you can use this page to set the default database and the default language for the user. If you don't select another database, the master database is used. And if you don't select a specific language, the default language for the server is used.

After you create a login ID, you can modify it using the Login Properties dialog box. The General page of this dialog box is almost identical to the General page of the Login - New dialog box shown in this figure. The main difference is that you can't change the type of authentication that's used from this page. To do that, you have to delete the login ID and create a new one with the authentication you want.

The Status page of the Login Properties dialog box lets you grant or deny a login ID permission to connect to SQL Server. It also lets you enable or disable a login ID. Although these options are similar, they differ in how they're implemented.

The General page of the Login - New dialog box



Description

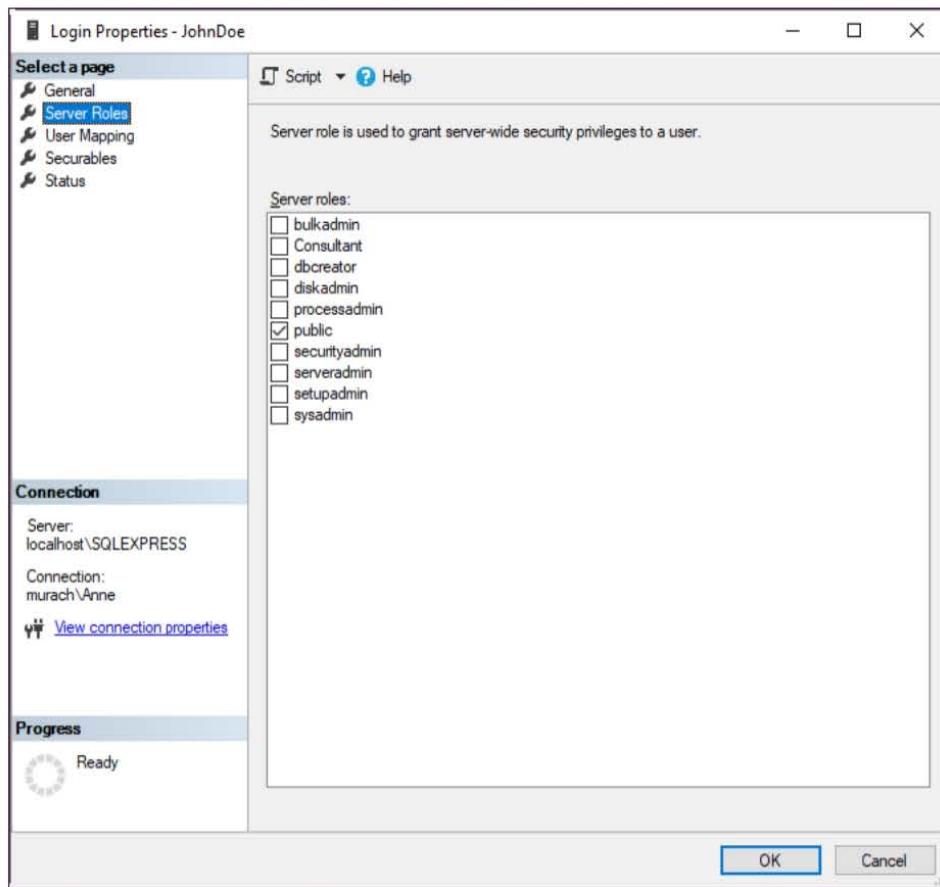
- To create a new login ID, expand the Security folder for the server in the Object Explorer. Then, right-click the Logins folder and select New Login to display the Login - New dialog box.
- Select the type of authentication you want to use and then specify the appropriate settings. The settings that are available depend on whether you select Windows or SQL Server authentication.
- To modify a login ID, right-click the ID in the Object Explorer and select Properties to display the Login Properties dialog box. The General page of this dialog box lets you change all the settings for the login ID except for the login name and the authentication mode.
- You can use the options on the Status page of the Login Properties dialog box to deny or grant a login ID permission to connect to SQL Server. You can also disable or enable a login ID.
- To delete a login ID, right-click the ID in the Object Explorer and select Delete to display the Delete Object dialog box. Click the OK button to delete the login ID.

Figure 17-20 How to work with login IDs

How to work with the server roles for a login ID

Figure 17-21 shows how to work with the server roles for a specific login ID. To do that, you use the Server Roles page of the Login Properties dialog box. This page lists all of the server roles and lets you select the ones you want the login ID assigned to. Note that this page lists user-defined server roles as well as fixed server roles. In this figure, for example, you can see the user-defined role named Consultant.

The dialog box for working with server roles



Description

- To work with the server roles for a login ID, right-click the ID in the Object Explorer and select Properties to display the Login Properties dialog box. Then, display the Server Roles page. The roles that the login ID is currently assigned to are checked in the list that's displayed.
- To add or remove a login ID from a server role, select or deselect the role.

Figure 17-21 How to work with the server roles for a login ID

How to assign database access and roles by login ID

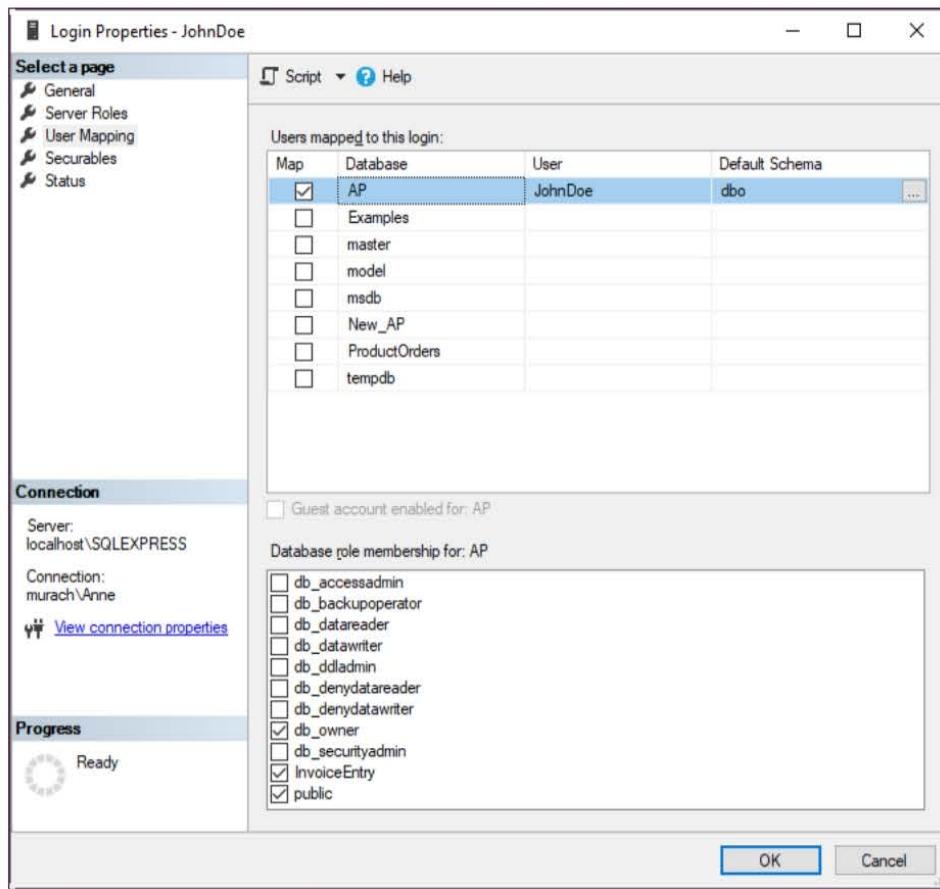
Figure 17-22 presents the User Mapping page of the Login Properties dialog box. This page lists all of the databases on the server and all of the database roles defined for the highlighted database. You can use this dialog box to grant database access to a login ID and to assign a login ID membership in one or more database roles.

To grant or revoke database access, simply select or deselect the check box to the left of the database name. If you grant a login ID access to a database, the name of the associated database user is displayed in the User column, and the default schema for the user is displayed in the Default Schema column. If no user is associated with the login ID, the user name is set to the login name by default and no default schema is specified. If that's not what you want, you can enter a different name for the user and select a schema. Then, when you click the OK button, a new user is created with the name and default schema you specified. This is a quick and easy way to create a user at the same time that you grant database access.

If you want to create a user before granting access to one or more databases, you can do that using the Database User - New dialog box. To display this dialog box, expand the database in the Object Explorer, right-click on the Users folder, and select New User.

After you grant a user access to a database, you can add that user as a member of any of the database roles defined for the database. That includes both the fixed database roles and any user-defined roles. To do that, just highlight the database and then select the check boxes to the left of the database roles to add the user to those roles.

The User Mapping page of the Login Properties dialog box



How to grant or revoke database access for a user

- To grant or revoke access to a database, display the User Mapping page of the Login Properties dialog box. Then, select or deselect the Map check box for that database.
- If you grant access to a login ID that's not associated with a database user, a user is created automatically when you complete the dialog box. By default, the user name is set to the login ID, but you can change this name in the User column.
- By default, the default schema for a new user is set to dbo. If you want to specify a different schema, click the button with the ellipsis on it in the Default Schema column and select the schema from the dialog box that's displayed.

How to add or remove a user from a database role

- If the user has access to a database, you can add or remove the user from the database roles for that database. To do that, highlight the database to display the database roles in the lower portion of the dialog box. Then, select or deselect the roles.

Figure 17-22 How to assign database access and roles by login ID

How to assign user permissions to database objects

To set the permissions for a user, you use the Securables page of the Database User dialog box shown in figure 17-23. The top of this dialog box lists the securables for which you can set permissions. When you first display this page, this list is empty. Then, you can use the Search button to add the securables you want to work with. The dialog box that's displayed when you click this button lets you add specific database objects, all the objects of one or more types, or all the objects in a selected schema. In this figure, three tables in the dbo schema are included in the list.

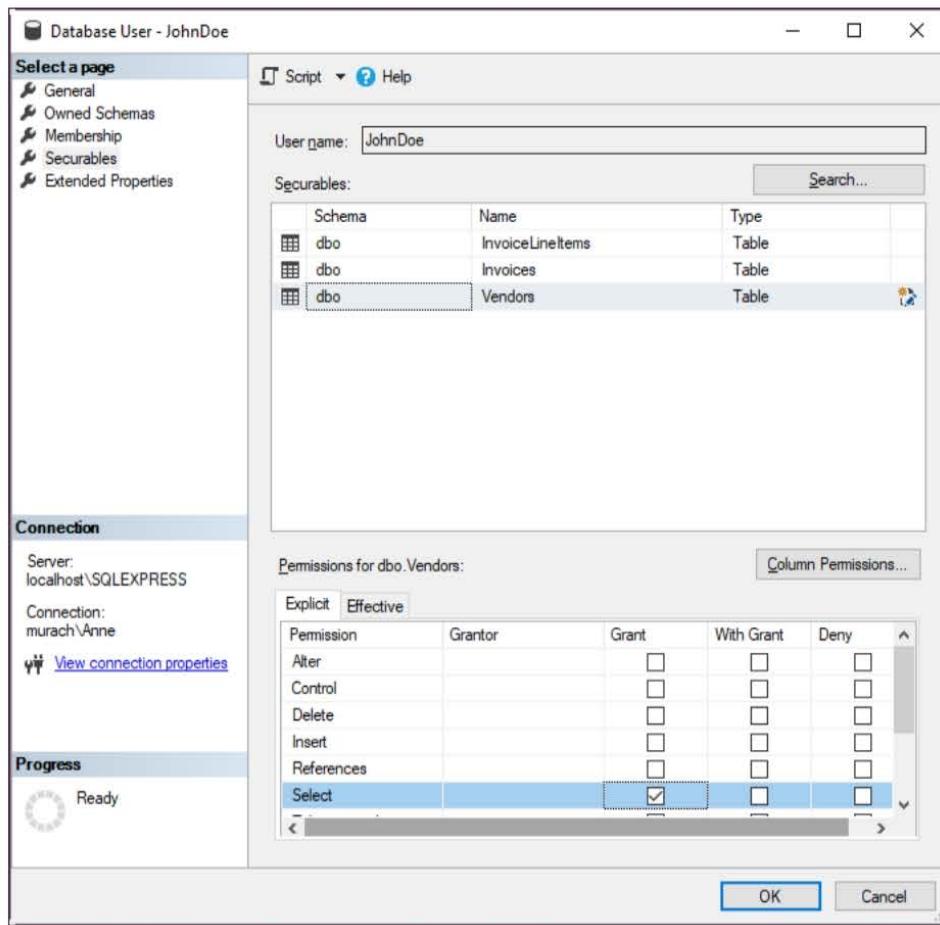
Once you add objects to the list of securables, the permissions for the highlighted securable are displayed in the list at the bottom of the page. Then, to grant the user a permission, you can select the Grant option for that permission. When you do that, you can also select the With Grant option to give the user permission to grant the same permission to other users.

If the user already has permission to an object, you can revoke that permission by deselecting the Grant option. You can also deny the user a permission that's granted by membership in a role by selecting the Deny option for the permission.

If you highlight a table, view, or table-valued function and then highlight the Select, References, or Update permission, you'll notice that the Column Permissions button becomes available. If you click on this button, a dialog box is displayed that lists the columns in that table, view, or function. You can use this dialog box to set the permissions for individual columns.

After you set the explicit permissions for a user from this dialog box, you might want to know what the user permissions are when the explicit permissions are combined with any permissions granted by role membership. To do that, you can highlight a securable and then display the Effective tab. This tab lists the permissions the user has for the securable, including column permissions for a table, view, or table-valued function.

The Securables page of the Database User dialog box



Description

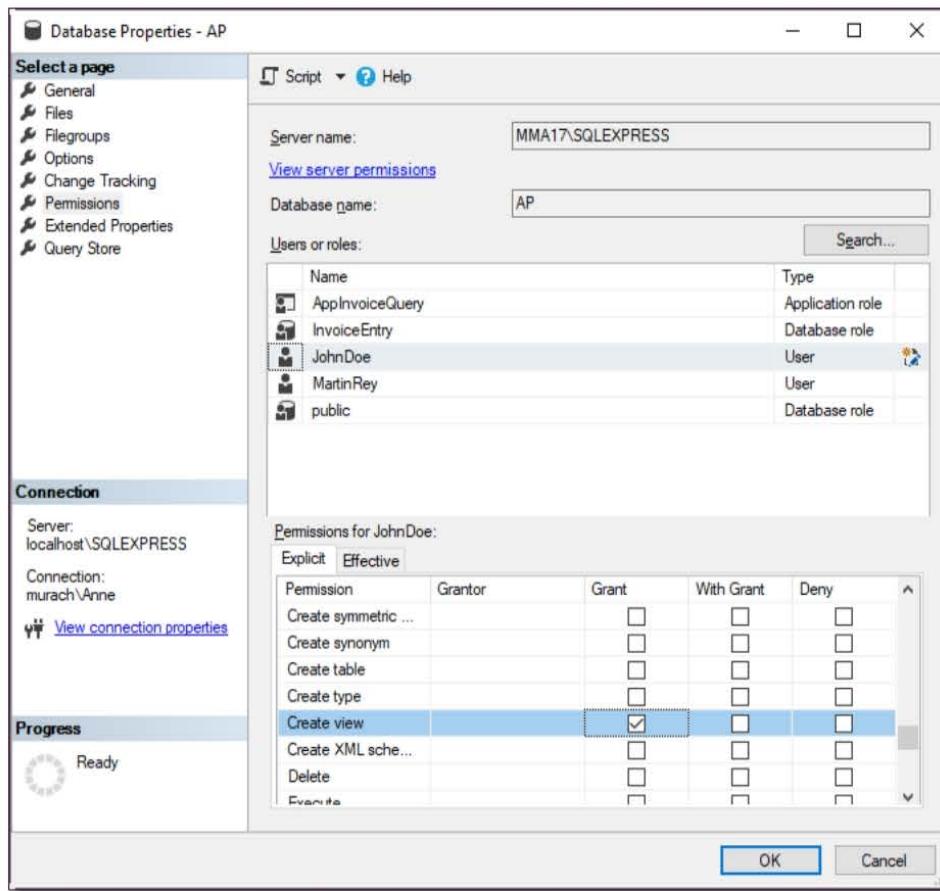
- To display the Database User dialog box, expand the database in the Object Explorer, expand the Security and Users folders, right-click the user name, and select Properties.
- To work with the permissions for a user, display the Securables page. Then, click the Search button and use it to select the securables you want to work with.
- To grant the user permission to a securable, select the securable and then select the Grant option for the permission. You can also select the With Grant option.
- To revoke the user permission to a securable, deselect the Grant option.
- To deny the user permission to a securable, select the Deny option.
- To set the permissions for the columns in a table, view, or table-valued function, highlight the securable and permission. Then, click the Column Permissions button.
- The permissions that are available change depending on the type of securable that's selected.
- To display the combination of the permissions granted with this dialog box and the permissions granted through roles to a securable, display the Effective tab.

Figure 17-23 How to assign user permissions to database objects

How to work with database permissions

To work with the database permissions for the users and roles in a database, you use the Permissions page of the Database Properties dialog box shown in figure 17-24. As you can see, this page is similar to the Securables page of the Database User dialog box you saw in figure 17-23. Instead of listing securables, however, it lists users and roles. And instead of listing object permissions, it lists database permissions. Then, you can select a user or role and grant, revoke, or deny permissions to perform specific database operations.

The Permissions page of the Database Properties dialog box



Description

- To display the permissions for a database, right-click the database in the Object Explorer, select Properties to display the Database Properties dialog box, and display the Permissions page.
- By default, all database users except for guest are included in the Users or roles list. To add the guest user or any system or user-defined roles, click the Search button and select the users and roles from the dialog box that's displayed.
- To grant a user or role permission to perform a database operation, select the user or role and then select the Grant option for the permission. You can also select the With Grant option to allow the user or role to grant the permission to other users and roles.
- To revoke a user or role permission to perform a database operation, deselect the Grant option.
- To deny the user permission to a database operation, select the Deny option.

Figure 17-24 How to work with database permissions

Perspective

Although managing security on a server can be complex, SQL Server provides useful tools to simplify the job. In this chapter, you've learned how to manage security for your server and database using both Transact-SQL and the Management Studio. Once you're familiar with both of these techniques, you can use the one that's easiest for the security task at hand.

Unfortunately, the techniques presented in this chapter don't secure your data when it's "at rest" on a hard drive that contains the data files for the database. In that case, if a thief steals the database files, he or she can attach the data files to a different server and gain access to the data. To close this security hole, you can encrypt the data files and store the encryption key in a different location. This encrypts every column of every table. That way, even if the data files are stolen, the thief won't be able to open them without the key. To encrypt data files, you can use *Transparent Data Encryption (TDE)*. TDE makes it easy to encrypt the entire database without affecting existing applications. For more information about TDE, you can refer to the SQL Server documentation.

However, TDE doesn't protect your data when it's "alive" in a database that's running on a server or when it's "in transit" between the client and the server. To do that, SQL Server 2016 introduced the *Always Encrypted* feature. This feature encrypts data for sensitive columns such as columns that store credit card numbers. When you use it, the client application uses an enhanced library to encrypt this data when it's "in transit". Then, this encrypted data is used by the database when it's "alive" and "at rest". As a result, the data is always encrypted. Since the encryption key is stored on the client, the unencrypted data is only available to the users of the client application, not to database administrators. This eases security concerns when storing data in a database that's running in the cloud. For more information about Always Encrypted, you can refer to the SQL Server documentation.

Terms

login ID	Windows authentication
permissions	SQL Server authentication
object permissions	strong password
schema permissions	scope qualifier
database permissions	fixed role
server permissions	fixed server role
role	user-defined server role
group	fixed database role
principal	user-defined database role
securable	application role
authentication mode	

Exercises

1. Write a script that creates a user-defined database role named PaymentEntry in the AP database. Give UPDATE permission to the new role for the Invoices table, UPDATE and INSERT permission for the InvoiceLineItems table, and SELECT permission for all user tables.
2. Write a script that (1) creates a login ID named “AAaron” with the password “AAar99999”; (2) sets the default database for the login to the AP database; (3) creates a user named “Aaron” for the login; and (4) assigns the user to the PaymentEntry role you created in exercise 1.
3. Write a script that creates four login IDs based on the contents of a new table named NewLogins:

```
CREATE TABLE NewLogins  
  (LoginName varchar(128));  
INSERT NewLogins  
  VALUES ('BBrown'), ('CChaplin'), ('DDyer'), ('EEbbers');
```

Use dynamic SQL and a cursor to perform four actions for each row in this table: (1) create a login with a temporary password that's based on the first four letters of the login name followed by “99999”; (2) set the default database to the AP database; (3) create a user for the login with the same name as the login; and (4) assign the user to the PaymentEntry role you created in exercise 1.

4. Using the Management Studio, create a login ID named “FFalk” with the password “FFal99999,” and set the default database to the AP database. Then, grant the login ID access to the AP database, create a user for the login ID named “FFalk”, and assign the user to the PaymentEntry role you created in exercise 1.

Note: If you get an error that says “The MUST_CHANGE option is not supported”, you can deselect the “Enforce password policy” option for the login ID.

5. Write a script that removes the user-defined database role named PaymentEntry. (Hint: This script should begin by removing all users from this role.)
6. Write a script that (1) creates a schema named Admin, (2) transfers the table named ContactUpdates from the dbo schema to the Admin schema, (3) assigns the Admin schema as the default schema for the user named Aaron that you created in exercise 2, and (4) grants all standard privileges except for REFERENCES and ALTER to AAaron for the Admin schema.

18

How to work with XML

In this chapter, you'll learn how to use SQL Server to work with XML data. That includes using the xml data type, using an XML schema to validate the data that's stored in an xml type, and converting relational data to XML and XML to relational data. These features make it easier 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
Perspective	616

An introduction to XML

Before you learn how to use SQL Server to work with XML, you need to understand some basic XML concepts. In particular, you need to understand how XML can be used to structure data, and you need to understand how an XML schema can be used to validate XML data.

An XML document

XML (Extensible Markup Language) can be used to create an *XML document* that contains data that has been structured with *XML tags*. For example, figure 18-1 shows an XML document that contains data about an event that caused a DDL trigger to fire. You learned how to return an XML document like this one in chapter 15 using the EVENTDATA function within a DDL trigger.

Within an XML document, an *element* begins with a *start tag* and ends with an *end tag*. In this figure, for example, the <EventType> tag marks the start of the EventType element, and the </EventType> tag marks the end of this element.

Within a start tag, one or more *attributes* can be coded. In this figure, for example, the start tag for the SetOptions element contains five attributes: ANSI_NULLS, ANSI_NULL_DEFAULT, ANSI_PADDING, QUOTED_IDENTIFIER, and ENCRYPTED. Here, each attribute consists of an attribute name, an equal sign, and a string value in quotes.

Although values can be assigned to attributes, a value can also be coded between the start and end tags for an element. That's the case with most of the elements in this figure. The EventType element, for example, contains the string value "CREATE_TABLE."

Elements can also contain other elements. An element that's contained within another element is known as a *child element*. Conversely, the element that contains a child element is known as the child's *parent element*. In this figure, for example, the SetOptions element is a child element of the TSQLCommand element, and the TSQLCommand element is the parent element of the SetOptions element. Although it's not shown here, a child element can also repeat within a parent element.

The highest-level element in an XML document is known as the *root element*. In this figure, the EVENT_INSTANCE element is the root element. A well-formed XML document can have only one root element.

An XML document

```
<EVENT_INSTANCE>
  <EventType>CREATE_TABLE</EventType>
  <PostTime>2020-02-10T12:38:23.147</PostTime>
  <SPID>54</SPID>
  <ServerName>MMA17\SQLEXPRESS</ServerName>
  <LoginName>murach\anne</LoginName>
  <UserName>dbo</UserName>
  <DatabaseName>AP</DatabaseName>
  <SchemaName>dbo</SchemaName>
  <ObjectName>VendorsTest</ObjectName>
  <ObjectType>TABLE</ObjectType>
  <TSQLCommand>
    <SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON" ANSI_PADDING="ON"
      QUOTED_IDENTIFIER="ON" ENCRYPTED="FALSE" />
    <CommandText>
      CREATE TABLE VendorsTest
      (VendorID int, VendorName varchar(50));
    </CommandText>
  </TSQLCommand>
</EVENT_INSTANCE>
```

Description

- *XML (Extensible Markup Language)* is used to structure data using *XML tags*. An XML tag begins with `<` and ends with `>`.
- An *XML document* contains data that has been structured with XML tags.
- An *element* begins with a *start tag* and ends with an *end tag*. The start tag provides the name of the element and can contain one or more *attributes*. An attribute consists of an attribute name, an equal sign, and a string value in quotes. The end tag repeats the name, prefixed with a slash (`/`).
- Text can also be coded between an element's start and end tags. This text is referred to as the element's *content*.
- Elements can contain other elements. An element that's contained within another element is known as a *child element*. The element that contains a child element is known as the child's *parent element*. Child elements can also repeat within a parent element.
- The highest-level element in an XML document is known as the *root element*. An XML document can have only one root element.
- The EVENTDATA function described in chapter 15 returns an `xml` data type that contains an XML document like the one shown in this figure.

Figure 18-1 An XML document

An XML schema

The *XML Schema Definition (XSD)* language can be used to define an *XML schema*, which is a set of rules that an XML document must follow to be valid. Although there are several languages for defining XML schemas, the XSD language is the only one that you can use with SQL Server. Figure 18-2 shows an XML schema that can be used to validate the XML document shown in figure 18-1. In addition, you may notice that the schema itself is an XML document.

To start, this XML schema uses the schema element to specify some attributes that apply to the entire schema. That includes the xmlns attribute that defines the prefix for the XML namespace that's used throughout the rest of the schema to qualify the name of each element. Then, this schema defines each of the elements and attributes that are used by the XML document shown in figure 18-1. This definition specifies whether an element can contain other elements, the sequence of the elements, and the name and data type of each element.

Unfortunately, the details of the XSD language are beyond the scope of this book. As a result, the examples in this chapter only use the XML schema presented in this figure. Fortunately, if you're working with an industry standard XML document, an XSD has probably already been created for it. As a result, you may be able to get the XSD from a colleague, or you may be able to find the XSD by searching the Internet.

If you need to create an XML schema for an XML document, you can use the Management Studio's XML Editor to generate one as described in figure 18-4. Then, if necessary, you can edit the generated XML schema so it's appropriate for your XML document. To do that, however, you may need to learn more about the XSD language by searching the Internet or by getting a book about working with XML.

An XML Schema Definition (XSD)

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="EVENT_INSTANCE">
<xs:complexType>
<xs:sequence>
<xs:element name="EventType" type="xs:string" />
<xs:element name="PostTime" type="xs:dateTime" />
<xs:element name="SPID" type="xs:unsignedByte" />
<xs:element name="ServerName" type="xs:string" />
<xs:element name="LoginName" type="xs:string" />
<xs:element name="UserName" type="xs:string" />
<xs:element name="DatabaseName" type="xs:string" />
<xs:element name="SchemaName" type="xs:string" />
<xs:element name="ObjectName" type="xs:string" />
<xs:element name="ObjectType" type="xs:string" />
<xs:element name="TSQLCommand">
<xs:complexType>
<xs:sequence>
<xs:element name="SetOptions">
<xs:complexType>
<xs:attribute name="ANSI_NULLS"
type="xs:string" use="required" />
<xs:attribute name="ANSI_NULL_DEFAULT"
type="xs:string" use="required" />
<xs:attribute name="ANSI_PADDING"
type="xs:string" use="required" />
<xs:attribute name="QUOTED_IDENTIFIER"
type="xs:string" use="required" />
<xs:attribute name="ENCRYPTED"
type="xs:string" use="required" />
</xs:complexType>
</xs:element>
<xs:element name="CommandText" type="xs:string" />
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Description

- The *XML Schema Definition (XSD)* language can be used to define an *XML schema*, which is a set of rules that an XML document must follow to be valid.
- You may be able to find an XSD for some types of XML documents by searching the Internet. If not, you can use the Management Studio's XML Editor to generate an XSD for an XML document as described in figure 18-4. The details for working with the XSD language are beyond the scope of this book.
- SQL Server can use the XML schema shown in this figure to validate the XML document shown in figure 18-1.

Figure 18-2 An XML schema

How to work with the xml data type

Now that you understand how XML works, you're ready to learn how to work with the xml data type. To get started quickly, the following topics show how to use the xml type without an XML schema. Since that means that the data that's stored in the xml type isn't validated, this data is known as *untyped XML*.

How to store data in the xml data type

Figure 18-3 shows how to store XML data in the xml type. For the most part, these examples show that you can use the xml data type just as you use most other SQL Server data types. In particular, you can use it as the type for a column in a table or a variable in a script, procedure, function, or trigger.

The first example shows how you can use the xml type to specify the data type for a column in a table. Here, the CREATE TABLE statement is used to create a table named DDLActivityLog that has two columns. The first column, named EventID, is an identity column that stores an int value. The second column, named EventData, stores XML data.

The second example shows a DDL trigger that inserts XML data into the second column of the DDLActivityLog table. Since a trigger like this one was described in chapter 15, you should understand that this trigger fires any time a CREATE TABLE or DROP TABLE statement is executed on the current database. After the AS keyword, the first statement declares a variable named @EventData with the xml type. Then, the second statement uses the EVENTDATA function to store an XML document like the one shown in figure 18-1 in the @EventData variable. Finally, the third statement uses an INSERT statement to insert a row with the data in the @EventData variable into the DDLActivityLog table.

The third example shows a CREATE TABLE statement that fires the trigger in the second example. This statement creates a table named VendorsTest that has two columns.

The fourth example shows a SELECT statement that retrieves all rows and columns from the DDLActivityLog table. The result set contains a single row that was inserted when the VendorsTest table was created. The first column is the identity value that's automatically generated, and the second column is the XML document that was returned by the EVENTDATA function.

The fifth and sixth examples show two INSERT statements that can be used to insert a row into the DDLActivityLog table. These statements show that SQL Server can implicitly cast a string to the xml type. As a result, when you're working with untyped XML, it's possible to insert any string into the xml type, regardless of whether the string contains well-formed XML. If you want SQL Server to enforce a specified schema for an xml type, you can use the techniques that are described later in this chapter.

A log table with a column of the xml data type

```
CREATE TABLE DDLActivityLog
(EventID int NOT NULL IDENTITY PRIMARY KEY,
EventData xml NOT NULL);
```

A trigger that inserts an XML document into the xml column

```
CREATE TRIGGER Database_CreateTable_DropTable
ON DATABASE
AFTER CREATE_TABLE, DROP_TABLE
AS
DECLARE @EventData xml;
SELECT @EventData = EVENTDATA();
INSERT INTO DDLActivityLog VALUES (@EventData);
```

A CREATE TABLE statement that fires the trigger

```
CREATE TABLE VendorsTest
(VendorID int, VendorName varchar(50));
```

A SELECT statement that retrieves data from the table

```
SELECT * FROM DDLActivityLog;
```

The result set

	EventID	EventData
1	1	<EVENT_INSTANCE><EventType>CREATE_TABLE</EventType><PostTime>2020-02-10T15...

An INSERT statement that inserts a row into the table

```
INSERT INTO DDLActivityLog VALUES ('<root><element1>test</element1></root>');
```

Another INSERT statement that inserts a row into the table

```
INSERT INTO DDLActivityLog VALUES ('this is not xml');
```

Description

- You can use the xml data type just as you use most other SQL Server data types. In particular, you can use it as the type for a column in a table or a variable in a script, procedure, function, or trigger.
- Within a DDL trigger, the EVENTDATA function returns an XML document like the one shown in figure 18-1.
- The xml type is implicitly cast to a string when necessary and a string is implicitly cast to the xml type.

Figure 18-3 How to store data in the xml data type

How to work with the XML Editor

When you use the Query Editor to run a query that returns an xml type, the Management Studio displays the result set as it normally does. However, the XML data is displayed in blue with underlining to indicate that it is a link. If you want to view the complete XML data, you can click on this link. Then, the XML data will be displayed in the Management Studio's *XML Editor* as shown in figure 18-4.

Although the XML Editor works much like the Query Editor, its IntelliSense and color coding are designed to work with XML. In addition, you can use the tree that's displayed to the left of the XML to collapse or expand parent elements. As a result, it's easy to edit the XML whenever that's necessary.

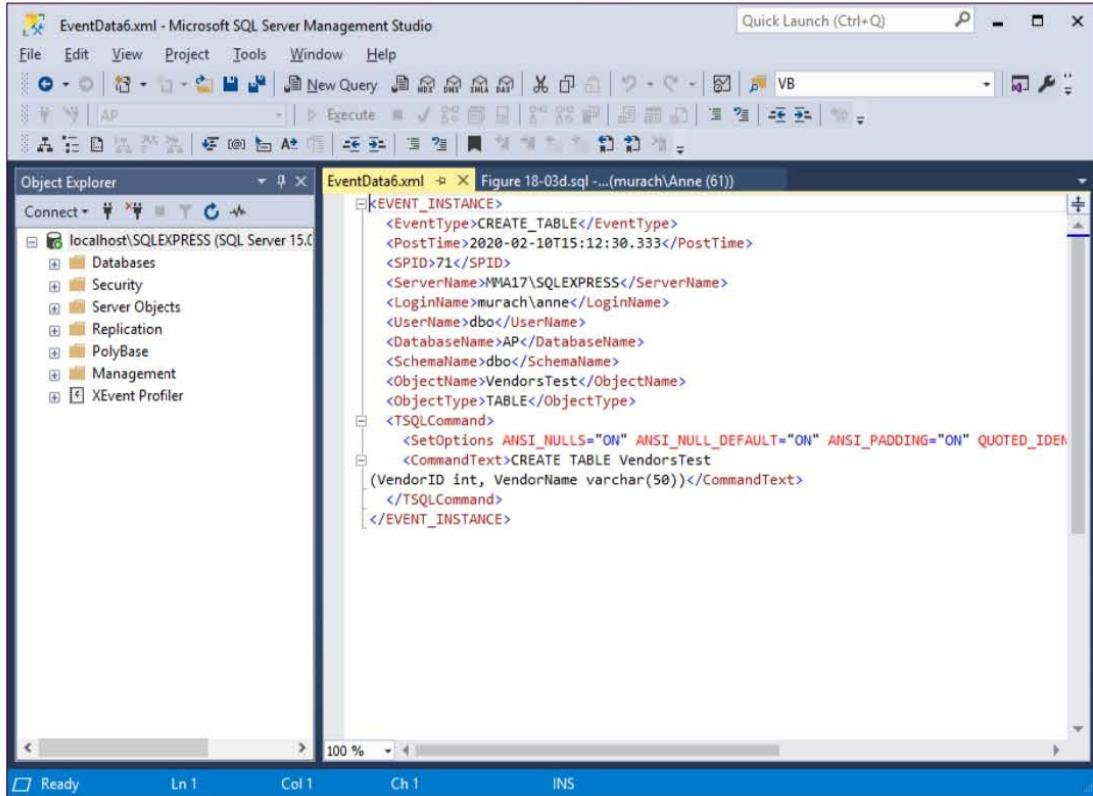
Once you display an XML document in the XML Editor, you can generate an XML schema for the document by clicking on the Create Schema button in the XML Editor toolbar. If this toolbar isn't displayed, you can display it as described in this figure. When you click the Create Schema button, the XML Editor will generate the XSD for the current XML document and display it in a new window. Then, you can add this schema to the database so you can use it to validate other instances of the current XML document. You'll learn how to add an XML schema to a database later in this chapter.

You can also use the XML Editor to work with existing XML files and related files such as XSD files. To do that, just use the Management Studio to open these files. When you do, the Management Studio will automatically display the file in the XML Editor.

A result set that returns XML data

EventID	EventData
1	<EVENT_INSTANCE><EventType>CREATE TABLE</EventType><PostTime>2020-02-10T15...

An xml data type displayed in the Management Studio's XML Editor



Description

- To view XML in the *XML Editor*, run a query that returns the XML that you want to view and then click on the cell that contains the XML. When you do, the XML will be displayed in the XML Editor.
- To work with the data in the XML Editor, you can display the XML Editor toolbar. To do that, right-click on a blank space in the toolbar area and select XML Editor.
- To create an XML Schema Definition for an XML document, display the XML document in the XML Editor and then click on the Create Schema button in the XML Editor toolbar.

Figure 18-4 How to work with the XML Editor

How to use the methods of the xml data type

After you store data in the xml type, you can use the five *methods* shown in figure 18-5 to work with that data. In case you're not familiar with methods, you can use them to perform operations on objects. In this case, you can use them to perform operations on an xml data type. To call a method for an xml type, you code the column or variable name that holds the type, followed by a period, followed by the name of the method, followed by a set of parentheses. Within the parentheses, you code the arguments that are required for the method.

Of the five methods shown here, four take a string argument that specifies an *XQuery*. XQuery is a language that's designed to query an XML document. The only method that doesn't take an XQuery string as an argument is the *modify()* method. This method takes a string argument that specifies an *XML Data Manipulation Language (XML DML)* statement. XML DML is a language that's designed to insert, update, or delete nodes from an XML document.

The first example shows how to use the *query()* method to return an xml type that contains the *SetOptions* element. To accomplish this, the XQuery argument specifies the root element of the XML document, followed by the *TSQLCommand* element, followed by the *SetOptions* element. Note that since the *TSQLCommand* element contains a single *SetOptions* element, it's not necessary to code an attribute number. Because of that, you don't need to enclose the XQuery path in parentheses. However, if two or more *SetOptions* elements were included within the *TSQLCommand* element and you wanted to retrieve only the first element, you could do that by enclosing the XQuery path in parentheses and using square brackets to specify the element number like this:

```
'(/EVENT_INSTANCE/TSQLCommand/SetOptions)[1]'
```

You can use the same coding technique with attributes, although a well-formed XML document shouldn't have more than one attribute with the same name.

The second example shows how to use the *exist()* method to return an int value that indicates whether the element or attribute specified by the XQuery exists and contains data. To start, this example declares a variable of the xml type and uses a *SELECT* statement to store an XML document in this variable. Then, it uses an IF statement to check if the variable has an *EventType* element that contains data. To accomplish this, the IF statement checks if the int value returned by the *exist()* method is equal to 1. If so, it prints a message that indicates that the *EventType* element exists and contains data.

If you want to check an attribute instead of an element, you can use the same skills. However, you must prefix the name of the attribute with an at sign (@) like this:

```
IF @EventData.exist(
    '/EVENT_INSTANCE/TSQLCommand/SetOptions/@ANSI_NULLS') = 1
```

The methods of the xml type

Method	Description
<code>query(XQuery)</code>	Performs an XQuery and returns an xml type that contains the XML fragment specified by the XQuery.
<code>exist(XQuery)</code>	Returns a value of 1 if the XQuery returns a result set. Otherwise, returns a value of 0.
<code>value(XQuery, SqlType)</code>	Performs an XQuery and returns a scalar value of the specified SQL data type.
<code>modify(XML_DML)</code>	Uses an XML DML statement to insert, update, or delete nodes from the current xml type.
<code>nodes(XQuery) AS Table(Column)</code>	Splits the nodes of the current xml data type into rows. You can often use the OPENXML statement described in figure 18-12 to achieve similar results.

The simplified XQuery syntax

```
(/rootElement/element1/element2/@attribute) [elementOrAttributeName]
```

A SELECT statement that uses the query() method

```
SELECT EventData.query('/EVENT_INSTANCE/TSQLCommand/SetOptions')
    AS SetOptions
FROM DDLActivityLog
WHERE EventID = 1;
```

The XML data that's returned

```
<SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON" ANSI_PADDING="ON"
    QUOTED_IDENTIFIER="ON" ENCRYPTED="FALSE" />
```

A script that uses the exist() method

```
DECLARE @EventData xml;
SELECT @EventData = EventData
FROM DDLActivityLog
WHERE EventID = 1;

IF @EventData.exist('/EVENT_INSTANCE/EventType') = 1
    PRINT 'The EventType element exists and contains data.';


```

The response from the system

```
The EventType element exists and contains data.
```

Rules for coding an XQuery

- Start each XQuery with a front slash (/).
- Use a front slash (/) to separate elements and attributes.
- Use an at symbol (@) to identify attributes.
- When necessary, use square brackets ([]) to specify an element or attribute instance.
- If you specify an element or attribute instance, you must code parentheses around the path specification for the element or attribute. Otherwise the parentheses are optional.

Figure 18-5 How to use the methods of the xml data type (part 1 of 2)

The third example shows how to use the value() method to return the value of the specified element or attribute. To do that, you use the first argument of the value() method to specify the XQuery string for the element or attribute. Then, you use the second argument to specify the SQL Server data type that you want to return. In this figure, for example, the varchar data type is used to store the data for the EventType element and for the ANSI_NULLS attribute of the SetOptions element. Since the value() method returns a single value, this method requires its XQuery argument to specify the element or attribute number within square brackets, even if the xml type contains only one attribute or element with the specified name. As a result, the XQuery path must be enclosed in parentheses whenever you use the value() method.

The fourth example shows how to use the modify() method to update the data that's stored in an xml type. To start, this example declares a variable of the xml type and uses a SELECT statement to store an XML document in this variable. Then, it uses a SET statement to replace the value that's stored in the EventType element of the xml variable with a string value of "TEST". To do that, it uses the modify() method with an XML DML "replace value of" statement. In addition, it uses the text() function to retrieve the string value that's stored in the EventType element. Finally, a SELECT statement is used to retrieve the modified value of the xml variable. Although you don't typically use a SELECT statement like this, you shouldn't have any trouble understanding how it works.

In addition to the "replace value of" statement, you can use the XML DML insert and delete statements to insert nodes into and delete nodes from an xml type. To learn more about these statements, you can begin by looking up "modify() Method (xml data type)" in the SQL Server documentation. Then, you can follow the links to learn more about the XML DML statements.

Although you can use the modify() method to modify the data that's stored in an xml data type, you should avoid extensive use of this method whenever possible. Instead, you should consider storing the data in one or more tables. Then, you can use the DML statements that are available from SQL to insert, update, and delete the data in those tables.

A SELECT statement that uses the value() method

```
SELECT
    EventData.value(
        '/EVENT_INSTANCE/EventType')[1],
        'varchar(40)' ) AS EventType,
    EventData.value(
        '/EVENT_INSTANCE/TSQLCommand/SetOptions/@ANSI_NULLS')[1],
        'varchar(40)' ) AS ANSI_NULLS_SETTING
FROM DDLActivityLog
WHERE EventID = 1;
```

The result set

	EventType	ANSI_NULLS_SETTING
1	CREATE_TABLE	ON

A SET statement that uses the modify() method

```
DECLARE @EventData xml;
SELECT @EventData = EventData
FROM DDLActivityLog
WHERE EventID = 1;

SET @EventData.modify
('replace value of (/EVENT_INSTANCE/EventType/text())[1] with "TEST"');

SELECT @EventData AS ModifiedEventData;
```

The result set

	ModifiedEventData
1	<EVENT_INSTANCE><EventType>TEST</EventType><Post...

Description

- *XQuery* is a language that's designed to query an XML document.
- *XML Data Manipulation Language (XML DML)* is a language that's designed to insert, update, or delete nodes from an XML document.

Figure 18-5 How to use the methods of the xml data type (part 2 of 2)

An example that parses the xml data type

If you use an xml type to define a column, you may occasionally need to use the value() method to parse the xml type into a relational result set as shown in figure 18-6. The code in this figure starts by using a SELECT statement to retrieve the EventID column of the DDLActivityLog table. Then, this code uses the value() method to retrieve the values of four elements from the EventData column. Here, the value() method specifies the smalldatetime type for the PostTime element, and it specifies the varchar type for the EventType, LoginName, and ObjectName elements.

In addition, the WHERE clause uses the value() method to specify that the SELECT statement should only return rows where the value stored in the EventType element is equal to “DROP_TABLE”. In this figure, for example, the result set shows three rows that meet that condition. However, the result set that’s displayed by your system may be different, depending on how many tables you have dropped since creating the trigger described in figure 18-3.

An example that parses an xml column into a relational result set

```
SELECT
    EventID,
    EventData.value('(/EVENT_INSTANCE/EventType)[1]', 'varchar(40)')
        AS EventType,
    EventData.value('(/EVENT_INSTANCE/PostTime)[1]', 'smalldatetime')
        AS PostTime,
    EventData.value('(/EVENT_INSTANCE/LoginName)[1]', 'varchar(40)')
        AS LoginName,
    EventData.value('(/EVENT_INSTANCE/ObjectName)[1]', 'varchar(40)')
        AS ObjectName
FROM DDLActivityLog
WHERE
    EventData.value('(/EVENT_INSTANCE/EventType)[1]', 'varchar(40)')
    = 'DROP_TABLE';
```

The result set

	EventID	EventType	PostTime	LoginName	ObjectName
1	2	DROP_TABLE	2020-02-10 15:28:00	murach\anne	VendorsTest
2	4	DROP_TABLE	2020-02-10 15:29:00	murach\anne	VendorsTest

Description

- You can use the value() method to parse an xml data type that has been stored in a database into multiple columns.

Figure 18-6 An example that parses the xml data type

Another example that parses the xml data type

If you find that you often need to write code like the code shown in figure 18-6, you might want to consider parsing the XML data before you store it in the database. Then, you can store the data in the database as relational data, and you can use SQL to query the database as shown in figure 18-7. Before you do that, though, you must determine the structure of the table or tables that will store the XML data, and you must create those tables.

The examples in this figure illustrate how this works. To start, the code in the first example creates a table named DDLActivityLog2 that contains five columns. Of these five columns, the last four get their data from the XML document that's returned by the EVENTDATA function.

Then, the code in the second example creates a trigger named Database_CreateTable_DropTable2. This trigger starts by using the EVENTDATA function to return an XML document that describes the event that caused the trigger to fire. Then, it uses the value() method to insert the values for the EventType, PostTime, LoginName, and ObjectName elements into the DDLActivityLog2 table.

Finally, the last example uses a SELECT statement to return all rows where the EventType column contains a value of "DROP_TABLE." If you compare this SELECT statement with the one in figure 18-6, you'll see that it's shorter and easier to write. If necessary, it would also be efficient and easy to create a join between this table and another table on any of the columns in this table. By contrast, it would require more complex code to join the DDLActivityLog table shown in figure 18-3 to another table, and the join wouldn't run as efficiently. As a result, if you frequently need to query an xml type, and you need to join the xml type with other tables, you should probably parse the XML data before you store it in the database as shown in this figure.

A log table that doesn't use the xml data type

```
CREATE TABLE DDLActivityLog2
(
    EventID int NOT NULL IDENTITY PRIMARY KEY,
    EventType varchar(40) NOT NULL,
    PostTime smalldatetime NOT NULL,
    LoginName varchar(40) NOT NULL,
    ObjectName varchar(40) NOT NULL
);
```

A trigger that parses XML data and inserts it into the table

```
CREATE TRIGGER Database_CreateTable_DropTable2
    ON DATABASE
    AFTER CREATE_TABLE, DROP_TABLE
AS
    DECLARE @EventData xml;
    SELECT @EventData = EVENTDATA();
    INSERT INTO DDLActivityLog2 VALUES
    (
        @EventData.value('/EVENT_INSTANCE/EventType')[1], 'varchar(40)'),
        @EventData.value('/EVENT_INSTANCE/PostTime')[1], 'varchar(40)'),
        @EventData.value('/EVENT_INSTANCE/LoginName')[1], 'varchar(40)'),
        @EventData.value('/EVENT_INSTANCE/ObjectName')[1], 'varchar(40)')
    );
```

A SELECT statement that retrieves data from the table

```
SELECT * FROM DDLActivityLog2
WHERE EventType = 'DROP_TABLE';
```

The result set

	EventID	EventType	Post Time	LoginName	ObjectName
1	2	DROP_TABLE	2020-02-10 15:28:00	murach\anne	VendorsTest
2	4	DROP_TABLE	2020-02-10 15:29:00	murach\anne	VendorsTest
3	6	DROP_TABLE	2020-02-10 15:31:00	murach\anne	VendorsTest

Description

- You can use the value() method to parse an xml data type into multiple columns before you store it in the database.

Figure 18-7 Another example that parses the xml data type

How to work with XML schemas

Now that you understand the basics for working with the xml data type, you're ready to learn how to use an XML schema with this data type. Since an XML schema validates the data that's stored in an xml type, this data is known as *typed XML*.

How to add an XML schema to a database

Before you can use an XML schema with the xml data type, you must add the XML Schema Definition to the database. To do that, you can use the CREATE XML SCHEMA COLLECTION statement. In figure 18-8, for example, this statement is used to add the XML schema presented in figure 18-2 to the database. Although this looks like a lot of code at first glance, most of it is the string that defines the XML schema. As I've already mentioned, you may be able to get this information from a colleague or from the Internet. Or, if you have a document that the schema will be based on, you can use the XML Editor to generate the XML schema.

The first line of code contains the CREATE XML SCHEMA COLLECTION clause. This clause specifies EventDataSchema as the name of the schema. Since the optional database schema name isn't specified, the XML schema will be stored in the default database schema (dbo). The second line of code contains the AS clause that indicates that the expression that follows will specify the XML schema. This expression can be coded as a string literal as shown in this figure or as a variable of the varchar, nvarchar, or xml types.

When you use the CREATE XML SCHEMA COLLECTION statement, you're actually adding the XML schema to a collection of XML schemas. In other words, you can add multiple XML schemas to the same collection. However, for this to work correctly, you must add a targetNamespace attribute to the XML schema that uniquely identifies each schema. Since this is more complicated than using one XML schema per collection, it's a common practice to only store one XML schema per collection as shown in this figure. In that case, you can think of the collection and the XML schema as being the same database object.

After you add a schema to a database, you can see it in the Object Explorer of the Management Studio. To do that, just expand the Programmability, Types, and XML Schema Collections folders. Then, you can use the menu that's displayed when you right-click on a schema to create scripts for creating and dropping the schema, to display the dependencies for the schema, and to delete the schema. This works just like it does for other database objects.

The syntax for the CREATE XML SCHEMA COLLECTION statement

```
CREATE XML SCHEMA COLLECTION [database_schema_name.]xml_schema_name  
AS xml_schema_expression
```

An example that creates a schema

```
CREATE XML SCHEMA COLLECTION EventDataSchema  
AS  
'  
  
<xs:schema attributeFormDefault="unqualified"  
elementFormDefault="qualified"  
xmlns:xs="http://www.w3.org/2001/XMLSchema">  
    <xs:element name="EVENT_INSTANCE">  
        <xs:complexType>  
            <xs:sequence>  
                <xs:element name="EventType" type="xs:string" />  
                <xs:element name="PostTime" type="xs:dateTime" />  
                <xs:element name="SPID" type="xs:unsignedByte" />  
                <xs:element name="ServerName" type="xs:string" />  
                <xs:element name="LoginName" type="xs:string" />  
                <xs:element name="UserName" type="xs:string" />  
                <xs:element name="DatabaseName" type="xs:string" />  
                <xs:element name="SchemaName" type="xs:string" />  
                <xs:element name="ObjectName" type="xs:string" />  
                <xs:element name="ObjectType" type="xs:string" />  
                <xs:element name="TSQLCommand">  
                    <xs:complexType>  
                        <xs:sequence>  
                            <xs:element name="SetOptions">  
                                <xs:complexType>  
                                    <xs:attribute name="ANSI_NULLS"  
                                        type="xs:string" use="required" />  
                                    <xs:attribute name="ANSI_NULL_DEFAULT"  
                                        type="xs:string" use="required" />  
                                    <xs:attribute name="ANSI_PADDING"  
                                        type="xs:string" use="required" />  
                                    <xs:attribute name="QUOTED_IDENTIFIER"  
                                        type="xs:string" use="required" />  
                                    <xs:attribute name="ENCRYPTED"  
                                        type="xs:string" use="required" />  
                                </xs:complexType>  
                            </xs:element>  
                            <xs:element name="CommandText" type="xs:string" />  
                        </xs:sequence>  
                    </xs:complexType>  
                </xs:element>  
            </xs:sequence>  
        </xs:complexType>  
    </xs:element>  
</xs:schema>  
';
```

Description

- Before you can use an XML Schema Definition with the `xml` data type, you must add the XSD to the database. To do that, you can use the `CREATE XML SCHEMA COLLECTION` statement.

Figure 18-8 How to add an XML schema to a database

How to use an XML schema to validate the xml data type

Figure 18-9 shows how to use an XML schema to validate the `xml` type. To do that, you code the name of the XML schema in parentheses immediately after you specify the `xml` type for a column or a variable. In this figure, for example, the `CREATE TABLE` statement specifies the `EventDataSchema` XML schema that was added to the database in figure 18-8 as the schema for the `EventData` column. As a result, any time an `INSERT` statement attempts to insert data into this column, SQL Server will use this XML schema to validate the data. Since this provides a standard way to validate XML data, it is known as *XML validation*.

If the data doesn't conform to the XML schema, SQL Server will raise an appropriate error. This is illustrated by the three `INSERT` statements in this figure. For example, the first `INSERT` statement contains a string that doesn't use XML tags. As a result, SQL Server raises an error that indicates that this isn't allowed by the XML schema.

Although the second `INSERT` statement specifies a well-formed XML document, the tags for this document don't match the tags specified by the XML schema. As a result, SQL Server raises an error that indicates that no declaration was found for the `MyRoot` element, which is the first element of the XML document that raised an error.

The third `INSERT` statement also specifies a well-formed XML document. However, this document only contains the first two elements specified by the XML schema. As a result, SQL Server raises an error that indicates that it expected the third element in the XML document, the `PostTime` element.

Although all of the examples in this figure work with a column in a table, you can also use an XML schema to validate a variable. In the last example in this figure, for instance, you could declare the `@CreateTableEvent` variable like this:

```
DECLARE @CreateTableEvent xml (EventDataSchema)
```

Then, SQL Server would raise an error like the one shown here when the `SET` statement in this figure attempted to store the invalid XML data in the variable. In other words, the XML data would be validated before you even attempted to store it in the table.

The syntax for declaring XML schema validation

```
column_or_variable_name XML ([database_schema_name.]xml_schema_name)
```

A log table with a column that specifies an XML schema

```
CREATE TABLE DDLActivityLog3
(EventID int NOT NULL IDENTITY PRIMARY KEY,
EventData xml (EventDataSchema) NOT NULL);
```

An INSERT statement that attempts to insert non-XML data

```
INSERT INTO DDLActivityLog3 VALUES ('this is not xml');
```

The response from the system

```
Msg 6909, Level 16, State 1, Line 1
XML Validation: Text node is not allowed at this location, the type was defined
with element only content or with simple content.
Location: /
```

An INSERT statement that attempts to insert XML data whose tags don't match the tags in the schema

```
INSERT INTO DDLActivityLog3
VALUES ('<MyRoot><MyElement>test</MyElement></MyRoot>');
```

The response from the system

```
Msg 6913, Level 16, State 1, Line 1
XML Validation: Declaration not found for element 'MyRoot'.
Location: /*:MyRoot[1]
```

An INSERT statement that attempts to insert XML data that doesn't contain all the elements specified by the schema

```
DECLARE @CreateTableEvent xml;
SET @CreateTableEvent = '
<EVENT_INSTANCE>
    <EventType>CREATE_TABLE</EventType>
</EVENT_INSTANCE>
';
INSERT INTO DDLActivityLog3
VALUES (@CreateTableEvent);
```

The response from the system

```
Msg 6908, Level 16, State 1, Line 2
XML Validation: Invalid content. Expected element(s): PostTime.
Location: /*:EVENT_INSTANCE[1]
```

Description

- To provide *XML validation*, you can specify an XSD for the xml data type when you use it to define a column or a variable. Then, when you attempt to store data in the xml data type, SQL Server will use the XSD to make sure the data is valid. If the data isn't valid, SQL Server will display an appropriate error message.

How to view an XML schema

Figure 18-10 shows how to view the XML schemas that have been added to the database. If you know the name of the XML schema, you can use the XML_SCHEMA_NAMESPACE function to return an xml type that contains the XML schema. In this figure, for instance, the first example uses a SELECT statement to return an xml type that contains the EventDataSchema that was added to the database in figure 18-8. Then, you can view this schema in the XML Editor by clicking on the link for the XML schema.

You can also use the sys.xml_schema_collections catalog view to get information about an XML schema. If you don't know the name of an XML schema, for example you can code a query like this:

```
SELECT name FROM sys.xml_schema_collections;
```

Then, you can find the name of the XML schema you want to view and use the XML_SCHEMA_NAMESPACE function to view it.

How to drop an XML schema

Figure 18-10 also shows how to drop an XML schema from the database. To do that, you can use the DROP XML SCHEMA COLLECTION statement. In this figure, for instance, the last example drops the EventDataSchema collection. But first, it queries the sys.xml_schema_collections catalog view to make sure that the EventDataSchema collection exists.

Note that you can't drop an XML schema if it's being used by another object. For example, you can't drop a schema if it's being used to validate a column in a table. To do that, you must first alter the table so the column doesn't name the schema.

The syntax for the XML_SCHEMA_NAMESPACE function

```
XML_SCHEMA_NAMESPACE('database_schema_name', 'xml_schema_name')
```

A statement that returns the XML schema

```
SELECT XML_SCHEMA_NAMESPACE('dbo', 'EventDataSchema');
```

The result set

1	EventDataSchema
	xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'

The syntax for the DROP XML SCHEMA COLLECTION statement

```
DROP XML SCHEMA COLLECTION [database_schema_name.]xml_schema_name
```

An example that drops a schema

```
IF EXISTS
    (SELECT * FROM sys.xml_schema_collections
     WHERE name = 'EventDataSchema')
BEGIN
    DROP XML SCHEMA COLLECTION EventDataSchema;
END;
```

Description

- To view an XML schema that has been added to the database, you can use the XML_SCHEMA_NAMESPACE function.
- To get information about XML schemas that have been added to the database, you can query the sys.xml_schema_collections catalog view.
- To drop an XML schema from the database, you can use the DROP XML SCHEMA COLLECTION statement. You can't drop an XML schema if it's being used by another object.

Figure 18-10 How to view or drop an XML schema

Two more skills for working with XML

When you work with XML, you may occasionally need to convert it to relational data. Conversely, you may occasionally need to convert relational data to XML. To do that, you use the skills that follow.

How to use the FOR XML clause of the SELECT statement

If you need to convert relational data that's stored in your database to XML, you can usually accomplish this by using the FOR XML clause of the SELECT statement as shown in figure 18-11. To use this clause, you use the SELECT statement to retrieve the data that you want to convert to XML. Then, you add a FOR XML clause after the SELECT statement to specify how the relational data should be converted to XML.

The first example shows how to use the FOR XML clause with the RAW and ROOT keywords. To start, the SELECT statement uses a join to select the VendorName column from the Vendors table and the InvoiceNumber and InvoiceTotal columns from the Invoices table. In addition, this SELECT statement uses the TOP 5 clause so it only returns 5 rows. Then, the FOR XML clause uses the RAW keyword, which returns one element named row for each row in the result set. In addition, the ROOT keyword is used to specify the name of the root element for the XML document. In this example, the root element is named VendorInvoices. Although the ROOT keyword isn't required, all valid XML documents must have a root element. As a result, if you want to return a complete XML document, you'll need to specify a root element.

The second example is similar, but it uses the AUTO keyword instead of the RAW keyword. The AUTO keyword causes the relationships between the Vendors and Invoices tables to be maintained in the XML document. As a result, one Vendor element is returned for each vendor, one Invoice element is returned for each invoice, and each Invoice element is stored within its related Vendor element. That means that the vendor name isn't repeated for each invoice as in the first example. Notice that to make this work more elegantly, the SELECT statement in this example uses correlation names for the Vendors and Invoices tables to make them singular instead of plural. That way, the data for a vendor is stored in an element named Vendor instead of an element named Vendors, and the data for an invoice is stored in an element named Invoice instead of an element named Invoices.

The simplified syntax of the FOR XML clause

```
select_statement
FOR XML {RAW|AUTO} [, ROOT ('RootName')] [, ELEMENTS ]
```

A SELECT statement that uses the RAW keyword

```
SELECT TOP 5 VendorName, InvoiceNumber, InvoiceTotal
FROM Vendors JOIN Invoices
    ON Vendors.VendorID = Invoices.VendorID
ORDER BY VendorName
FOR XML RAW, ROOT ('VendorInvoices');
```

The XML document that's returned

```
<VendorInvoices>
<row VendorName="Abbey Office Furnishings"
    InvoiceNumber="203339-13" InvoiceTotal="17.5000" />
<row VendorName="Bertelsmann Industry Svcs. Inc"
    InvoiceNumber="509786" InvoiceTotal="6940.2500" />
<row VendorName="Blue Cross"
    InvoiceNumber="547481328" InvoiceTotal="224.0000" />
<row VendorName="Blue Cross"
    InvoiceNumber="547479217" InvoiceTotal="116.0000" />
<row VendorName="Blue Cross"
    InvoiceNumber="547480102" InvoiceTotal="224.0000" />
</VendorInvoices>
```

A SELECT statement that uses the AUTO keyword

```
SELECT TOP 5 VendorName, InvoiceNumber, InvoiceTotal
FROM Vendors AS Vendor JOIN Invoices AS Invoice
    ON Vendor.VendorID = Invoice.VendorID
ORDER BY VendorName
FOR XML AUTO, ROOT ('VendorInvoices');
```

The XML document that's returned

```
<VendorInvoices>
<Vendor VendorName="Abbey Office Furnishings">
    <Invoice InvoiceNumber="203339-13" InvoiceTotal="17.5000" />
</Vendor>
<Vendor VendorName="Bertelsmann Industry Svcs. Inc">
    <Invoice InvoiceNumber="509786" InvoiceTotal="6940.2500" />
</Vendor>
<Vendor VendorName="Blue Cross">
    <Invoice InvoiceNumber="547481328" InvoiceTotal="224.0000" />
    <Invoice InvoiceNumber="547479217" InvoiceTotal="116.0000" />
    <Invoice InvoiceNumber="547480102" InvoiceTotal="224.0000" />
</Vendor>
</VendorInvoices>
```

Description

- To specify the name of the root element for the XML document, use the ROOT keyword.
- To return one XML element for each row, use the RAW keyword.
- To automatically parse the rows of the result set so the elements reflect the structure of the tables in the database, use the AUTO keyword.

Figure 18-11 How to use the FOR XML clause of the SELECT statement (part 1 of 2)

As you saw in the first two examples in figure 18-11, the data for each column that's returned by a SELECT statement that includes the FOR XML clause is stored in attributes. However, if you want to store the data for each column in XML elements instead, you can add the ELEMENTS keyword to the FOR XML clause. The third example in this figure, for instance, is identical to the second example except that it includes the ELEMENTS keyword. Although the XML document that's returned by this example is longer than in the previous example, it's also easier to read because it uses elements instead of attributes. In some cases, you may prefer to use this type of XML document.

Most of the time, the skills presented in this figure are all you need to use the FOR XML clause to create XML documents from the relational data in a database. However, several other options are available from this clause that give you even more control over the format of the XML document that's returned by the SELECT statement. For more information, look up "FOR XML (SQL Server)" in the SQL Server documentation.

A SELECT statement that uses the AUTO and ELEMENTS keywords

```
SELECT TOP 5 VendorName, InvoiceNumber, InvoiceTotal
FROM Vendors AS Vendor JOIN Invoices AS Invoice
    ON Vendor.VendorID = Invoice.VendorID
ORDER BY VendorName
FOR XML AUTO, ROOT ('VendorInvoices'), ELEMENTS;
```

The XML document that's returned

```
<VendorInvoices>
    <Vendor>
        <VendorName>Abbey Office Furnishings</VendorName>
        <Invoice>
            <InvoiceNumber>203339-13</InvoiceNumber>
            <InvoiceTotal>17.5000</InvoiceTotal>
        </Invoice>
    </Vendor>
    <Vendor>
        <VendorName>Bertelsmann Industry Svcs. Inc</VendorName>
        <Invoice>
            <InvoiceNumber>509786</InvoiceNumber>
            <InvoiceTotal>6940.2500</InvoiceTotal>
        </Invoice>
    </Vendor>
    <Vendor>
        <VendorName>Blue Cross</VendorName>
        <Invoice>
            <InvoiceNumber>547481328</InvoiceNumber>
            <InvoiceTotal>224.0000</InvoiceTotal>
        </Invoice>
        <Invoice>
            <InvoiceNumber>547479217</InvoiceNumber>
            <InvoiceTotal>116.0000</InvoiceTotal>
        </Invoice>
        <Invoice>
            <InvoiceNumber>547480102</InvoiceNumber>
            <InvoiceTotal>224.0000</InvoiceTotal>
        </Invoice>
    </Vendor>
</VendorInvoices>
```

Description

- By default, the data that's stored in a database column is stored in an XML attribute. To store data in XML elements instead, use the ELEMENTS keyword.
- The names that are used for elements and attributes correspond to the names of the tables and columns that are used in the query. However, you can specify correlation names for the tables and columns in your query to modify these names whenever that's necessary.

Figure 18-11 How to use the FOR XML clause of the SELECT statement (part 2 of 2)

How to use the OPENXML statement

If you need to convert XML data into relational data, you can usually accomplish this by using the OPENXML statement to open the XML data as a result set. The example in figure 18-12 shows how this works.

Before you can use the OPENXML statement, you must read the XML document into memory and get a handle that points to that document. This handle is a unique integer value that SQL Server can use to refer to the XML document. In the example in this figure, the code starts by declaring a variable to store this handle. Then, it declares a variable to store the XML, and it sets this variable to the XML shown in part 2 of figure 18-11. Next, it calls the `sp_Xml_Prepardocument` procedure to read the XML document into memory and return the handle to this document.

Once you execute the `sp_Xml_Prepardocument` procedure, you can use the OPENXML statement to open an `xml` data type as a result set. Since this statement returns a result set, it's coded in the `FROM` clause of a `SELECT` statement as shown in this example.

The OPENXML statement begins by accepting an integer argument that specifies the handle to the XML document that was prepared by the `sp_Xml_Prepardocument` procedure. Then, the second argument uses an *XPath* string to identify the XML elements to be processed. *XPath* is an XML query language similar to XQuery. As a result, you should be able to understand how it works. In this figure, for example, the *XPath* argument specifies that the table should have one row for each `Invoice` element in the XML document.

The `WITH` clause of the OPENXML statement allows you to specify a definition for the table. To start, you specify the name and data type for each column in the table just as you would with a `CREATE TABLE` statement. Then, you specify an *XPath* string that identifies the element or attribute in the XML document that contains the data for the column. If necessary, this *XPath* string can use two dots (..) to navigate back one level in the XML hierarchy. In this figure, for example, the *XPath* string for the `VendorName` column uses two dots to navigate from the `Invoice` element to the `Vendor` element.

After you use the OPENXML statement, you should use the `sp_Xml_RemoveDocument` procedure to remove the XML document from memory. This frees system resources and allows SQL Server to run more efficiently.

Although this figure should get you started quickly with the OPENXML statement, it's only intended to be an introduction to this feature. For more information, look up "OPENXML (Transact-SQL)" in the SQL Server documentation.

The simplified syntax for the OPENXML statement

```
OPENXML (xml_document_handle_int, x_path)
WITH ( table_definition )
```

The simplified syntax for the sp_Xml_Prepardocument procedure

```
EXEC sp_Xml_Prepardocument xml_document_handle_int OUTPUT, xml_document
```

The simplified syntax for the sp_Xml_RemoveDocument procedure

```
EXEC sp_Xml_RemoveDocument xml_document_handle_int
```

Code that uses the OPENXML statement to parse XML

```
-- Declare an int variable that's a handle for the internal XML document
DECLARE @VendorInvoicesHandle int;

-- Create an xml variable that stores the XML document
DECLARE @VendorInvoices xml;
SET @VendorInvoices = ' xml from part 2 of 18-11 goes here ';

-- Prepare the internal XML document
EXEC sp_Xml_Prepardocument @VendorInvoicesHandle OUTPUT, @VendorInvoices;

-- SELECT the data from the table returned by the OPENXML statement
SELECT *
FROM OPENXML (@VendorInvoicesHandle, '/VendorInvoices/Vendor/Invoice')
WITH
(
    VendorName      varchar(50)  '../VendorName',
    InvoiceNumber   varchar(50)  'InvoiceNumber',
    InvoiceTotal    money        'InvoiceTotal'
);

-- Remove the internal XML document
EXEC sp_Xml_RemoveDocument @VendorInvoicesHandle;
```

The result set

	VendorName	InvoiceNumber	InvoiceTotal
1	Abbey Office Furnishings	203339-13	17.50
2	Bertelsmann Industry Svcs. Inc	509786	6940.25
3	Blue Cross	547479217	116.00
4	Blue Cross	547480102	224.00
5	Blue Cross	547481328	224.00

Description

- Before you can use the OPENXML statement, you must use the sp_Xml_Prepardocument procedure to read the XML document into memory and return a handle to this document.
- You can use the OPENXML statement to open an xml data type as a result set. In the WITH clause, you can use two dots (..) to navigate back one level in the XML hierarchy.
- After you use the OPENXML statement, you should use the sp_Xml_RemoveDocument procedure to remove the XML document from memory.

Figure 18-12 How to use the OPENXML statement

Perspective

XML is a useful technology for storing and transferring data, and the features that SQL Server provides make it easy to work with XML. Of course, SQL Server is a relational database that is primarily designed to work with relational data. As a result, you should think twice before using the xml data type to store XML data in a table.

Instead, whenever possible, you should parse the XML data and store it in one or more tables. That way, you can still use SQL to easily retrieve and update the data. However, if the data is structured in a way that makes it difficult to store in tables, or if you won't need to retrieve or update the data very often, you can use the skills presented in this chapter to store and work with that data.

Terms

Extensible Markup Language (XML)	XML Schema Definition (XSD)
XML document	XML schema
tag	untyped XML
element	XML Editor
start tag	method
end tag	XQuery
attribute	XML Data Manipulation Language (XML DML)
content	typed XML
child element	XML validation
parent element	XPath
root element	

Exercises

1. Write a SELECT statement that returns an XML document that contains all of the invoices in the Invoices table that have more than one line item. This document should include one element for each of these columns: InvoiceNumber, InvoiceDate, InvoiceTotal, InvoiceLineItemDescription, and InvoiceLineItemAmount. Then, save the XML document that's returned in a file named MultipleLineItems.xml. Finally, generate an XML schema for the file and save it in a file named MultipleLineItems.xsd.
2. Write a script that uses the XML document shown below to update the contact information in the Vendors table.

```
<ContactUpdates>
    <Contact VendorID="4">
        <LastName>McCrystle</LastName>
        <FirstName>Timothy</FirstName>
    </Contact>
    <Contact VendorID="10">
        <LastName>Flynn</LastName>
        <FirstName>Erin</FirstName>
    </Contact>
</ContactUpdates>
```

To accomplish this, begin by storing this XML document in a variable of the XML type. Then, you can use two UPDATE statements to update the Vendors table.

3. Write a script that returns a result set that contains all of the data stored in the XML document in exercise 2.
4. Write a script that (1) creates a table named Instructions, (2) inserts the XML document shown below into the table, and (3) selects all rows from this table. The Instructions table should have two columns. The first column should be an identity column named InstructionsID, and the second column should be an xml column named Instructions.

```
<Instructions>
    <Step>
        <Description>This is the first step.</Description>
        <SubStep>This is the first substep.<SubStep>
        <SubStep>This is the second substep.<SubStep>
    </Step>
    <Step>
        <Description>This is the second step.</Description>
    </Step>
    <Step>
        <Description>This is the third step.</Description>
    </Step>
</Instructions>
```


How to work with BLOBs

In chapter 8, you were introduced to the data types for working with large values, including the varbinary(max) data type that's used to work with large binary values such as images, sound, and video. In this chapter, you'll learn how to use SQL and a .NET application to work with large binary values, which are often referred to as binary large objects (BLOBs). Then, you'll learn how to use a feature known as FILESTREAM storage. This feature provides some enhancements that you can use to efficiently work with BLOBs that are larger than 1 megabyte.

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
Perspective	646

An introduction to BLOBs

Figure 19-1 describes the pros and cons of three approaches that you can use for working with large binary values, which are often referred to as *binary large objects (BLOBs)*.

Pros and cons of storing BLOBs in files

The first approach is the oldest approach and was commonly used prior to SQL Server 2005. This approach stores a string value in a database column that points to a binary file that's stored on the file system. For an image, for example, you can include a column with a string value such as "8601_cover.jpg" that points to a JPG image file. If necessary, you can also store a relative path or absolute path to the file within the database.

There are two advantages to this approach. First, there is no limit on the size of the BLOB unless the file system begins to run out of disk space. Second, the file system provides fast access to the BLOB.

There are also a couple of disadvantages to this approach. First, the BLOB is not backed up as part of the database backup. Second, access to the BLOB is controlled by network permissions, not by database permissions. This can create additional work for the network administrator, and it can lead to data consistency and security issues.

Pros and cons of storing BLOBs in a column

The second approach was introduced with SQL Server 2005, and it solves the problems of storing a pointer to a file by storing the binary data in a varbinary(max) column within the database. As a result, the binary data is backed up when the database is backed up, and access to this data is controlled by database permissions.

However, this approach has two limitations. First, the binary data must be less than 2 gigabytes (GB). Second, database access is not as fast as file system access. As a result, you can't use this approach if you need to store a BLOB that's larger than 2GB, and you won't want to use it if fast access to your BLOBs is critical to your application.

When to use FILESTREAM storage for BLOBs

The third approach, which is known as *FILESTREAM storage*, was introduced with SQL Server 2008. This approach overcomes all of the limitations of the first two approaches. However, it also requires more work to set up and to use. As a result, you'll only want to use this approach when most of the BLOBs that you need to store are larger than 1 megabyte (MB) and when fast read access is critical. Otherwise, the second approach usually provides adequate performance without requiring any additional work.

Three approaches to storing binary data

- Use a varchar column in the database to store a string that points to a file in the file system that contains the binary data.
- Use a varbinary(max) column to store the binary data in the database.
- Use a varbinary(max) column with the FILESTREAM attribute to store the binary data.

Pros and cons of using a pointer to a binary file

Pros

- There is no limit on the size of the BLOB.
- The file system provides fast access to the BLOB.

Cons

- The BLOB is not backed up with the database.
- Access to the BLOB is controlled by network security, not database security.

Pros and cons of using the varbinary(max) data type

Pros

- The BLOB is backed up with the database.
- Database security can be used to control access to the BLOB.

Cons

- The BLOB must be smaller than 2GB.
- Database access is not as fast as file system access.

Pros and cons of using FILESTREAM storage

Pros

- The BLOB can be larger than 2GB.
- The BLOB access is as fast as file system access.
- The BLOB is backed up with the database.
- Database security can be used to control access to the BLOB.

When to use FILESTREAM storage

- When most of the BLOBS in the column are larger than 1MB.
- When fast read access is critical to the application.

Description

- With SQL Server 2008 and later, you can use a feature known as *FILESTREAM storage* to overcome several limitations for working with *binary large objects (BLOBS)* that existed in previous versions of SQL Server.
- To be able to use FILESTREAM storage, the drive that stores the files must be in NTFS format.

How to use SQL to work with a varbinary(max) column

Since using a varbinary(max) column is adequate for storing binary data in many situations, this topic shows how to use this approach. To start, figure 19-2 shows how to use SQL to work with a varbinary(max) column of a table.

How to create a table with a varbinary(max) column

To create a table that has a varbinary(max) column, you can specify varbinary(max) as the data type for a column within a CREATE TABLE statement as shown in this figure. Here, the first column is the primary key for the table. This column stores an ID value for an image, and this value is automatically generated by the database. Then, the second column stores an ID value for a product. This value can be used as a foreign key to relate an image to a product. Finally, the third column uses the varbinary(max) data type to store the binary data for the image.

To keep this example simple, I only included three columns. However, a table like this might include additional columns such as a name for the image.

How to insert, update, and delete binary data

To work with binary data, you can use INSERT, UPDATE, and DELETE statements just as you would for other types of data. In this figure, for example, the three INSERT statements insert three rows into the ProductImages table. Here, the first statement uses the NULL keyword to insert a NULL value into the ProductImage column. Then, the second statement inserts an integer value of zero. Finally, the third statement uses the CAST function to convert a hexadecimal string to a varbinary(max) value.

Although these statements show how the INSERT statement works, the data that's stored in the ProductImage column is not valid data for an image. To insert valid data, you can use a .NET application like the one shown in the next figure to upload data from the file system to the database.

How to retrieve binary data

To retrieve binary data, you can use a SELECT statement just as you would for other types of data. In this figure, for example, the SELECT statement selects all columns and rows from the ProductImages table. This shows the data that was inserted by the three INSERT statements. Here, the first row in the ProductImage column stores a NULL value and the next two rows store binary data.

How to create a table with a varbinary(max) column

```
CREATE TABLE ProductImages
(
    ImageID int PRIMARY KEY IDENTITY,
    ProductID int NOT NULL,
    ProductImage varbinary(max)
);
```

Three INSERT statements that insert rows into the table

```
INSERT INTO ProductImages VALUES (1, NULL);

INSERT INTO ProductImages VALUES (2, 0);

INSERT INTO ProductImages
VALUES (3, CAST('0123456789ABCDEF' AS varbinary(max)));
```

A statement that displays the values in the table

```
SELECT * FROM ProductImages;
```

The result set

	ImageID	ProductID	ProductImage
1	1	1	NULL
2	2	2	0x00000000
3	3	3	0x30313233343536373839414243444546

Description

- You can use the varbinary(max) data type for a column that stores binary data.

Figure 19-2 How to create a table and insert binary data

A .NET application that uses a varbinary(max) column

Once you understand how to use SQL to work with binary data that's stored in a varbinary(max) column, you can use a .NET application to write binary data that's in a file to a column in a database. Then, you can use a .NET application to present binary data in a way that's meaningful to the user. In this chapter, for example, you'll learn how to read binary data for an image from a column in the database and display it in a picture box control on a Windows form.

This topic presents a simple application that uses C# to work with BLOBS. All of the principles in this application apply to Visual Basic and the other .NET languages. The differences mainly have to do with the syntax of each language. If you prefer to use Visual Basic, you can download the Visual Basic code for this application from our website (see appendix A).

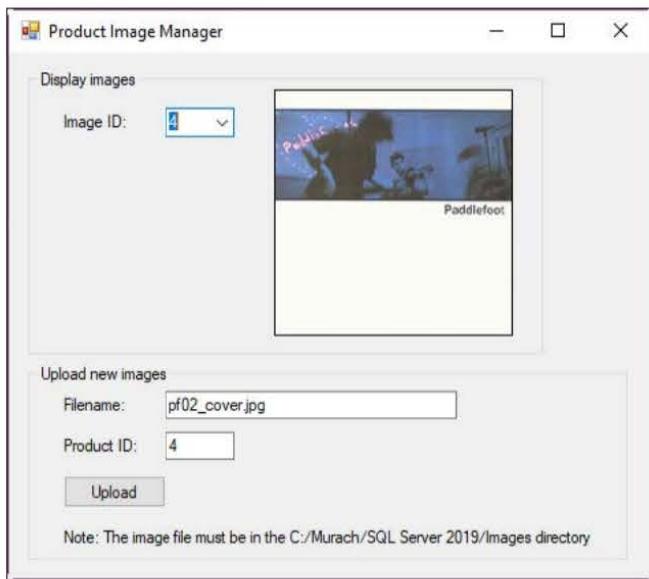
If you have some C# programming experience, you shouldn't have much trouble understanding this code. If you don't have C# experience, that's fine too. In that case, you can focus on how this code uses the .NET Framework to execute SQL statements against a database. Then, if you want to learn more about writing C# applications, we recommend our current book on C#. For more information, please see www.murach.com.

The user interface for the application

Figure 19-3 shows the user interface for the Product Image Manager application that's presented in this topic. You can use this application to view the images that have been stored in the database. To do that, you select the ID for the image from the combo box that's displayed at the top of the form. Then, the application displays the image in a picture box control to the right of the combo box.

You can also use this application to upload images from the file system to the database. To do that, you enter a filename for the image, and an ID for the product. In this figure, for example, an image file named "pf02_cover.jpg" with a product ID of 4 has just been added to the database. Note that this only works if the image is stored in the directory that's listed at the bottom of the form.

The user interface



Description

- To view an image, select the ID for the image from the combo box.
- To upload an image, enter a filename and a product ID for the image, and click on the Upload button.

Figure 19-3 The user interface for the Product Image Manager application

The event handlers for the form

Figure 19-4 presents the event handlers for the form. To start, the event handler for the Load event of the form calls the `LoadImageIDComboBox()` method. This method reads all image IDs from the database and adds them to the Image ID combo box. To accomplish this task, this method calls the `GetImageIDList()` method of the `ProductDB` class that's presented in the next figure. After loading the combo box, the event handler for the Load event calls the event handler for the `SelectedIndexChanged` event of the combo box. This causes the first image in the database to be displayed on the form when the form is loaded.

The event handler for the `SelectedIndexChanged` event of the combo box begins by getting the image ID that's selected in the combo box and converting this ID from a string type to an int type. Then, this event handler uses the `ReadImage()` method of the `ProductDB` class to get an array of bytes for the image. Next, it converts the array of bytes to a `MemoryStream` object. Finally, it sets the `Image` property of the picture box on the form to the image that's returned by the static `FromStream()` method of the `System.Drawing.Image` class. If this event handler encounters an error, it uses a dialog box to display an error message.

The event handler for the Click event of the Upload button begins by getting the product ID that's entered into the Product ID text box on the form and converting this ID from a string type to an int type. Then, it gets the filename from the Filename text box on the form. After that, it uses the `WriteImage()` method of the `ProductDB` class to write the image from the specified file to the database, and it displays a dialog box to confirm that the image has been successfully uploaded. Finally, this event handler clears all image ID values from the Image ID combo box and calls the private `LoadImageIDComboBox()` method to load this combo box with fresh values that include the ID for the new image that was uploaded. Like the previous event handler, this event handler displays a dialog box that displays an error message if it encounters an error.

Although you can't see it here, you should know that the `program.cs` file for this application contains code that writes three rows to the `ProductImages` table if the table doesn't contain any rows. This code is executed before the Load event for the form. That way, the combo box on the form will contain at least three rows when it's first displayed.

The event handlers for the form

```
private void ImageManagerForm_Load(object sender, EventArgs e)
{
    this.LoadImageIDComboBox();
    imageIDComboBox_SelectedIndexChanged(sender, e);
}

private void LoadImageIDComboBox()
{
    // load the combo box
    List<int> imageIDList = ProductDB.GetImageIDList();
    foreach (int i in imageIDList)
        imageIDComboBox.Items.Add(i);
}

private void imageIDComboBox_SelectedIndexChanged(
    object sender, EventArgs e)
{
    try
    {
        int imageID = Convert.ToInt32(imageIDComboBox.Text);

        // read image bytes from the database and display in picture box
        Byte[] imageByteArray = ProductDB.ReadImage(imageID);
        MemoryStream ms = new MemoryStream(imageByteArray);
        imagePictureBox.Image = System.Drawing.Image.FromStream(ms);
        ms.Close();
    }
    catch (Exception ex)
    {
        MessageBox.Show(this, ex.Message, "Error");
    }
}

private void uploadButton_Click(object sender, EventArgs e)
{
    try
    {
        int productID = Convert.ToInt32(productIDTextBox.Text);
        string filename = filenameTextBox.Text;
        ProductDB.WriteImage(productID, filename);
        MessageBox.Show(this, "Image upload was successful!",
            "Upload Confirmation");

        // refresh combo box
        imageIDComboBox.Items.Clear();
        this.LoadImageIDComboBox();

    }
    catch (Exception ex)
    {
        MessageBox.Show(this, ex.Message, "Error");
    }
}
```

Figure 19-4 The event handlers for the Product Image Manager form

A data access class that reads and writes binary data

Figure 19-5 shows the ProductDB class that's used to read and write binary data from the ProductImage column of the ProductImages table. To start, this class defines a string that points to the directory for the image files that are going to be uploaded into the database. As a result, this class only allows you to upload image files that are stored in this directory. Of course, this class could be enhanced to allow the user to specify the directory for the file that he or she wants to upload.

The static WriteImage() method writes the specified image file to the database. To start, this method accepts two parameters. The first parameter is an int value for the product ID that's stored in the same row as the image. The second parameter is a string value that specifies the name of the file.

The body of the WriteImage() method begins by reading the specified file into an array of bytes. To start, this code creates a variable named filePath that contains an absolute path that points to the file for the image. Then, the static Exists() method of the File class is used to check whether the specified file exists. If it doesn't exist, this code throws an exception and skips directly to the catch block. If the file does exist, this code continues by creating a FileStream object named sourceStream that's used to read the file from the file system into an array of bytes. To do that, this code uses the Read() method of the sourceStream object to read an array of bytes from the specified file into a variable named productImage.

After reading the image from the file into an array of bytes, the WriteImage() method continues by writing the product ID and image to the database. To do that, this method calls the GetConnection() method to get a connection to the database. You'll see the code for this method in just a minute. Next, this method creates a SqlCommand object that contains an INSERT statement like the one shown in figure 19-2. However, instead of hard-coding values, this INSERT statement accepts two parameters: @ProductID and @ProductImage. These parameters are then added to the Parameters collection of the SqlCommand object. The @ProductID parameter is given the value of the productID parameter that was passed to the method, and the @ProductImage parameter is given the value of the image that was retrieved from the file. Finally, this method calls the Open() method to open the connection, and it calls the ExecuteNonQuery() method to execute the INSERT statement that's stored in the SqlCommand object.

If the method causes an exception to be thrown, the catch block throws the exception again. In most cases, this causes the exception to be caught by one of the catch blocks in the event handlers for the form.

Whether or not this method executes cleanly or throws an exception, the finally block attempts to close the Connection object that was opened. To do that, this code first checks to make sure that the connection object is not null. If it isn't, it closes the open connection.

The ProductDB class**Part 1**

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.IO;

namespace MusicStoreImageManager
{
    class ProductDB
    {
        // The directory for the images
        static string imagesPath = "C:/Murach/SQL Server 2019/Images/";

        public static void WriteImage(int productID, string imageName)
        {
            SqlConnection connection = null;
            try
            {
                // 1. Read image from file
                string filepath = imagesPath + imageName;
                if (File.Exists(filepath) == false)
                    throw new Exception("File Not Found: " + filepath);
                FileStream sourceStream = new FileStream(
                    filepath,
                    FileMode.OpenOrCreate,
                    FileAccess.Read);

                int streamLength = (int) sourceStream.Length;
                Byte[] productImage = new Byte[streamLength];
                sourceStream.Read(productImage, 0, streamLength);
                sourceStream.Close();

                // 2. Write image to database
                connection = GetConnection();

                SqlCommand command = new SqlCommand();
                command.Connection = connection;
                command.CommandText =
                    "INSERT INTO ProductImages " +
                    "VALUES (@ProductID, @ProductImage)";
                command.Parameters.AddWithValue("@ProductID", productID);
                command.Parameters.AddWithValue("@ProductImage", productImage);

                connection.Open();
                command.ExecuteNonQuery();
            }
            catch (Exception e)
            {
                throw e;
            }
            finally
            {
                if (connection != null)
                    connection.Close();
            }
        }
    }
}
```

Figure 19-5 The ProductDB class for varbinary(max) storage (part 1 of 3)

The ReadImage() method of the ProductDB class returns an array of bytes for the specified image. To start, this method accepts a single parameter named imageID that's used to specify the image to be read. Then, the body of this method begins by getting a connection to the database. Next, it creates a SqlCommand object that contains a SELECT statement that retrieves the ProductImage column for the specified imageID value. This SELECT statement accepts a single parameter named @ImageID. After creating the SqlCommand object, this method adds a parameter to that object with the value that was passed to the method. Finally, this method opens the connection and calls the ExecuteReader() method to execute the SELECT statement that's stored in the SqlCommand object and return a SqlDataReader object.

Once the SqlDataReader object is returned, the ReadImage() method reads the image that's stored within this object. This is easy because the SqlDataReader object only contains a single row and a single column. In other words the SqlDataReader object only contains the binary data for the specified image. To start, the Read() method of the SqlDataReader object is called to move the cursor onto the first and only row in the result set. If the Read() method isn't able to move to this row, an exception is thrown and execution skips into the catch block. Otherwise, the SqlDataReader object is used to return the first column in the result set and to convert it to an array of bytes. Finally, this code closes the SqlDataReader object and returns the array of bytes for the image.

The ProductDB class**Part 2**

```
public static Byte[] ReadImage(int imageID)
{
    SqlConnection connection = null;
    try
    {
        connection = GetConnection();

        SqlCommand command = new SqlCommand();
        command.Connection = connection;
        command.CommandText =
            "SELECT ProductImage " +
            "FROM ProductImages " +
            "WHERE ImageID = @ImageID";
        command.Parameters.AddWithValue("@ImageID", imageID);

        connection.Open();
        SqlDataReader reader = command.ExecuteReader();

        Byte[] imageByteArray = null;
        if (reader.Read() == false)
            throw new Exception("Unable to read image.");
        imageByteArray = (Byte[]) reader[0];
        reader.Close();

        return imageByteArray;
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        if (connection != null)
            connection.Close();
    }
}
```

Figure 19-5 The ProductDB class for varbinary(max) storage (part 2 of 3)

The GetImageIDList() method of the ProductDB class reads all image ID values from the database and returns them as a List<int> object. Although this method doesn't contain any code that's used to work with binary values, it is needed by the form for this application.

To start, the body of this method gets a connection. Then, it creates a SqlCommand object that contains a SELECT statement that returns a list of all image IDs that are stored in the database. Since this SELECT statement doesn't contain any parameters, this SqlCommand object is easy to create. Next, it opens the connection and uses the ExecuteReader() method of the SqlCommand object to return a SqlDataReader object for the result set.

After the SqlDataReader object has been created, the code uses a while loop to read each image ID value that's stored in the reader object and store it in a List<int> object named imageIDList. Finally, this code closes the reader object and returns the List<int> object.

The static GetConnection() method that's used by all three of the other methods in this class returns a SqlConnection object for a SQL Server instance named SqlExpress that's running on the same computer as the ProductDB class. In addition, the code within this method uses the Examples database that's running on the server, and it uses integrated security to connect to this database.

The ProductDB class**Part 3**

```
public static List<int> GetImageIDList()
{
    SqlConnection connection = null;
    try
    {
        connection = GetConnection();

        SqlCommand command = new SqlCommand();
        command.Connection = connection;
        command.CommandText =
            "SELECT ImageID FROM ProductImages " +
            "ORDER BY ImageID";

        connection.Open();
        SqlDataReader reader = command.ExecuteReader();

        List<int> imageIDList = new List<int>();
        while (reader.Read())
        {
            int imageID = (int)reader[0];
            imageIDList.Add(imageID);
        }
        reader.Close();

        return imageIDList;
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        if (connection != null)
            connection.Close();
    }
}

public static SqlConnection GetConnection()
{
    SqlConnection connection = new SqlConnection();
    connection.ConnectionString =
        "Data Source=localhost\\SqlExpress;" +
        "Initial Catalog=Examples;Integrated Security=True";
    return connection;
}
```

Figure 19-5 The ProductDB class for varbinary(max) storage (part 3 of 3)

How to use FILESTREAM storage

Now that you know how to use a varbinary(max) column to store binary data, you're ready to learn how to add FILESTREAM support to a varbinary(max) column. Although adding FILESTREAM support adds complexity to the database and to the applications that work with the binary data in the database, remember that this provides two benefits. First, it lets you store BLOBs that are larger than 2GB. Second, it improves performance, especially for BLOBs that are larger than 1MB.

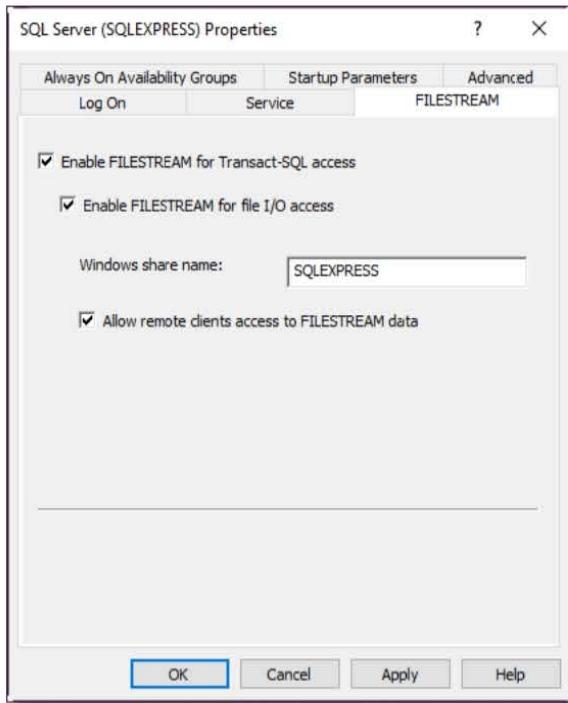
How to enable FILESTREAM storage on the server

By default, FILESTREAM storage is disabled for the server. As a result, if you want to use FILESTREAM storage, you must enable it for the server as shown in figure 19-6. Here, I enabled all three levels of FILESTREAM access for the only instance of SQL Server that's running on my computer, which is named SQLEXPRESS.

If you want to allow a .NET application to work with the database, you need to allow at least the first two levels. In other words, you need to select the “Enable FILESTREAM for Transact-SQL access” check box, and you need to select the “Enable FILESTREAM for file I/O access” check box. For some applications, that's all you need to do. However, if you want to allow remote clients to access the FILESTREAM data, you'll also need to select the “Allow remote clients access to FILESTREAM data” check box.

After you set the server properties for enabling FILESTREAM storage, you have to set the filestream_access_level server configuration option before you can access FILESTREAM data. To do that, you execute the sp_configure stored procedure as shown in this figure. Here, the access level is set to 2 so FILESTREAM storage can be used from both Transact-SQL and Windows applications. Then, the RECONFIGURE statement applies the new setting to the server instance.

The FILESTREAM tab of the SQL Server Properties dialog box



How to enable FILESTREAM storage

1. Start the SQL Server Configuration Manager tool, and select the SQL Server Services node to display the services that are available to your computer.
2. Right-click on the instance of SQL Server that you want to use and select the Properties command to display the Properties dialog box.
3. Select the FILESTREAM tab.
4. Select the “Enable FILESTREAM for Transact-SQL access” check box.
 - If you want to read and write FILESTREAM data from Windows, select the “Enable FILESTREAM for file I/O access” check box and enter the name of the Windows share in the Windows Share Name box.
 - If you want to allow remote clients to access the FILESTREAM data, select the “Allow remote clients access to FILESTREAM data” check box.
5. Select OK.
6. Execute these statements in the Management Studio:

```
EXEC sp_configure filestream_access_level, 2;
RECONFIGURE;
```
7. Use the Configuration Manager to stop and then restart the SQL Server service.
8. Close and reopen the Management Studio.

Description

- By default, FILESTREAM storage is disabled for the server.

Figure 19-6 How to enable FILESTREAM storage on the server

How to create a database with FILESTREAM storage

Before you can use FILESTREAM storage, you must create a database that provides for it. To do that, you can use the CREATE DATABASE statement as shown in figure 19-7. This CREATE DATABASE statement works somewhat like the CREATE DATABASE statements described in chapter 11, but it also specifies a file group that provides for FILESTREAM storage.

To start, you specify a name for the database, and you specify the primary data file (an mdf file) for the database. In this figure, for example, the statement creates a database named MusicStore with a primary data file named MusicStore.mdf.

To provide for FILESTREAM storage, you must also specify a file group for the files that have FILESTREAM access. To do that, you type a comma after the closing parenthesis for the primary data file. Then, you code the FILEGROUP keyword, followed by a name for the file group. Next, you code the CONTAINS FILESTREAM DEFAULT keywords, followed by a set of parentheses. Within the parentheses, you code a name for the file group directory, and you code a path to the directory that will store the binary files. In this figure, for example, the binary files will be stored in the MusicStore_images directory.

If you prefer using the Management Studio to create a database as described in chapter 12, you can also use that tool to create a database that provides for FILESTREAM storage. In that case, you can use the New Database dialog box to add a FILESTREAM file group. Once you understand the code in this figure, you shouldn't have any trouble doing that.

How to create a database with FILESTREAM storage

```
CREATE DATABASE MusicStore
ON PRIMARY
(
    NAME = MusicStore,
    FILENAME = 'C:\Murach\SQL Server 2019\Datasets\MusicStore.mdf'
),
FILEGROUP FileStreamImages CONTAINS FILESTREAM DEFAULT
(
    NAME = MusicStoreImages,
    FILENAME = 'C:\Murach\SQL Server 2019\Datasets\MusicStore_images'
);
```

Description

- If you want to use the FILESTREAM feature, you must create a database that includes a file group that provides for FILESTREAM storage.

Figure 19-7 How to create a database with FILESTREAM storage

How to create a table with a FILESTREAM column

Once you've created a database that provides for FILESTREAM storage, you must create a table that provides for FILESTREAM storage as shown in figure 19-8. If you compare this table with the table shown in figure 19-2, you'll see that they're similar. However, the ProductImage column includes the FILESTREAM attribute that enables FILESTREAM storage for this column.

In addition, this table includes a column named RowID. This column stores a *globally unique identifier (GUID)*, which is a value that's unique within the current database and other networked versions of the database around the globe. This column is required for FILESTREAM storage, and SQL Server uses it to locate the file that stores the data for the ProductImage column. Note that this column uses the uniqueidentifier data type and the ROWGUIDCOL property. In addition, this column uses the NEWID function to return a globally unique value for the column. As a result, if you don't specify a value for this column, the NEWID function will automatically return a value.

How to insert, update, and delete FILESTREAM data

Once you create a table that provides for FILESTREAM storage, you can use INSERT, UPDATE, and DELETE statements just as you would for other types of data. However, you may need to use the NEWID function to return a globally unique value for the GUID column. In this figure, for example, the three INSERT statements insert three rows into the ProductImages table. Here, the first statement doesn't specify a value for the GUID column. As a result, the table uses the NEWID function to generate this value. By contrast, the second and third statements use the NEWID function explicitly.

How to retrieve FILESTREAM data

To retrieve FILESTREAM data, you can use a SELECT statement just as you would for other types of data. In this figure, for example, the first SELECT statement selects all columns and rows from the ProductImages table. This shows the data that was inserted by the three INSERT statements. Here, the first two rows in the ProductImage column store a binary value of zero and the third row stores a longer binary value.

When you use FILESTREAM storage, you can use the PathName function to return the path to the binary file stream. Unlike most functions, this function is case sensitive. As a result, you must use the capitalization shown in the second SELECT statement. Here, the SELECT statement displays the ImageID value and the path to the binary file stream. In the next figure, you'll learn how to use this function in a .NET data access class to read and write binary data to the file stream that's returned by this method.

When you use an INSERT statement to insert a row that contains FILESTREAM data, it's important to insert a zero value instead of a NULL value to initialize the FILESTREAM column. Otherwise, the PathName function

How to create a table with FILESTREAM storage

```
CREATE TABLE ProductImages
(
    ImageID int PRIMARY KEY IDENTITY,
    ProductID int NOT NULL,
    RowID uniqueidentifier ROWGUIDCOL NOT NULL UNIQUE DEFAULT NEWID(),
    ProductImage varbinary(max) FILESTREAM NOT NULL
);
```

Three INSERT statements that insert rows into the table

```
INSERT INTO ProductImages (ProductID, ProductImage)
VALUES (1, 0);

INSERT INTO ProductImages
VALUES (2, NEWID(), 0);

INSERT INTO ProductImages
VALUES (3, NEWID(), CAST('0123456789ABC' AS varbinary(max)));
```

A statement that displays the values in the table

```
SELECT * FROM ProductImages;
```

The result set

	ImageID	ProductID	RowID	ProductImage
1	1	1	9FC421D5-F0B6-4BE5-8F6C-AFED731BAA66	0x00000000
2	2	2	6142A789-154F-45E5-9A9A-6AB318177387	0x00000000
3	3	3	ECF80860-28FB-414D-809A-99C3C46FD1B9	0x30313233343536373839414243

A SELECT statement that displays the filepath

```
SELECT ImageID, ProductImage.PathName() AS FileStreamPath
FROM ProductImages;
```

The result set

	ImageID	FileStreamPath
1	1	\MMA1\SQLEXPRESS\w02-A60EC2F8-2B24-11DF-9CC3-AF2E56D89593\Music Store\dbo\Product...
2	2	\MMA1\SQLEXPRESS\w02-A60EC2F8-2B24-11DF-9CC3-AF2E56D89593\Music Store\dbo\Product...
3	3	\MMA1\SQLEXPRESS\w02-A60EC2F8-2B24-11DF-9CC3-AF2E56D89593\Music Store\dbo\Product...

Description

- To create a table that provides for FILESTREAM storage, you include a column definition that contains the FILESTREAM keyword. In addition, you create a column with a *globally unique identifier (GUID)* that's used to locate the file that stores the FILESTREAM data.
- A GUID is a value that's unique within the current database and other networked versions of the database around the globe. To define a column that contains a GUID, you specify the uniqueidentifier data type for the column, and you specify the ROWGUIDCOL property for the column.
- You can use the NEWID function to generate a globally unique value.
- You can use the PathName function to return the path to the binary file stream. This function is case sensitive, so you must use exact capitalization.

Figure 19-8 How to create a table and insert FILESTREAM data

won't return a path to the file stream, and the application won't be able to write data to the file stream.

A data access class that uses FILESTREAM storage

Figure 19-9 shows a data access class named ProductDB that uses FILESTREAM storage. If you compare this class with the ProductDB class presented in figure 19-5, you'll see that it contains the same methods and performs the same tasks. As a result, you can use the ProductDB class shown in this figure with an application like the Product Image Manager application that was presented earlier in this chapter.

Within the ProductDB class, the code begins by defining a string for the directory for the images. Then, the GetConnection() method returns a SqlConnection object. Since this works like the GetConnection() method presented earlier in this chapter, you shouldn't have much trouble understanding how it works. However, the method shown in this figure uses the MusicStore database that was created by the SQL statement presented in figure 19-7.

The WriteImage() method works much like the WriteImage() method presented earlier in this chapter. However, since this method uses FILESTREAM storage, it's more complex. For instance, you must use the NewGuid() method of the Guid class to return a globally unique identifier for the row. This has the same effect as using the NEWID function in SQL, but it allows you to use the globally unique identifier again later in this method.

The ProductDB class**Part 1**

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.IO;

namespace MusicStoreImageManager
{
    class ProductDB
    {
        // define the directory for the images
        static string imagesPath = "C:/Murach/SQL Server 2019/Images/";

        public static SqlConnection GetConnection() {
            SqlConnection connection = new SqlConnection();
            connection.ConnectionString =
                "Data Source=localhost\\SqlExpress;" +
                "Initial Catalog=MusicStore;Integrated Security=True";
            return connection;
        }

        public static void WriteImage(int productID, string imageName) {
            SqlConnection connection = null;
            SqlTransaction transaction = null;
            try {
                // 1. Set up the input stream from the image file
                string filepath = imagesPath + imageName;
                if (File.Exists(filepath) == false)
                    throw new Exception("File Not Found: " + filepath);

                FileStream sourceStream = new FileStream(
                    filepath,
                    FileMode.Open,
                    FileAccess.Read);

                // 2. Initialize the row in the table
                connection = GetConnection();

                SqlCommand command = new SqlCommand();
                command.Connection = connection;
                command.CommandText =
                    "INSERT INTO ProductImages " +
                    "VALUES (@ProductID, " +
                    "        CAST(@RowID AS uniqueidentifier), 0)";

                Guid rowID = Guid.NewGuid();
                command.Parameters.AddWithValue("@ProductID", productID);
                command.Parameters.AddWithValue("@RowID", rowID);

                connection.Open();
                command.ExecuteNonQuery();
            }
        }
    }
}
```

Figure 19-9 The ProductDB class for FILESTREAM storage (part 1 of 3)

After executing the query that inserts the row into the database, you must get a reference to the file stream for the BLOB. To do that, you begin by defining a SELECT statement that returns two columns. The first column uses the PathName function to return the path to the file stream. The second column uses the GET_FILESTREAM_TRANSACTION_CONTEXT function to get the context for the transaction. Note that the GUID value that was created earlier in this method is used in the WHERE clause of this SELECT statement to specify the row that was inserted by the INSERT statement earlier in the same method.

After the SELECT statement is defined, this code executes this statement. Then, it uses the reader object that's returned to check whether the result set contains data. If so, the first column is stored in a variable named path, and the second column is stored in a variable named context. Finally, this code closes the reader object.

At this point, the WriteImage() method has all the data it needs to set up an output stream to the database. To do that, it creates a FileStream object with write access.

Now that the WriteImage() method has an input stream and an output stream, it's ready to read data from the input stream (the image file) and write data to the output stream (the BLOB in the database). To do that, this code defines a block size of half a megabyte (524,288 bytes), a size that's usually efficient for working with streams. Then, this code defines a buffer variable that stores this array of bytes. Finally, it uses a loop to read from the input stream and write to the output stream.

Note that this allows this class to read half of a megabyte into memory at a time. For example, let's assume that you have an image file that's 4 megabytes. In that case, it would take eight trips through the loop to write the file to the database. On the other hand, the WriteImage() method presented earlier in this chapter reads the entire file into memory before it begins to write the file. As a result, it can use a lot of memory if you use it with images that are larger than 1 megabyte.

Note also that this method uses a transaction. If the method completes successfully, the last statement in the try block commits the transaction, and the image is written to the database. However, if the method encounters an error, the catch block rolls back the transaction, and the image is not written to the database.

The ProductDB class**Part 2**

```
// 3. Get a reference to the BLOB
transaction = connection.BeginTransaction();
command.Transaction = transaction;
command.CommandText =
    "SELECT ProductImage.PathName(), " +
    "      GET_FILESTREAM_TRANSACTION_CONTEXT() " +
    "FROM ProductImages " +
    "WHERE RowID = CAST(@RowID AS uniqueidentifier)";
command.Parameters.Clear();
command.Parameters.AddWithValue("@RowID", rowID);

SqlDataReader reader = command.ExecuteReader();
if (reader.Read() == false)
    throw new Exception(
        "Unable to get path and context for BLOB.");
string path = (string)reader[0];
byte[] context = (byte[])reader[1];
reader.Close();

// 4. Set up the output stream to the database
SqlFileStream targetStream =
    new SqlFileStream(path, context, FileAccess.Write);

// 5. Read from file and write to database
int blockSize = 1024 * 512;
byte[] buffer = new byte[blockSize];
int bytesRead = sourceStream.Read(buffer, 0, buffer.Length);
while (bytesRead > 0) {
    targetStream.Write(buffer, 0, bytesRead);
    bytesRead = sourceStream.Read(buffer, 0, buffer.Length);
}

targetStream.Close();
sourceStream.Close();
transaction.Commit();
}
catch (Exception e) {
    if (transaction != null)
        transaction.Rollback();
    throw e;
}
finally {
    if (connection != null)
        connection.Close();
}
}
```

Figure 19-9 The ProductDB class for FILESTREAM storage (part 2 of 3)

The ReadImage() method works much like the ReadImage() method presented earlier in this chapter. However, since this method uses FILESTREAM storage, it's more complex. To start, this method executes a SELECT statement to get a path to the file stream and the transaction context. Then, it sets up a file stream for the BLOB by creating a FileStream object with read access. Since this works much like the SELECT statement of the WriteImage() method that's presented in part 2 of this figure, you shouldn't have much trouble understanding how it works.

After setting up the input file stream, this code uses a loop to read the binary data from the database and stores this data in a List<byte> object named imageBytes. Then, it converts the List<byte> object to an array of byte values and returns it.

This loop works similarly to the loop in the WriteImage() method that reads from one stream and writes to another. However, instead of writing to a stream, this code stores the entire stream in the List<byte> object. The advantage to this approach is that it allows you to separate the data access layer (the ProductDB class) from the presentation layer (the form). The disadvantage of this approach is that the entire image is stored in memory. If this isn't satisfactory for your application, you can add a PictureBox control as a second argument of the ReadImage() method. Then, this method can stream the data from the BLOB in the database to the PictureBox control that displays the image.

For this ProductDB class to work with the Product Image Manager application presented earlier in this chapter, it must include a GetImageIDList() method. However, the code for this method is the same as the code presented in part 3 of figure 19-5. To save space, it isn't presented here.

The ProductDB class**Part 3**

```
public static Byte[] ReadImage(int imageID) {
    SqlConnection connection = null;
    SqlTransaction transaction = null;
    try {
        connection = GetConnection();
        connection.Open();
        transaction = connection.BeginTransaction();

        SqlCommand command = new SqlCommand();
        command.Connection = connection;
        command.Transaction = transaction;
        command.CommandText =
            "SELECT ProductImage.PathName(), " +
            "       GET_FILESTREAM_TRANSACTION_CONTEXT() " +
            "FROM ProductImages " +
            "WHERE ImageID = @ImageID";
        command.Parameters.AddWithValue("@ImageID", imageID);

        SqlDataReader reader = command.ExecuteReader();
        if (reader.Read() == false)
            throw new Exception(
                "Unable to get path and context for BLOB.");
        string path = (string)reader[0];
        byte[] context = (byte[])reader[1];
        reader.Close();

        FileStream sourceStream =
            new FileStream(path, context, FileAccess.Read);

        int blockSize = 1024 * 512;
        byte[] buffer = new byte[blockSize];
        List<byte> imageBytes = new List<byte>();
        int bytesRead = sourceStream.Read(buffer, 0, buffer.Length);
        while (bytesRead > 0) {
            bytesRead = sourceStream.Read(buffer, 0, buffer.Length);
            foreach(byte b in buffer)
                imageBytes.Add(b);
        }
        sourceStream.Close();
        return imageBytes.ToArray();
    }
    catch (Exception e) {
        throw e;
    }
    finally {
        if (connection != null)
            connection.Close();
    }
}
public static List<int> GetImageIDList() {
    // same as part 3 of figure 19-5
}
```

Figure 19-9 The ProductDB class for FILESTREAM storage (part 3 of 3)

Perspective

In this chapter, you learned how to write BLOBS to a database table and to read BLOBS from a database table. In addition, you learned how to use the FILESTREAM storage feature. At this point, you have the core concepts and skills for working with BLOBS.

Now, if you want to develop a more sophisticated application for working with BLOBS, you should be able to do that. For example, you may want to enhance the application presented in this chapter so it can be used to update or delete an existing image. Or, you may want to enhance this application so it works more efficiently.

Terms

binary large object (BLOB)
FILESTREAM storage
globally unique identifier (GUID)

Exercises

1. Modify the first ProductDB class presented in this chapter so the ReadImage() method accepts a PictureBox control as a second argument like this:

```
public static void ReadImage(int imageID, PictureBox pictureBox)
```

Then, modify the ReadImage() method so it streams data from the database to the PictureBox control, and modify the code for the form so it works with this new method. To do this, you can begin by copying code from the form into the ProductDB class.

2. Modify the second ProductDB class so it works as described in exercise 1.

Appendix A

How to set up your computer for this book

To run the SQL statements described in chapters 1 through 18 of this book, you only need to have two software products installed: the SQL Server 2019 database engine and the SQL Server Management Studio (SSMS). Both of these products are available from Microsoft's website for free, and you can download and install them both on your computer as described in this appendix.

Once you install these software products, you can install the files for this book. To do that, you can download these files from www.murach.com. Then, you can use the Management Studio to create the databases for this book. After that, you can start experimenting with the SQL scripts for this book.

To use a .NET language such as C# or VB to work with BLOBs as described in chapter 19, you can use Visual Studio. If you don't already have Visual Studio installed on your system, you can install Visual Studio 2019 Community as described in this appendix. This product is also available from Microsoft's website for free.

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

Three editions of SQL Server 2019 Express

Figure A-1 describes three editions of SQL Server 2019 Express. Of the three editions listed in this figure, we recommend that you install SQL Server 2019 Express. This edition includes the database engine that provides for all of the features covered in this book.

The Express with Advanced Services edition includes all of the features of the Express edition, plus two additional features that aren't covered in this book. If you eventually want to learn about these features, you can install this edition and use it with this book. However, it requires more system resources than the Express edition.

The LocalDB edition also has the same features as the Express edition, but it's designed to be embedded within an application. As a result, you should be aware of this edition in case you ever need to embed a database into an application. It has the same features as the Express edition, but it doesn't accept remote connections and can't be administered remotely.

The tool for working with all editions of SQL Server

This figure also describes a tool named SQL Server Management Studio (SSMS). You can use this tool to work with any edition of SQL Server, including non-Express editions of SQL Server such as the Enterprise edition.

Three editions of SQL Server 2019 Express

Edition	Description
LocalDB	A lightweight version of Express that can be embedded into an application. Doesn't accept remote connections and can't be administered remotely.
Express	The core Express database server. Contains only the database engine. Accepts remote connections and can be administered remotely.
Express with Advanced Services	Contains the Full Text Search and Reporting Services features in addition to the database engine. These features aren't covered in this book.

The tool for working with all editions of SQL Server

Tool	Description
SQL Server Management Studio (SSMS)	You can use the Management Studio to connect to SQL Server and work with its databases.

Description

- For this book, we recommend that you install the Express edition of SQL Server 2019, but you can install the Express with Advanced Services edition if you want to install the advanced features and don't mind a larger download and install.

How to install SQL Server 2019 Express

Figure A-2 shows how to install SQL Server 2019 Express. Before you get started, you should know that SQL Server 2019 only runs on Windows 10 and later. As a result, if you’re still using an older version of Windows, such as Windows 8, you need to upgrade to Windows 10 or later.

If you don’t already have an instance of SQL Server Express installed on your system, the procedure in this figure installs SQL Server 2019 Express with an instance name of SQLEXPRESS, which is what you want for this book. However, it’s possible that you may have an older instance of SQL Server Express, such as SQL Server 2016 Express, on your computer. If you do, the older instance of SQL Server is probably named SQLEXPRESS. As a result, the 2019 instance can’t use this name.

In that case, the procedure in this figure leaves the old instance of SQL Server on your computer and installs the 2019 instance of the database engine with a new name such as SQLEXPRESS01. That’s what the Basic installation does automatically. However, the examples in this book assume that SQL Server 2019 Express has an instance name of SQLEXPRESS. As a result, if you choose this approach, you may have to modify some of the examples in this book to get them to run successfully on your computer.

If an older instance of SQL Server is installed on your computer, you also have two other options. First, you may be able to upgrade the database server by selecting the Custom installation in step 4 instead of selecting the Basic installation. Then, you can select the “Upgrade” option. This will upgrade the old instance to 2019, but it will run the existing databases as if they are running on the earlier version. In that case, if you want to update a database so it can use the features of SQL Server 2019, you can change its compatibility level. For information on how to do that, see chapter 2.

Second, if you can’t upgrade the old instance of SQL Server Express, you can uninstall it. Then, you can install a 2019 instance. That way, you can use the default name of SQLEXPRESS for the 2019 instance. To do that, you should start by backing up any databases that are running on the older SQL Server instance. One way to do that is to detach them from the server and copy the data (mdf) and log (ldf) files for the databases to a safe location. Then, you can uninstall all components of SQL Server Express, including the Management Studio components. Next, you can install SQL Server 2019 Express as described in this figure. Finally, you can attach the databases that you backed up to this server. For more information about detaching and attaching databases, see chapter 2.

How to install SQL Server Management Studio

This figure also shows how to install SQL Server Management Studio (SSMS), the main tool for working with databases. If you already have SQL Server 2019 Express installed on your computer, you only need to install Management Studio. To do that, you can skip the steps for installing SQL Server and just follow the steps for installing the Management Studio.

How to install SQL Server 2019 Express

1. Search the Internet for “SQL Server 2019 Express download”.
2. Follow the links to the official download page for SQL Server 2019 Express at Microsoft’s website (www.microsoft.com).
3. Download the setup program for SQL Server 2019 Express. This program should be stored in a file named SQL2019-SSEI-Expr.exe.
4. Start the setup program and select the Basic installation. Then, respond to the resulting prompts and dialogs.
5. At the last dialog (the “Installation has completed successfully” dialog), make a note of the instance name for the server.
 - If SQL Server Express wasn’t already installed on your computer, the instance name should be SQLEXPRESS.
 - If SQL Server Express was already installed on your computer, the instance name might be slightly different such as SQLEXPRESS01.
6. At the last dialog, click the Install SSMS button to go to the web page for downloading SQL Server Management Studio.

How to install the Management Studio

7. At the web page for downloading the Management Studio, follow the links to download the setup program. This program should be stored in a file named SSMS-Setup-ENU.exe.
8. Start the setup program and respond to the resulting prompts and dialogs.

Notes

- This book assumes that an instance of SQL Server 2019 Express is installed with a name of SQLEXPRESS. If it’s installed on your system with a different name, you may have to modify some of the examples to get them to run successfully on your computer.
- SQL Server 2019 supports Windows 10 and later. Support for Windows 8 has been dropped.

Chapter 2 presents the basic techniques for using the Management Studio. You can use it to develop and run all of the SQL statements in this book.

How to install the files for this book

Figure A-3 begins by describing the files for this book that are contained in the self-extracting zip file (an exe file) that you can download from www.murach.com. When you download and execute this zip file, it will unzip the five directories described in this figure into this directory:

C:\Murach\SQL Server 2019

The Databases directory contains the SQL scripts used to create the three databases that are used throughout the book. To do that, you can use the Management Studio to open and execute these scripts as described in the next figure.

The Scripts directory contains the SQL code that's described throughout this book. You can use the Management Studio to open these scripts. Then, you can run them to view the results. Or, you can experiment with these scripts by modifying them before you run them.

The Exercises directory contains the solutions to the exercises that are presented at the end of each chapter. You can use these solutions to check that the solutions you develop are correct. You can also use these solutions to find out how to solve an exercise if you're unable to do it on your own. Keep in mind, though, that you'll get more out of the exercises if you try to solve them on your own first.

The Projects directory contains two subdirectories that contain the Visual Studio projects for chapters 1 and 19. These projects are available in two versions: a C# version and a Visual Basic version. You can use Visual Studio Community to open these projects.

The files for this book

Directory	Description
Databases	The SQL scripts that create the databases for this book.
Scripts	The SQL scripts for the examples shown throughout this book.
Exercises	The solutions to the exercises at the end of each chapter.
Projects	The Visual Studio projects for chapters 1 and 19. These projects use C# and Visual Basic code to work with SQL Server.
Images	The image files that are used by the application presented in chapter 19.

The databases for this book

Database	Description
AP	The Accounts Payable (AP) database that's used in the examples throughout this book. This database only includes tables so you can add other objects such as views and stored procedures yourself.
ProductOrders	The Product Orders database that's used in some of the examples in this book.
Examples	A database that contains several small tables that are used in some of the examples for which the AP database couldn't be used.

How to install the files for this book

1. Go to www.murach.com.
2. Find the page for *Murach's SQL Server 2019 for Developers*.
3. If necessary, scroll down. Then, click the “FREE Downloads” tab.
4. Click the link to download the exe file for the book examples and exercises. Then, respond to the resulting pages and dialog boxes. This should download an installer file named sq19_allfiles.exe.
5. Double-click this file and respond to the dialog boxes that follow. If you accept the defaults, this installs the files into the directory shown below.

The default installation directory

C:\Murach\SQL Server 2019

Description

- All of the files for the databases and code described in this book are contained in a self-extracting zip file (an exe file) that can be downloaded from www.murach.com.

How to create the databases for this book

Before you can run the SQL statements presented in this book, you need to create the three databases described in the previous figure. The easiest way to do that is to use the SQL Server Management Studio to run the SQL scripts that create the databases. For example, the `create_ap.sql` script creates the AP database. The procedure for running these scripts is described in figure A-4.

To determine if a script ran successfully, you can review the results in the Messages tab. In this figure, for example, the Messages tab shows a series of statements that have executed successfully. In addition, the Object Explorer window shows the three databases.

If the script encounters problems, the SQL Server Management Studio displays one or more errors in the Messages tab. Then, you can read these errors to figure out why the script didn't execute correctly.

Before you can run the SQL scripts that create the databases, the database server must be running. By default, the database server is automatically started when you start your computer, so this usually isn't a problem. However, if it isn't running on your system, you can start it as described in chapter 2.

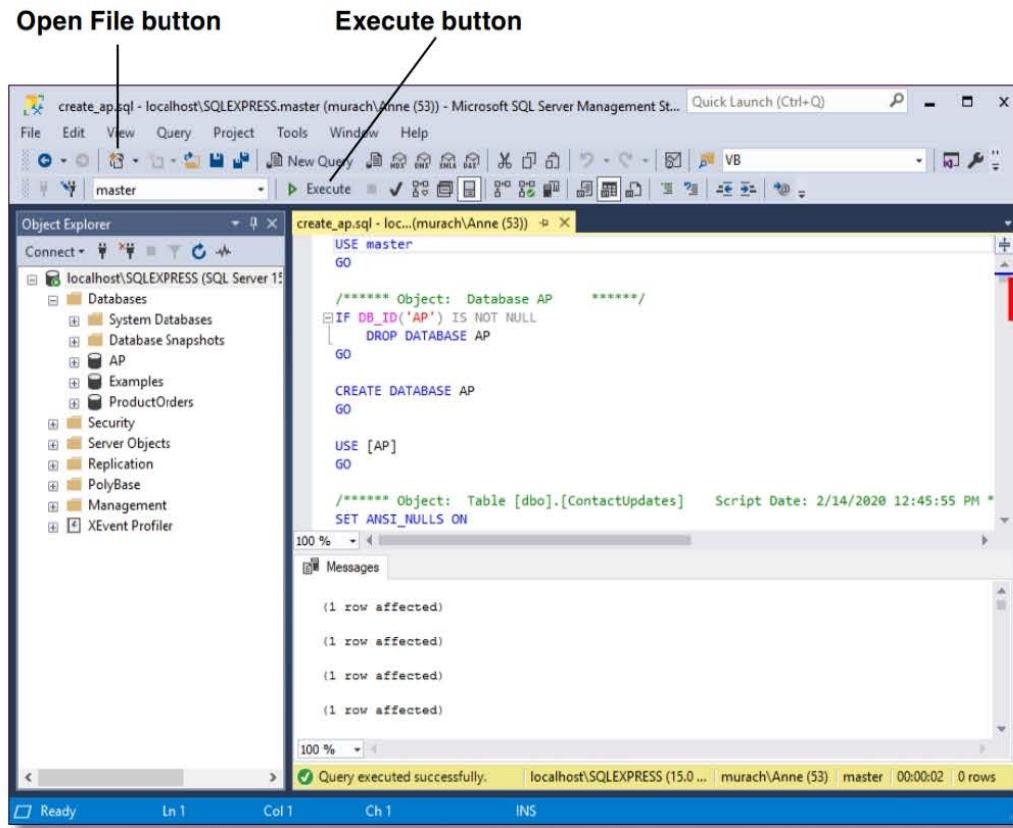
How to restore the databases for this book

As you work with the examples in this book, you may make changes to the databases or tables that you don't intend to make. In that case, you may want to restore a database to its original state. To do that, you can run the script that creates the database again. This will drop the database and recreate it.

The directory that contains the scripts for creating the databases

C:\Murach\SQL Server 2019\Datasets

The Management Studio after executing the three database scripts



How to create the databases

1. Start the SQL Server Management Studio.
2. Connect to the database server as shown in chapter 2.
3. Open a script file by clicking the Open File button and then using the resulting dialog box to locate the script that creates the database. To create the AP database, for example, open the create_ap.sql file. When you do, the Management Studio displays this script in a Query window.
4. Execute the script by clicking the Execute button. When you do, the Messages tab indicates whether the script executed successfully.
5. Repeat steps 3 and 4 until you have created all three databases.

How to restore a database

- Run the create database script again to drop the database and recreate it.

Description

- For these scripts to work, the database server must be running. By default, the database server is automatically started when you start your system. If it isn't running on your system, you can start it as described in chapter 2.

Figure A-4 How to create and restore the databases for this book

How to install Visual Studio 2019 Community

You only need Visual Studio for chapter 19 of this book. As a result, if Visual Studio isn't already installed on your computer, you can wait until you get to chapter 19 to install it. Then, you can install Visual Studio 2019 Community as shown in figure A-5. This edition of Visual Studio is available for free from Microsoft's website.

How to install Visual Studio 2019 Community

1. Search the Internet for “Visual Studio 2019 Community download”.
2. Follow the links to the download page for Visual Studio 2019 Community at the Visual Studio website (visualstudio.microsoft.com).
3. Follow the directions to download the setup program for Visual Studio 2019 Community.
4. Run the setup program, select the “.NET desktop development” workload, and click the Install button.

Description

- You only need to install Visual Studio for chapter 19 of this book. If you don’t already have Visual Studio installed on your system, you can install Visual Studio Community, which is available for free from Microsoft.

Index

-- characters (comment), 32, 33
 - operator (subtraction), 97
 - wildcard character, 112, 113
 # character (local temporary name), 426, 427, 462, 463
 ## characters (global temporary name), 426, 427, 462,
 463
 % operator, 97
 % wildcard character, 112, 113
 * operator
 all columns, 90, 91
 multiplication, 97
 *= operator, 140, 141
 .bak file, 60, 61
 .NET data provider, 40, 41
 / operator, 97
 /*...*/ characters (block comment), 32, 33
 :: scope qualifier, 552, 553
 @ character
 parameter name, 464, 465
 variable name, 422, 423
 @@ERROR system function, 442, 443
 @@FETCH_STATUS system function, 436, 437
 @@IDENTITY system function, 442, 443
 @@ROWCOUNT system function, 442, 443, 512, 513
 @@SERVERNAME system function, 442, 443
 @@TRANCOUNT system function, 514, 515
 [] characters
 delimiter, 336, 337
 in column name, 92, 93
 wildcard, 112, 113
 ^ wildcard character, 112, 113
 wildcard character, 112, 113
 + operator
 addition, 97
 concatenation, 94, 95
 < operator, 104, 105
 <= operator, 104, 105
 <> operator, 104, 105, 444, 445
 = operator
 assign column alias, 92, 93
 assign variable value, 423
 comparison, 104, 105, 114, 115, 444, 445
 rename column, 92, 93
 != operator, 140, 141
 > operator, 104, 105
 >= operator, 104, 105
 1NF, 2NF, 3NF, 4NF, 5NF, 6NF, 320, 321

A

ABS function, 268, 269
 Action query, 30, 31
 Ad hoc relationship, 126, 127
 Addition operator, 97
 ADO.NET, 38-43
 with C# code, 44, 45
 with Visual Basic code, 42, 43
 AFTER trigger, 494-497
 Aggregate
 in view, 404, 405
 query, 160-163
 Aggregate function, 160-163
 AIX operating system, 21
 Alias
 column, 92, 93
 column and ORDER BY, 118, 119
 table, 128, 129
 All columns operator, 90, 91
 ALL keyword, 90, 91
 and subquery, 192, 193
 in aggregate function, 160-163
 in SELECT clause, 100, 101
 with union, 150, 151
 ALL SERVER keyword, 502, 503
 ALTER COLUMN clause, 354, 355
 ALTER FUNCTION statement, 335, 492, 493
 ALTER LOGIN statement, 542, 543
 ALTER PROC statement, 335, 480, 481
 ALTER ROLE statement, 564-597
 ALTER SCHEMA statement, 546, 547
 ALTER SEQUENCE statement, 358, 359
 ALTER SERVER ROLE statement, 558-561
 ALTER TABLE statement, 25, 335, 352-355
 ALTER TRIGGER statement, 335, 504, 505
 ALTER USER statement, 544, 545
 ALTER VIEW statement, 335, 406, 407
 Always Encrypted feature, 584
 Ambiguous column name, 126, 127
 American National Standards Institute (ANSI), 18, 19
 American Standard Code for Information Interchange,
 see ASCII
 Analytic functions, 296-299
 Anchor member (recursive CTE), 210, 211
 AND operator, 106, 107
 ANSI, 18, 19
 ANSI/ISO SQL, 18, 19
 ANSI_NULLS system option, 444, 445
 ANSI_PADDING system option, 445
 ANSI-standard data type, 240, 241
 ANSI-standard SQL, 18, 19
 ANY keyword, 194, 195
 AP database (script), 368-371
 API, 6, 7
 Application
 server, 8, 9
 web, 8, 9
 Application program, 38-45
 Application programming interface (API), 6, 7
 Application role, 572, 573
 Application software, 6, 7
 Approximate numeric data type, 242, 243
 Argument, 98, 99
 Arithmetic expression, 96, 97
 Arithmetic operator, 96, 97
 AS keyword
 in CAST function, 252, 253
 in FROM clause, 128, 129
 in SELECT clause, 90-93
 ASC keyword (*ORDER BY*), 116, 117

ASCII, 244
 character set, 360, 361
 control character, 258, 259
 function, 258, 259
 Associate table, 312, 313
 Attach database, 58, 59, 338, 339
 Attribute
 SQL, 10, 11, 304, 305, 340, 341
 XML, 588, 589
 Authentication
 login, 54, 55
 mode, 538, 539
 AUTHORIZATION clause, 546, 547
 AUTO keyword (FOR XML clause), 610-613
 Autocommit mode, 512, 513
 AVG function, 160-163

B

Back end, 6, 7
 Back up database, 60, 61
 Base table, 26, 27, 396, 397
 Base view, 412, 413
 Batch, 368, 369, 418, 419
 stored procedure, 462, 463
 BCNF, 320, 321
 bcp, 527, 528
 BEGIN keyword, 421
 BEGIN TRAN statement, 512, 513
 BEGIN...END block, 430, 431, 486, 487
 BEGIN...END statement, 420, 421, 440, 441
 BETWEEN phrase, 110, 111
 bigint data type, 242, 243
 Binary data, 620-645
 Binary file pointer, 620, 621
 Binary large objects (BLOBS), 620-645
 binary varying data type, 241
 Bit, 244, 361
 bit data type, 242, 243
 BLOBS, 620-645
 Block comment, 32, 33
 Book files (downloading), 650, 651
 Boolean expression, 86, 87
 Boyce-Codd normal form, 320, 321
 BREAK statement, 421, 434, 435
 Browser (web), 8, 9
 Bulk copy program (bcp), 527, 528
 BULK INSERT statement, 527, 528
 Bulk Update (BU) lock, 527, 528
 Business component, 8, 9
 Byte-pair, 245

C

C# code (with ADO.NET), 44, 45
 CACHE keyword, 356, 357
 Calculated column, 26, 27, 90, 91-93
 assigning name, 92, 93
 in view, 402-405
 Call a function, 484, 485
 Call a procedure, 460, 461, 466, 467

Cartesian product, 148, 149
 CASCADE clause
 DENY, 570, 571
 REVOKE, 548, 549
 CASCADE keyword, 350, 351
 Cascading deletes and updates, 350, 351, 380, 381, 499
 CASE function, 284, 285
 Case sensitivity, 104, 105
 CASE software tool, 310, 311
 Cast as data type, 252, 253
 CAST function, 252, 253, 266, 267, 278, 279
 Catalog (system), 412, 413
 Catalog views, 412, 413
 database roles, 568, 569
 server roles, 562, 563
 XML schema information, 608, 609
 CATCH block, 438, 439, 470, 471
 CEILING function, 268, 269
 Cell, 10, 11
 Change script (saving), 390, 391
 char data type, 241, 244, 245
 CHAR function, 258, 259
 char varying data type, 241
 character data type, 241
 character set, 360-361
 character varying data type, 241
 CHARINDEX function, 262-267
 check constraint, 346-349, 384, 385
 CHECK_EXPIRATION option, 540, 541
 CHECK_POLICY option, 540, 541
 Child element (XML), 588, 589
 CHOOSE function, 286, 287
 Client, 4, 5
 software, 6, 7
 Client tools, 50, 51
 Client/server system, 4, 5
 architectures, 7, 8, 9
 compared to file-handling system, 6, 7
 Clustered index, 318, 319, 382, 383
 CLUSTERED keyword, 342, 343
 COALESCE function, 288, 289
 Coarse-grain lock, 524, 525
 Codd, E.F., 10, 18, 19
 Coding guidelines, 32, 33
 COLLATE clause, 366, 367
 Collation, 244, 245, 362-367
 specifying, 366, 367
 viewing, 364, 365
 Collation options, 362, 363
 Collation sets, 362, 363
 Column, 10, 11
 alias, 92, 93
 alias and ORDER BY, 118, 119
 ambiguous name, 126, 127
 attribute, 340, 341
 column-level constraint, 340, 341
 definition, 14, 15, 66, 67
 function, 160, 161
 list (with INSERT), 218, 219
 name, 92, 93
 order, 368, 369
 position (ORDER BY), 118, 119

properties, 66, 67, 378, 379
qualified name, 126, 127
specification, 90, 91
Column-level constraint, 346, 347
Command line switches, 452, 453
Command object (ADO.NET), 40, 41
Command prompt utilities, 452, 453
Comment, 32, 33
Commit a transaction, 510, 511
COMMIT TRAN statement, 512-517
Common table expressions (CTE), 208-211
Comparison operator, 104, 105
Compatibility level (database), 62, 63
Compile, 460, 461
Completion list, 70, 71
Complex query, 204-207
Complex search condition, 170, 171
Component (business), 8, 9
Components (client/server system), 4-7
Composite index, 318, 319
Composite primary key, 10, 11
Compound condition, 106, 107
Compound join condition, 132, 133
Compound search condition, 106, 107, 170, 171
Computer-aided software engineering (CASE), 310, 311
CONCAT function, 262-265
CONCAT_WS function, 262-265
Concatenation, 94, 95
Concurrency, 518-523
defined, 519
problems with, 520, 521
Condition
compound, 106, 107
compound join, 132, 133
join, 126, 127
Conditional processing, 430, 431
Configuration (network), 52, 53
Configuration Manager, 50-53
Conformance with SQL-92, 18, 19
Connect to server, 54, 55
Connecting table, 312, 313
Connection object (ADO.NET), 40, 41
Consistency (data), 500, 501
Constant, *see* Literal
Constraint, 346, 347
column-level, 340, 341
compared to trigger, 500, 501
FOREIGN KEY, 314, 315
NOT NULL, 340, 341
PRIMARY KEY, 340, 341
table-level, 340, 341
UNIQUE, 340, 341
Content (XML), 589
CONTINUE statement, 421, 434, 435
Control character, 258, 259
Control-of-flow language, 36, 37
Conventional file system, 16, 17
Conversion (data type), 250-257
CONVERT function, 98, 99, 254, 255
Core specification (SQL-99 standard), 18, 19
Correlated subquery, 196, 197, 202, 203
Correlation name, 128, 129, 196, 197
COUNT function, 160-163
Covering index, 318
CREATE APPLICATION ROLE statement, 572, 573
CREATE DATABASE statement, 25, 335, 338, 339
CREATE FUNCTION statement, 335, 486-491
CREATE INDEX statement, 25, 335, 342, 343
CREATE LOGIN statement, 540, 541
CREATE PROC statement, 462-467
CREATE PROCEDURE statement, 36, 37, 335
CREATE ROLE statement, 566, 567
CREATE SCHEMA statement, 546, 547
CREATE SEQUENCE statement, 356, 357
CREATE SERVER ROLE statement, 560, 561
CREATE TABLE statement, 25, 335, 340, 341
CREATE TRIGGER statement, 335, 494, 495
CREATE TYPE statement, 478, 479
CREATE USER statement, 544, 545
CREATE VIEW statement, 34, 35, 335, 396, 397, 400-403
CREATE XML SCHEMA COLLECTION statement, 604, 605
Criteria pane, 76, 77
CROSS JOIN keywords, 148, 149
CTE, 208-211
recursive, 210, 211
CUBE operator, 174, 175
CUME_DIST function, 296-299
Cumulative total, 178, 179
Cursor, 436, 437
CYCLE keyword

D

Data
consistency, 500, 501
derived, 326, 327
redundancy, 316, 317
structure, 304, 305, 316, 317, 328, 329
validation, 470-477
Data access API, 6, 7
Data access model, 38, 39
Data adapter (ADO.NET), 40, 41
Data definition language (DDL), 22, 23, 334, 335
Data element, 306-309
subdivide, 308, 309
Data files for a database, 376, 377
Data integrity, 510, 511
Data manipulation language (DML), 22, 23
Data reader, 40, 41
Data table (ADO.NET), 40, 41
Data type, 14, 15, 240-249
and disk storage, 14
and performance, 14
ANSI-standard, 240, 241
comparing expressions, 106
conversion, 250, 251
order of precedence, 250, 251

Database
 administrator, 22, 23
 attach, 58, 59
 back up, 60, 61
 compatibility level, 62, 63
 copy, 216, 217
 create, 338, 339, 376, 377, 654, 655
 definition, 64, 65
 delete, 352, 353, 376, 377
 design, 304-329
 detach, 58, 59
 diagram, 64, 65
 engine, 50-53
 hierarchical, 16, 17
 lock, 524, 525
 name, 130, 131
 network, 16, 17
 object, 16, 334, 432, 433
 relational, 10, 11, 16, 17
 restore, 60, 61, 654, 655
 schema, 400, 401
 schema (of a function), 484, 485
 server, 4, 5, 50, 51, 54, 55
 summarize, 448, 449
 table, 10, 11
 user, 544, 545
 Database access
 grant, 578, 579
 revoke, 578, 579
 DATABASE keyword, 502, 503
 Database management system (DBMS), 6, 7
 Database objects, 24, 25
 navigate, 56, 57
 permissions, 580, 581
 Database permission, 536, 554, 555, 582, 583
 DENY, 570, 571
 Database role, 578, 579
 fixed, 564, 565
 information, 568, 569
 user-defined, 566, 567
 Database system, 20, 21
 DB2, 18, 19
 open-source, 20, 21
 Oracle, 18, 19
 SQL/DS, 18, 19
 Dataset (ADO.NET), 40, 41
 Date
 format, 444, 445
 literal, 104, 105
 part abbreviations, 274, 275
 part values, 274, 275
 search, 280, 281
 date data type, 246, 247
 Date/time data type, 240, 241, 246, 247, 272-275
 DATEADD function, 272-275, 278, 279
 DATEDIFF function, 98, 99, 272, 273, 278, 279
 DATEFORMAT system option, 444, 445
 DATENAME function, 272-277
 DATEPART function, 272-277
 datetime2 data type, 246, 247
 datetimeoffset data type, 246, 247
 DAY function, 272, 273, 276, 277
 DB_ID function, 432, 433
 DB2 database system, 18-21
 DBA, 22, 23
 dbcreator role, 558, 559
 DBMS, 6, 7
 dbo schema, 544, 545
 DDL, 22, 23, 334, 335
 DDL statement, 24, 25
 Deadlock, 530, 531
 preventing, 532, 533
 dec data type, 241
 decimal data type, 240-243
 Declarative referential integrity (DRI), 314, 315
 DECLARE statement, 421-425
 for a table variable, 424, 425
 Declare (parameter), 464, 465
 Default backup directory, 60, 61
 Default column order, 368, 369
 Default data directory, 58, 59
 Default database for login, 540, 541
 DEFAULT keyword, 220, 221, 340, 341
 and function, 486, 487
 and UPDATE, 224, 225
 Default schema, 56
 for database, 546, 547
 for user, 544, 545
 Default value, 14, 15
 and INSERT, 220, 221
 DELETE statement, 30, 31, 230-233
 in trigger, 494, 495
 multiple rows with subquery, 232, 233
 through view, 410, 411
 with join, 232, 233
 Deleted table, 494, 495
 Deletion anomaly, 315
 Delimited identifier, 336, 337
 Delimiter, 336, 337
 Denormalization, 328, 329
 DENSE_RANK function, 292-295
 DENY statement, 570, 571
 Dependency, 320, 321, 386, 387, 504, 505
 Derived data, 326, 327
 Derived table, 200, 201, 428, 429
 Derived view, 412, 413
 DESC keyword (ORDER BY), 116, 117
 Design a database, 304-329
 Detach a database, 58, 59
 Diagram (database), 64, 65
 Diagram pane (Query Designer), 76, 77
 Dialect (SQL), 18, 19
 Dirty read, 520-523
 Disconnected data architecture, 38-41
 DISTINCT keyword, 90, 91
 and self-joins, 134, 135
 in aggregate function, 160-163
 in SELECT clause, 100, 101
 in view, 404, 405
 with CUBE, 174, 175
 with ROLLUP, 172, 173

Division (integer), 252, 253
 Division operator, 97
 DKNF, 320, 321
 DML, 22, 23
 Document (XML), 588, 589
 Domain, 320, 321
 Domain-key normal form, 320, 321
 double precision data type, 241
 Double precision number, 242, 243
 Double quotes
 delimiter, 336, 337
 in column name, 92, 93
 Downloadable files (installing), 652, 653
 DRI, 314, 315
 Driver, 38, 39
 DROP APPLICATION ROLE statement, 572, 573
 DROP DATABASE statement, 335, 352, 353
 DROP FUNCTION statement, 335, 492, 493
 DROP INDEX statement, 335, 352, 353
 DROP LOGIN statement, 542, 543
 DROP PROC statement, 335, 480, 481
 DROP ROLE statement, 566, 567
 DROP SCHEMA statement, 546, 547
 DROP SEQUENCE, 358, 359
 DROP SERVER ROLE statement, 560, 561
 DROP TABLE statement, 335, 352, 353
 DROP TRIGGER statement, 335, 504, 505
 DROP USER statement, 544, 545
 DROP VIEW statement, 335, 406, 407
 DROP XML SCHEMA COLLECTION statement,
 608, 609
 Duplicate rows (eliminating), 100, 101
 Dynamic SQL, 446, 447

E

Edit table data, 68, 69
 Editions (SQL Server 2019 Express), 648, 649
 Element
 data, 306-309
 XML, 588, 589
 ELEMENTS keyword (FOR XML clause), 610-613
 ELSE clause, 421, 430, 431
 Enable remote connection, 52, 53
 Encoding, 360-361
 Encryption, 584
 ENCRYPTION clause
 in stored procedure, 462, 463
 in scalar-valued function, 486, 487
 in trigger, 494, 495
 END clause, 421
 End tag (XML), 588, 589
 Engine (database), 50, 51
 Enterprise system, 4, 5
 Entity, 304, 305
 Entity Framework (EF), 38, 39
 Entity-relationship (ER) modeling, 304, 305
 Entry level of conformance (SQL-92), 18, 19
 EOMONTH function, 272-275
 Equal operator, 104, 105

ER modeling, 304, 305
 Error
 handling, 438, 439
 number, 438-441
 query, 72, 73
 syntax, 72, 73
 ERROR_MESSAGE function, 438, 439
 ERROR_NUMBER function, 438, 439
 ERROR_SEVERITY function, 438, 439
 ERROR_STATE function, 438, 439
 Escalation (lock), 524, 525
 EVENTDATA function, 502, 503, 592, 593
 Exact numeric data type, 242, 243
 EXCEPT operator, 154, 155
 Exception, 438, 439
 Exclusive (X) lock, 526, 527
 EXEC statement, 36, 37, 421, 446, 447, 460, 461
 EXECUTE AS clause, 462, 463, 480, 481, 486, 487,
 494, 495
 Execution plan, 460, 461
 exist() method (xml data type), 596, 597
 EXISTS operator
 IF statement, 500, 501
 WHERE clause, 198, 199

Explicit
 cross join syntax, 148, 149
 data type conversion, 250, 251
 inner join syntax, 126, 127
 outer join, 140, 141
 transaction, 510-513
 Express Edition (SQL Server), 50, 51
 Expression, 90, 91
 arithmetic, 96, 97
 common table, 208-211
 string, 94, 95
 Extensible Markup Language (XML), 588, 589
 Extension (SQL), 18, 19
 Extent lock, 524, 525

F

FETCH clause, 120, 121
 FETCH statement, 436, 437
 Field, 10, 11
 Fifth normal form, 320, 321, 328, 329
 File system, 16, 17
 File-handling system (compared to client/server
 system), 6, 7
 FILEGROUP clause (CREATE DATABASE), 636, 637
 Files for this book (installing), 652, 653
 FILESTREAM storage, 634-645
 enabling, 634, 635
 pros and cons, 620, 621
 Filter, 86, 87
 Filtered index, 342, 343
 Fine-grain lock, 524, 525
 Fire a trigger, 494, 495
 First normal form, 320-323
 FIRST_VALUE function, 296, 297

- Fixed role
 - database, 564, 565
 - server, 558, 559
- Fixed-length encoding, 361
- Fixed-length string, 244, 243
 - float data type, 240-243
 - Floating-point number, 242, 243
 - FLOOR function, 268, 269
 - FOR ATTACH clause, 338, 339
 - FOR LOGIN clause, 544, 545
 - FOR trigger, 494, 495
 - FOR XML clause, 610-613
 - Foreign key, 12, 13
 - constraint, 314, 315, 346, 347, 350, 351
 - how to identify, 312, 313
 - referential integrity, 314, 315
 - relationship, 380, 381
 - Form (normal), 316, 317, 320, 321, 444, 445
 - Formatting object identifiers, 336, 337
 - Fourth normal form, 320, 321
 - FROM clause, 86, 87, 126-149
 - and subquery, 200, 201
 - Front end, 6, 7
 - FULL JOIN keyword, 140-145
 - Full level of conformance (SQL-92), 18, 19
 - Full outer join, 140-145
 - Full-table index, 342, 343
 - Full-Text Search, 112
 - Fully-qualified object name, 130, 131
 - Function, 98, 99, 484, 485
 - aggregate, 160-163
 - call, 484, 485
 - change, 492, 493
 - column, 160, 161
 - data conversion, 252-259
 - delete, 492, 493
 - invoke, 484, 485
 - scalar-valued, 160, 484-487
 - table-valued, 484, 485, 488-491
 - user-defined, 484-493
 - Functions
 - @@ERROR, 442, 443
 - @@FETCH_STATUS, 436, 437
 - @@IDENTITY, 442, 443
 - @@ROWCOUNT, 442, 443, 512, 513
 - @@SERVERNAME, 442, 443
 - @@TRANCOUNT, 514, 515
 - ABS, 268, 269
 - Aggregate, 160-163
 - Analytic, 296-299
 - ASCII, 258, 259
 - AVG, 160-163
 - CASE, 284, 285
 - CAST, 252, 253, 266, 267, 278, 279
 - CEILING, 268, 269
 - CHAR, 258, 259
 - CHARINDEX, 262-267
 - CHOOSE, 286, 287
 - COALESCE, 288, 289
 - CONCAT, 262-265
 - CONCAT_WS, 262-265
 - CONVERT, 98, 99, 254, 255
 - COUNT, 160-163
 - CUME_DIST, 296-299
 - date/time, 273-391
 - DATEADD, 272-275, 278, 279
 - DATEDIFF, 98, 99, 272, 273, 278, 279
 - DATENAME, 272-277
 - DATEPART, 272-277
 - DAY, 272, 273, 276, 277
 - DB_ID, 432, 433
 - DENSE_RANK, 292-295
 - EOMONTH, 272-275
 - ERROR_MESSAGE, 438, 439
 - ERROR_NUMBER, 438, 439
 - ERROR_SEVERITY, 438, 439
 - ERROR_STATE, 438, 439
 - EVENTDATA, 502, 503, 592, 593
 - FIRST_VALUE, 296, 297
 - FLOOR, 268, 269
 - GETDATE, 98, 99, 272-275
 - GETUTCDATE, 272-275
 - GROUPING, 172, 173, 290, 291
 - HOST_NAME, 442, 443
 - IDENT_CURRENT, 442, 443
 - IIF, 286, 287
 - ISDATE, 272, 273
 - ISNULL, 288, 289
 - ISNUMERIC, 268, 269
 - LAG, 296-299
 - LAST_VALUE, 296, 297
 - LEAD, 296-299
 - LEFT, 98, 99, 262-267
 - LEN, 262-267
 - LOWER, 262-265
 - LTRIM, 262-265
 - MAX, 160-163
 - MIN, 160-163
 - MONTH, 272-277
 - NCHAR, 258, 259
 - NEWID, 638, 639
 - NEXT VALUE FOR, 356, 357
 - NTILE, 293-295
 - numeric, 268, 269
 - OBJECT_ID, 432, 433
 - PathName, 638, 639
 - PATINDEX, 262-265
 - PERCENT_RANK, 296-299
 - PERCENTILE_CONT, 296-299
 - PERCENTILE_DISC, 296-299
 - RAND, 268, 269
 - RANK, 292-295
 - ranking, 292-295
 - REPLACE, 262-265
 - REVERSE, 262, 263
 - RIGHT, 262-267
 - ROUND, 268, 269
 - ROW_NUMBER, 292, 293
 - RTRIM, 262-265
 - SPACE, 262, 263

SQRT, 268, 269
 SQUARE, 268, 269
 STR, 258, 259
 SUBSTRING, 262-265
 SUM, 160-163
 SWITCHOFFSET, 272, 273
 SYSDATETIME, 272-275
 SYSDATETIMEOFFSET, 272-275
 SYSTEM_USER, 442, 443
 SYSUTCDATETIME, 272-275
 TODATETIMEOFFSET, 272, 273
 TRANSLATE, 262-265
 TRIM, 262-265
 TRY_CONVERT, 256, 257
 UNICODE, 258, 259
 UPPER, 262-265
 XML_SCHEMA_NAMESPACE, 608, 609
 YEAR, 272, 273, 276, 277

G

Generate a script, 388-391
 geography data type, 240, 241
 geometry data type, 240, 241
 GETDATE function, 98, 99, 272-275
 GETUTCDATE function, 272-275
 Global
 temporary table, 426, 427
 variable, 422, 442, 443
 temporary procedure, 462, 463
 Globally unique identifier (GUID), 638, 639
 GMT, 246, 272
 GO command, 368, 369, 418, 419
 GOTO statement, 420, 421
 Grant database access, 578, 579
 Grant statement, 548-557
 database permissions, 554, 555
 object permissions, 548-551
 schema permissions, 552, 553
 server permissions, 556, 557
 user-defined database roles, 566, 567
 user-defined server roles, 560, 561
 Granularity, 524, 525
 Greater than operator, 104, 105
 Greater than or equal to operator, 104, 105
 Greenwich Mean Time (GMT), 246, 272
 Group (Windows), 537
 GROUP BY clause, 164-167
 in view, 404, 405
 GROUPING function, 290, 291
 with ROLLUP, 172, 173
 GROUPING SETS operator, 176, 177
 GUID (Globally unique identifier), 638, 639
 Guidelines
 coding, 32, 33
 complex query, 204-207
 object identifier, 336, 337
 procedure name, 462, 463

H

Hardware components (client/server system), 4, 5
 HAVING clause, 164-169
 compared to WHERE clause, 168, 169
 in view, 404, 405
 Help system, *see SQL Server documentation*
 Hibernate, 38, 39
 Hierarchical database model, 16, 17
 hierarchyid data type, 240, 241
 HOST_NAME function, 442, 443

I

IBM, 18, 19
 ID (login), 540-543, 574, 575
 IDENT_CURRENT function, 442, 443
 Identifier (object), 336, 337
 Identity column, 14, 15, 442, 443
 and INSERT, 220, 221
 IDENTITY keyword, 340, 341
 IF clause, 430, 431
 IF EXISTS clause, 432, 433
 IF statement, 440, 441
 IF...ELSE statement, 421, 430, 431
 IIF function, 286, 287
 image data type, 248, 249
 Implicit
 cross join syntax, 148, 149
 data type conversion, 250, 251
 inner join syntax, 138, 139
 transaction, 512, 513
 IN phrase, 108, 109
 and subquery, 188, 189
 INCREMENT BY clause, 356, 357
 Index, 10, 11, 318, 319, 382, 383
 clustered, 382, 383
 create, 342, 343
 database, 10, 11
 delete, 352, 353
 filtered, 342, 343
 full-table, 342, 343
 identifying columns, 318, 319
 nonclustered, 382, 383
 Information schema view, 412
 Inline table-valued function, 488, 489
 Inner join, 28, 29, 126-139
 combined with outer join, 146, 147
 explicit syntax, 126, 127
 implicit syntax, 138, 139
 SQL-92 syntax, 126, 127
 INNER keyword, 126, 127
 Input parameter, 36, 464-467, 484
 INSERT statement, 30, 31, 218-223
 column list, 218, 219
 default value, 220, 221
 identity value, 220, 221
 multiple rows, 218, 219

multiple rows from subquery, 222, 223,
 null value, 220, 221
 through view, 404, 405, 410, 411
 and trigger, 494, 495
 Inserted table, 494, 495
 Insertion anomaly, 315
 Instance, 304, 305
 INSTEAD OF trigger, 404, 405, 494, 495, 498, 499
 int data type, 240-243
 integer data type, 240-243
 Integer division, 252, 253
 Integrity (referential), 314, 315, 496, 497
 IntelliSense feature, 70, 71
 Intent Exclusive (IX) lock, 527, 528
 Intent locks, 527, 528
 Intent Shared (IS) lock, 527, 528
 Interim result set, 136, 137
 Interim table, 136, 137
 Intermediate level of conformance (SQL-92), 18, 19
 INTERSECT operator, 154, 155
 INTO clause
 in INSERT, 218, 219
 in SELECT, 216, 217
 Introduce a subquery, 184, 185
 Invoke a function, 484, 485
 IS NULL clause, 114, 115, 444, 445
 ISDATE function, 272, 273
 ISNULL function, 288, 289
 ISNUMERIC function, 268, 269
 ISO, 19
 Isolation level
 deadlocks, 532, 533
 SERIALIZABLE, 120
 SNAPSHOT, 120
 transaction, 522, 523

J

Java Database Connectivity (JDBC), 38, 39
 JavaScript Object Notation (JSON), 616
 JDBC, 38, 39
 Join, 28, 29, 126-149
 compared to subquery, 186, 187
 compound condition, 132, 133
 condition, 126, 127
 cross, 28, 148, 149
 explicit syntax, 126, 127
 in DELETE statement, 232, 233
 in UPDATE statement, 228, 229
 inner, 28, 29, 126-139
 inner and outer combined, 146, 147
 keyword, 126-149
 more than two tables, 136, 137
 outer, 28, 29, 140-145
 self, 134, 135
 SQL-92 syntax, 126, 127
 JSON (JavaScript Object Notation), 616

K

Key, 382, 383
 composite primary, 10, 11
 foreign, 12, 13
 identify, 312, 313
 lock, 524, 525
 non-primary, 10, 11
 primary, 10, 11
 unique, 10, 11
 Keyword, 86, 87

L

LAG function, 296-299
 LAN, 4, 5
 Large value data types, 248, 249
 LAST_VALUE function, 296, 297
 Latin1 encoding, 360, 361
 LEAD function, 296-299
 LEFT function, 98, 99, 262-267
 LEFT JOIN keyword, 140-145
 Left operator, 140, 141
 Left outer join, 140-145
 LEN function, 262-267
 Less than operator, 104, 105
 Less than or equal to operator, 104, 105
 Levels of conformance (SQL-92 standard), 18, 19
 LIKE operator, 112, 113
 Linked server, 130, 131
 Linking table, 312, 313
 Linux operating system, 20, 21
 Literal
 date, 104, 105
 numeric, 104, 105
 string, 94, 95
 Unicode, 258, 259
 value, 94, 95
 Local
 temporary procedure, 462, 463
 temporary table, 426, 427
 variable, 422, 423
 Local area network (LAN), 4, 5
 Lock, 518, 519
 escalation, 524, 525
 granularity, 524, 525
 manager, 524, 525
 mode compatibility, 528, 529
 modes, 526, 527
 promotion, 526, 527
 Lockable resource, 524, 525
 Locking, 518-523
 Log file
 database, 376, 377
 transaction, 338, 339
 Logical operator, 106, 107
 Login, 54, 55
 creating, 540, 541, 574, 575
 deleting, 542, 543, 574, 575

ID, 536, 537, 574, 575
 modifying, 542, 543, 574, 575
 Loop, 434, 435
 Lost update, 518-523
 LOWER function, 262-265
 LTRIM function, 262-265

M

Management Studio, 50, 51, 54-77, 376-391
 installing, 650, 651
 security, 536, 537, 574-583
 Object Explorer, 56, 57
 Query Designer, 76, 77
 View Designer, 414, 415
 Many-to-many relationship, 12, 13, 312, 313
 Mask, 112, 113
 MAX function, 160-163
 Member information
 database role, 568, 569
 server role, 562, 563
 fixed server role, 558, 559
 MERGE statement, 234, 235
 Method (xml data type), 596-599
 Microsoft SQL Server, *see* SQL Server
 Microsoft Windows PowerShell, 452
 MIN function, 160-163
 Mixed mode, 538, 539
 Model
 data access, 38, 39
 database, 16, 17
 modify() method (xml data type), 596-599
 Modify table data, 68, 69, 378, 379
 Modulo operator, 97
 money data type, 242, 243
 MONTH function, 272-277
 Moving average, 178, 179
 MSDE, 50
 Multi-statement table-valued function, 484, 485, 490, 491
 Multiple rows (and INSERT), 218, 219
 Multiplication operator, 97
 Multivalued dependency, 320, 321
 MUST_CHANGE option, 540, 541
 MySQL database system, 20, 21

N

Name
 column, 92, 93
 correlation, 128, 129, 196, 197
 passing parameters by, 466, 467
 rules, 336, 337
 national char data type, 241
 national char varying data type, 241
 national character data type, 241
 national character varying data type, 241
 National character, 244, 243
 national text data type, 241
 nchar data type, 241, 244, 245
 NCHAR function, 258, 259

Nested

IF...ELSE statements, 430, 431
 sort, 116, 117
 subqueries, 184, 185
 transactions, 514, 515
 views, 400, 401
 Network, 4, 5
 Network database model, 16, 17
 Network configuration, 52, 53
 Network operating system, 6
 New Database dialog box, 376, 377
 NEWID function, 638, 639
 NEXT VALUE FOR function, 356, 357
 NO ACTION keyword, 350, 351
 NOCOUNT system option, 445
 nodes() method (xml data type), 596, 597
 Nonclustered index, 318, 319, 382, 383
 NONCLUSTERED keyword, 342, 343
 Noncorrelated subquery, 196, 197
 Non-primary key, 10, 11
 Nonrepeatable read, 520-523
 Normal form, 316, 317, 320, 321
 Normalization, 316, 317, 320-329
 Normalized data structure, 316, 317
 Not equal operator, 104, 105
 NOT IN phrase (and subquery), 188, 189
 NOT NULL constraint, 340, 341, 346, 347
 NOT OPERATOR, 104-115
 with ISNULL, 114, 115
 ntext data type, 241, 248, 249
 NTILE function, 293-295
 Null, 14, 15, 114, 115
 and aggregate function, 160-163
 and INSERT, 220, 221
 functions, 288-291
 searching, 114, 115
 NULL keyword
 and CREATE TABLE, 340, 341
 and INSERT, 220, 221
 and UPDATE, 224, 225
 Numeric data, 262-271
 common problems, 270, 271
 data type, 240, 241
 functions, 268, 269
 literal, 104, 105
 nvarchar data type, 241, 244, 245, 248, 249

O

Object
 dependencies, 386, 387
 identification number, 432, 433
 identifier, 336, 337
 name, 130, 131, 336, 337
 permissions, 536, 550-551, 570, 571
 Object database, 16, 56, 57, 334, 338, 339
 Object Explorer (Management Studio), 56, 57
 Object relational mapping (ORM) framework, 38, 39
 OBJECT_ID function, 432, 433
 OFFSET clause, 120, 121

ON clause
 MERGE statement, 234, 235
 trigger, 494, 495
 ON DELETE clause, 350, 351
 ON phrase, 126, 127
 ON PRIMARY clause, 338, 339
 ON UPDATE clause, 350, 351
 One-to-many relationship, 12, 13, 312, 313
 One-to-one relationship, 12, 13, 312, 313
 Online help, *see SQL Server documentation*
 Open-source database, 20, 21
 OPENXML statement, 614, 615
 Operating system
 AIX, 21
 Linux, 20, 21
 Unix, 20, 21
 Windows, 20, 21
 z/OS, 21
 Operators
 addition, 97
 AND, 106, 107
 arithmetic, 96, 97
 comparison, 104, 105
 concatenation, 94, 95
 division, 97
 equal, 104, 105
 greater than, 104, 105
 greater than or equal to, 104, 105
 less than, 104, 105
 less than or equal to, 104, 105
 logical, 106, 107
 modulo, 97
 multiplication, 97
 NOT, 106-115
 not equal, 104, 105
 OR, 106, 107
 order of precedence, 96, 97, 106, 107
 string, 94, 95
 subtraction, 97
 UNION, 150, 151, 404, 405
 Optional parameter, 464-467
 OR operator, 106, 107
 Oracle, 18-21
 ORDER BY clause, 86, 87, 102, 103, 116-121
 by string column, 266, 267
 in view, 400-403
 of a ranking function, 292-295
 Order of precedence
 data types, 250, 251
 operators, 96, 97
 ORM framework, 38, 39
 Orphan, 314, 315
 OS/390 operating system 20, 21
 OSQL utility, 452, 453
 Outer join, 28, 29, 140-145
 combined with inner join, 146, 147
 examples, 142, 143
 explicit syntax, 140, 141
 more than two tables, 144, 145
 Output parameter, 36, 464-467
 OVER clause, 178, 179

P

Package (SQL-99 standard), 18, 19
 Page lock, 524, 525
 Parameter, 98, 99, 458, 459, 462-467, 484
 pass a table, 478, 479
 pass a value, 466, 467
 stored procedure, 36
 Parent element (XML), 588, 589
 Parent/child relationship, 16, 17
 Parentheses, 96, 97, 106, 107
 Parsing, 266, 267
 Partially-qualified object name, 130, 131
 PARTITION BY clause
 aggregate function, 178, 179
 analytic function, 296, 297
 ranking function, 292, 293
 Pass a parameter, 466, 467
 Password (strong), 540, 541
 PathName function, 638, 639
 PATINDEX function, 262-265
 Pattern (string), 112, 113
 PERCENT keyword, 102, 103
 PERCENT_RANK function, 296-299
 PERCENTILE_CONT function, 296-299
 PERCENTILE_DISC function, 296-299
 Permission (defined), 536, 537
 Permissions, 548-557
 database, 554, 555, 582, 583
 object, 550, 551
 schema, 552, 553
 server, 556, 557
 Phantom read, 520-523
 Position (passing parameters by), 466, 467
 PostgreSQL database system, 20
 PowerShell (Windows), 452
 Precedence, 96, 97
 Precision, 242, 243
 Precompile, 460, 461
 Predicate, 86, 87
 subquery, 184, 185
 Primary key, 10, 11, 382, 383
 and referential integrity, 314, 315
 composite, 10, 11
 constraint, 340, 341, 346, 347
 how to identify, 312, 313
 PRINT statement, 420, 421
 Procedural programming, 458, 459
 Promote a lock, 526, 527
 Properties of a column, 66, 67
 Pseudocode, 206, 207
 Public database role, 564, 565
 public server role, 558

Q

Qualified column name, 126, 127
 Qualified object name, 130, 131
 Qualifier (scope), 552, 553

Query, 27, 30, 31
 action, 30, 31
 aggregate, 160-163
 complex, 204, 205
 enter and execute, 70, 71
 error, 72, 73
 open and save, 74, 75
 recursive, 210, 211
 SQL, 6, 7
 summary, 160-177
Query Designer (Management Studio), 76, 77
Query Editor (Management Studio), 70-73
query() method (xml data type), 596, 597
Query results, 7
Quotes
 delimiter, 336, 337
 for string literal, 94, 95
 in column name, 92, 93
 within string literal, 94, 95

R

RAISERROR statement, 470-472
RAND function, 268, 269
RANGE clause, 296, 297
RANK function, 292-295
Ranking functions, 292-295
RAW keyword (FOR XML clause), 610, 611
RDBMS, 18
READ COMMITTED keyword, 522, 523
READ UNCOMMITTED keyword, 522, 523
READONLY keyword, 478, 479
Read-only view, 404, 405
real data type, 240-243
Real numeric value, 270, 271
Recommendations
 coding, 32, 33
 procedure name, 462, 463
RECOMPILE keyword, 462, 463
Record, 10, 11
Recursion, 461
Recursive call, 461
Recursive CTE, 210, 211
Recursive member, 210, 211
Recursive query, 210, 211
Redundant data, 316, 317
Reference constraint, 350, 351
Reference manual, *see SQL Server documentation*
REFERENCES clause, 25, 346, 347, 350, 351
Referential integrity, 314, 315
Regular identifier, 336, 337, 496, 497
Related tables, 12, 13
Relational database management system (RDBMS), 18
Relational database, 10, 11
 advantages, 10, 16, 17
 compared to other data models, 16, 17
Relational Software, Inc., 18, 19
Relationship
 ad hoc, 126, 127
 between tables, 12, 13, 312, 313

Remote connection, (enable), 52, 53
REPEATABLE READ keyword, 522, 523
Repetitive processing, 434, 435
REPLACE function, 262-265
Replication, 380, 381
Required parameter, 464-467
Resources
 lockable, 524, 525
RESTART clause, 358, 359
Restore database, 60, 61, 654, 655
Result set, 26, 27
 interim, 136, 137
Result table, 26, 27
Results (query), 7
RETURN statement, 420, 421
 in function, 484, 486-491
 in procedure, 468, 469
Return value, 468, 469
REVERSE function, 262, 263
Revoke database access, 578, 579
Revoke permissions, 548-557
REVOKE statement
 database permissions, 554, 555
 object permissions, 548-551
 schema permissions, 552, 553
 server permissions, 556, 557
RIGHT function, 262-267
RIGHT JOIN keyword, 140-145
Right operator, 140, 141
Right outer join, 140, 141
Role, 558-573
 application, 572, 573
 database, 578, 579
 defined, 536, 537
 fixed database, 564, 565
 fixed server, 558, 559
 server, 576, 577
 user-defined database, 566, 567
 user-defined server, 560, 561
Role information
 database, 568, 569
 server, 562, 563
Roll back a transaction, 510, 511
ROLLBACK TRAN statement, 512-517
 in trigger, 496, 497
ROLLUP operator, 172, 173
Root element (XML), 588, 589
ROOT keyword (FOR XML clause), 610-613
ROUND function, 268, 269
Row, 10, 11
Row count, 442, 443
Row lock, 524, 525
Row versioning, 522, 523
ROW_NUMBER function, 292, 293
ROWCOUNT system option, 444, 445
ROWGUIDCOL property, 638, 639
ROWS clause, 296, 297
rowversion data type, 240, 241
RTRIM function, 262-265

S

Save a change script, 390, 391
 Save point, 512, 513, 516, 517
 SAVE TRAN statement, 512, 513
 Scalar aggregate, 164, 165
 Scalar function, 160
 Scalar variable, 422, 423
 Scalar-valued function (user-defined), 484-487
 Scale, 242, 243
 Scan (table), 318
 Schema, 400, 401
 default, 56
 lock, 526, 527
 name, 130, 131
 permission, 536, 552, 553
 view, 412, 413
 working with, 546, 547
 Schema (XML), 590, 591
 Schema Modification (Sch-M) lock, 526, 527
 Schema permissions (DENY), 570, 571
 Schema Stability (Sch-S) lock, 526, 527
 SCHEMABINDING keyword,
 UDF, 486, 487
 view, 400-403, 406, 407
 Scientific notation, 242
 Scope
 table objects, 428, 429
 variable, 422
 Scope qualifier, 552, 553
 Script, 368-371, 418-447, 458, 459
 change, 390, 391
 generating, 388-391
 Script to create the AP database, 368-371
 Search
 by date value, 280, 281
 by time value, 282, 283
 for null, 114, 115
 for real numeric value, 270, 271
 Search condition, 86, 87
 compound, 106, 107, 170, 171
 subquery, 188-199
 Second normal form, 320, 321, 324, 325
 Securables, 536, 537
 Security, 536-583
 Transact-SQL compared to Management Studio, 536, 537
 using view, 398, 399
 securityadmin role, 558, 559
 SELECT clause, 86-101
 SELECT statement, 26, 27, 86-119
 and subquery, 202, 203
 in view, 400, 401
 INTO, 216, 217, 448, 449
 variable assignment, 422, 423
 Self-join, 134, 135
 SEQUEL, 18, 19
 Sequence, 356-359
 SERIALIZABLE keyword, 522, 523

Server, 4, 5
 application, 8, 9
 connect, 54, 55
 database, 50, 51
 linked, 130, 131
 login, 54, 55
 permissions, 556, 557
 software, 6, 7
 web, 8, 9
 Server authentication, 54, 55
 Server name, 130, 131
 Server permission, 536
 Server permissions (DENY), 570, 571
 Server role, 576, 577
 fixed, 558, 559
 information, 562, 563
 user-defined, 560, 561
 Service (web), 8, 9
 Service Manager, 50, 51
 Services (SQL Server), 52, 53
 Session setting, 444, 445
 Set (result), 26, 27
 SET clause (UPDATE), 224, 225
 SET statement, 421-423
 ANSI_NULLS, 444, 445
 ANSI_PADDING, 445
 DATEFORMAT, 444, 445
 NOCOUNT, 445
 ROWCOUNT, 444, 445
 TRANSACTION ISOLATION LEVEL, 522, 523
 Shared (S) lock, 526, 527
 Shared locks, 526, 527
 Shared with Intent Exclusive (SIX) lock, 527, 528
 Significant digit, 242, 243
 Simple table-valued function, 484, 485, 488, 489
 Single precision number, 242, 243
 Single quotes
 for string literal, 94, 95
 in column name, 92, 93
 within string literal, 94, 95
 Single-line comment, 32, 33
 Sixth normal form, 320, 321
 smalldatetime data type, 246, 247
 smallint data type, 242, 243
 smallmoney data type, 242, 243
 SNAPSHOT isolation level, 120
 SNAPSHOT keyword, 522, 523
 Snippets, 344, 345
 surround-with, 440, 441
 Software components (client/server system), 6, 7
 SOME keyword, 194, 195
 Sort, *see ORDER BY clause*
 sp_AddLinked Server, 130, 131
 sp_AddRoleMember, 565
 sp_AddSrvRoleMember, 559
 sp_Columns, 483
 sp_DropRoleMember, 565
 sp_DropServer, 131, 132
 sp_DropSrvRoleMember, 559
 sp_Help, 483

sp_HelpDb, 483
sp_HelpRole, 568, 569
sp_HelpRoleMember, 568, 569
sp_HelpSrvRole, 562, 563
sp_HelpSrvRoleMember, 562, 563
sp_HelpText, 483
sp_SetAppRole, 572, 573
sp_UnsetAppRole, 572, 573
sp_Who, 483
sp_Xml_PrepDocument, 614, 615
sp_Xml_RemoveDocument, 614, 615
SPACE function, 262, 263
SPARSE attribute, 340, 341
Specification (column), 90, 91
Sproc, 460-481
 name, 462, 463
SQL, 6, 7
 ANSI-standards, 18, 19
 batch, 418, 419
 coding guidelines, 32, 33
 dialect, 18, 19
 dynamic, 446, 447
 extensions to, 18, 19
 script, 418-447, 458, 459
 standards, 18, 19
 variant, 18, 19
SQL Anywhere database system, 20
SQL query, 6, 7
SQL Server, 18, 19
 authentication, 54, 55
 compared to Oracle, MySQL, and DB2, 20, 21
 Configuration Manager, 50-53
 data types, 14, 15, 240-249
 default backup directory, 60, 61
 default data directory, 58, 59
 documentation, 78, 79
 editions, 648, 649
 Express Edition, 50, 51
 login authentication, 54, 55
 login ID, 540, 541
 Management Studio, 50, 51, 54-77, 376-391
 when first released, 19
SQL Server 2019 Express (installing), 650, 651
SQL Server authentication, 538, 539
SQL Server documentation, 50, 51, 78, 79
SQL Server Management Studio, *see Management Studio*
SQL Server Services, 52, 53
SQL statements, 22, 23
SQL/DS (SQL/Data System), 18, 19
SQL-92 syntax, 126, 127
SQLCMD utility, 452, 453, 458, 459
SQRT function, 268, 269
SQUARE function, 268, 269
Standard table, 428, 429
Start database engine, 52, 53
Start tag (XML), 588, 589
START WITH clause, 356, 357
State argument, 470, 471
Statements
 ALTER FUNCTION, 335, 492, 493
 ALTER LOGIN, 542, 543
 ALTER PROC, 335, 480, 481
 ALTER ROLE, 564-597
 ALTER SCHEMA, 546, 547
 ALTER SEQUENCE, 358, 359
 ALTER SERVER ROLE, 558-561
 ALTER TABLE, 25, 335, 352-355
 ALTER TRIGGER, 335, 504, 505
 ALTER USER, 544, 545
 ALTER VIEW, 335, 406, 407
 BEGIN TRAN, 512, 513
 BULK INSERT, 527, 528
 COMMIT TRAN, 512-517
 CONTINUE, 421, 434, 435
 CREATE APPLICATION ROLE, 572, 573
 CREATE INDEX, 25, 335, 342, 343
 CREATE LOGIN, 540, 541
 CREATE PROC, 466, 467
 CREATE PROCEDURE, 335
 CREATE ROLE, 566, 567
 CREATE SCHEMA, 546, 547
 CREATE SEQUENCE, 356, 357
 CREATE SERVER ROLE, 560, 561
 CREATE TYPE, 478, 479
 CREATE USER, 544, 545
 CREATE VIEW, 34, 35, 335, 396, 397, 400-403
 CREATE XML SCHEMA COLLECTION, 604, 605
 DECLARE, 421-425
 DELETE, 30, 31, 230-233
 DENY, 570, 571
 DROP APPLICATION ROLE, 572, 573
 DROP DATABASE, 335, 352, 353
 DROP FUNCTION, 335, 492, 493
 DROP INDEX, 335, 352, 353
 DROP LOGIN, 542, 543
 DROP PROC, 335, 480, 481
 DROP ROLE, 566, 567
 DROP SCHEMA, 546, 547
 DROP SEQUENCE, 358, 359
 DROP SERVER ROLE, 560, 561
 DROP TABLE, 335, 352, 353
 DROP TRIGGER, 335, 504, 505
 DROP USER, 544, 545
 DROP VIEW, 335, 406, 407
 DROP XML SCHEMA COLLECTION, 608, 609
 EXEC, 36, 37, 421, 446, 447, 460, 461
 FETCH, 436, 437
 GOTO, 420, 421
 GRANT, 548-557
 IF..ELSE, 421, 430, 431
 INSERT, 30, 31, 218-223, 404, 405, 410, 411, 494, 495
 MERGE, 234, 235
 OPENXML, 614, 615
 PRINT, 420, 421
 RAISERROR, 470-472
 RETURN, 420, 421, 468, 469, 484, 485, 488-491
 REVOKE, 548-557
 ROLLBACK TRAN, 512-517
 SAVE TRAN, 512, 513
 SELECT, 26, 27, 86-119, 202, 203, 422, 423

SELECT INTO, 216, 217, 448, 449
 SET, 421-423, 444, 445
 THROW, 470-472
 TRY...CATCH, 421, 438, 439, 470, 471
 UPDATE, 30, 31, 224-229, 404, 405, 408, 409, 494, 495
 Upsert, 234, 235
 USE, 368, 369, 420, 421, 432, 433
 WAITFOR DELAY, 530, 531
 WHILE, 421, 434, 435, 440, 441, 450, 451
 Stop database engine, 52, 53
 Stored procedure, 458-481
 change, 480, 481
 defined, 36, 37
 delete, 480, 481
 name, 462, 463
 system, 482, 483
 STR function, 258, 259
 String
 common problems, 266, 267
 constant, 94, 95
 data types, 240, 241, 244, 245
 expression, 94, 95
 functions, 262-267
 literal, 94, 95, 104, 105
 parsing, 266, 267
 pattern, 112, 113
 Strong password, 540, 541
 Structure (data), 304, 305
 Structured English Query Language (SEQUEL), 18, 19
 Structured Query Language, *see SQL*
 Style code (data conversion), 254, 255
 Subdivide data element, 308, 309
 Subquery, 108, 109, 184-211
 and ALL keyword, 192, 193
 and ANY keyword, 194, 195
 and comparison operator, 190, 191
 and EXISTS operator, 198, 199
 and IN phrase, 188, 189
 and NOT IN phrase, 188, 189
 and SOME keyword, 194, 195
 compared to join, 186, 187
 correlated, 196, 197, 202, 203
 in DELETE statement, 232, 233
 in FROM clause, 200, 201
 in INSERT statement, 222, 223
 in SELECT clause, 202, 203
 in UPDATE statement, 226, 227
 in WHERE clause, 188-199
 nested, 184, 185
 noncorrelated, 196, 197
 procedure for coding, 204-207
 search condition, 188-199
 Subquery predicate, 184, 185
 Subquery search condition, 184, 185
 Substitute name, 92, 93
 SUBSTRING function, 262-265
 Subtraction operator, 97
 SUM function, 160-163
 Summary query, 160-177

Supplementary characters, 360, 361
 Surround-with snippets, 440, 441
 SWITCHOFFSET function, 272, 273
 Symbols (wildcard), 112, 113
 Syntax conventions, 86, 87
 Syntax error (query), 72, 73
 sysadmin role, 558, 559
 SYSDATETIME function, 272-275
 SYSDATETIMEOFFSET function, 272-275
 System catalog, 412, 413
 System function, 422, 423, 442, 443
 System stored procedure, 482, 483
 System/R, 18, 19
 SYSTEM_USER function, 442, 443
 SYSUTCDATETIME function, 272-275

T

T-SQL, *see Transact-SQL*
 Table, 10, 11
 alias, 128, 129
 associate, 312, 313
 attribute, 340, 341
 base, 26, 27
 change, 352-355
 column properties, 378, 379
 connecting, 312, 313
 create, 340, 341, 378, 379
 data type, 424, 425, 488-491
 database, 10, 11
 delete, 352, 353, 378, 379
 Deleted, 494, 495
 dependency, 386, 387
 derived, 200, 201, 428, 429
 foreign key, 380, 381
 Inserted, 494, 495
 interim, 136, 137
 linking, 312, 313
 lock, 524, 525
 modify, 378, 379
 name, 130, 131
 objects, 428, 429
 primary key, 378, 379
 relationships between, 12, 13
 scan, 318
 standard, 428, 429
 table-level constraint, 340, 341
 temporary, 426-429
 test, 216, 217
 user-defined function, 484, 485
 variable, 424, 425, 428, 429
 Table data
 modify, 68, 69
 view, 68, 69
 Table Designer, 378, 379
 Table-level constraint, 346, 347
 Table-type parameter, 478, 479
 Table-valued function
 multi-statement, 490, 491
 user-defined, 488-491

TDE (Transparent Data Encryption), 584
Temporal data type, *see Date/time data type*
Temporary stored procedure, 462, 463
Temporary table, 426-429
Test for database object, 432, 433
Test table, 216, 217
text data type, 248, 249
Theta syntax, 138, 139
Thin client, 8
Third normal form, 320, 321, 326, 327
THROW statement, 470-472
time data type, 246, 247
Time search, 282, 283
Times (parsing), 276, 277
timestamp data type, 240, 241
tinyint data type, 242, 243
TODATETIMEOFFSET function, 272, 273
Tools
 client, 50, 51
 SQL Server, 50, 51
TOP clause, 90, 91, 102, 103, 444, 445
 in view, 400-405
Transact-SQL, 19, 22, 23, 420, 421
 and security, 536, 537
 programming, 458, 459
 table objects, 428, 429
Transaction, 496, 497, 510-517
 nested, 514, 515
 when to use, 511
Transaction isolation level, 522, 523
 deadlocks, 532, 533
Transaction log file, 338, 339
Transitional level of conformance (SQL-92), 18, 19
Transitive dependency, 320, 321
TRANSLATE function, 262-265
Transparent Data Encryption (TDE), 584
Trigger, 458, 459, 494-505
 AFTER, 494-497
 and referential integrity, 314, 315, 496, 497
 change, 504, 505
 compared to constraint, 500, 501
 data consistency, 500, 501
 defined, 36, 37
 delete, 504, 505
 FOR, 494, 495
 INSTEAD OF, 404, 405, 494, 495, 498, 499
 with DDL statements, 502, 503
TRIM function, 262-265
TRY...CATCH statement, 421, 438, 439, 470, 471
TRY_CONVERT function, 256, 257
Typed XML, 604-609

U

UCS-2 encoding, 360, 361
UDF (user-defined function), 36, 37, 458, 459, 484-493
Unicode
 character, 244, 245
 character set, 360, 361
 literal, 258, 259

UNICODE function, 258, 259
UNION, 150-153
 keyword, 150, 151
 in view, 404, 405
 with same table, 152, 153
UNIQUE constraint, 340, 341, 346, 347
Unique key, 10, 11, 382, 383
Universal Time Coordinate (UTC), 246, 272
Unix operating system, 20, 21
Unnormalized data structure, 316, 317
Untyped XML, 592-603
Updatable view, 404, 405, 408-411, 498, 499
Update (lost), 518-523
Update anomaly, 315
UPDATE clause, 224, 225
Update (U) lock, 526, 527
UPDATE statement, 30, 31, 224-229
 and trigger, 494, 495
 default value, 224, 225
 multiple rows with subquery, 226, 227
 null value, 224, 225
 through view, 404, 405, 408, 409
 with join, 228, 229
UPPER function, 262-265
Upsert statement, 234, 235
USE statement, 368, 369, 420, 421, 432, 433
User database, 544, 545
User permissions, 580, 581
User-defined function (UDF), 36, 37, 458, 459, 484-493
User-defined role
 database, 566, 567
 server, 560, 561
User-defined table type, 478, 479
USING clause (MERGE statement), 234, 235
UTC, 245, 272
UTF-16 encoding, 360, 361
UTF-8 encoding, 360, 361
Utility
 OSQL, 452, 453
 SQLCMD, 452, 453

V

Validate data, 470-477
Validation (XML), 606, 607
Value
 literal, 94, 95
 null, 114, 115
 return, 468, 469
value() method (xml data type), 596-603
VALUES clause, 30, 31, 220, 221
 and INSERT, 218, 219
varbinary data type, 241, 248, 249
varbinary(max) data type, 620-633
varchar data type, 241, 244, 245, 248, 249
Variable, 422, 423
 global, 422, 423, 442, 443
 local, 422, 423
 scalar, 422, 423
 table, 424, 425, 428, 429

Variable-length encoding, 361
 Variable-length string, 244, 243
 Variant (SQL), 18, 19
 Vector aggregate, 164, 165
 Versioning (row), 522, 523
 Victim (deadlock), 530, 531
 View, 396-415, 428, 429
 benefits, 398, 399
 catalog, 412, 413
 change, 406, 407
 defined, 35
 delete, 406, 407
 information schema, 412, 413,
 nested, 400, 401
 read-only, 404, 405
 restrictions on SELECT, 400-403
 updateable, 404, 405, 408-411, 498, 499
 View Designer (Management Studio), 414, 415
 View table data, 68, 69
 Viewed table, 34, 35, 397
 Violate referential integrity, 314, 315
 Virtual table, 35
 Visual Basic code (with ADO.NET), 42, 43
 Visual Studio 2019 Community (installing), 656, 657

W

WAITFOR DELAY statement, 530, 531
 WAN, 4, 5
 Web application, 8, 9
 Web browser, 8, 9
 Web services, 8, 9
 Web-based systems, 8, 9
 WHEN clause (MERGE statement), 234, 235
 WHERE clause, 86-88, 104-115
 and DELETE, 230, 231
 and subquery, 188-199
 and UPDATE, 224, 225
 compared to HAVING clause, 168, 169
 WHILE statement, 421, 434, 435, 440, 441, 450, 451
 Wide-area network (WAN), 4, 5
 Wildcard symbols (LIKE phrase), 112, 113
 Windows
 account (login ID), 540, 541
 authentication, 538, 539
 login authentication, 54, 55
 operating system, 20, 21
 PowerShell, 452
 WITH CHECK keyword (ALTER TABLE), 354, 355
 WITH CHECK OPTION clause, 406-409
 in view, 400, 401
 WITH CUBE phrase, 174, 175
 WITH ENCRYPTION clause, 482
 in trigger, 494, 495
 in view, 400, 401, 406, 407
 WITH GRANT OPTION clause (GRANT), 548, 549
 WITH keyword, 208, 209
 WITH NOCHECK keyword (ALTER TABLE), 354, 355

WITH ROLLUP phrase, 172, 173
 WITH SCHEMABINDING clause (view), 400-403,
 406, 407
 WITH TIES keyword, 102, 103

X

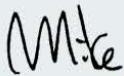
XML, 588-615
 attribute, 588, 589
 create schema, 594, 595
 defined, 588, 589
 document, 588, 589
 element, 588, 589
 tags, 588, 589
 typed, 604-609
 untyped, 592-603
 validate against schema, 606, 607
 view in editor, 594, 595
 with CREATE TABLE, 592, 593
 with DECLARE, 592, 593
 with INSERT, 592, 593
 with SELECT, 592, 593, 610-613
 XML Data Manipulation Language (DML), 596-599
 xml data type, 592-603
 methods, 596-599
 parsing, 600-603
 XML DML, 596-599
 XML Editor, 594, 595
 XML schema, 590, 591, 604-609
 XML Schema Definition (XSD), 590, 591
 XML_NAMESPACE function, 608, 609
 XPath, 614, 615
 XQuery, 596-599
 XSD, 590, 591

YZ

YEAR function, 272, 273, 276, 277
 z/OS operating system 20, 21

100% Guarantee

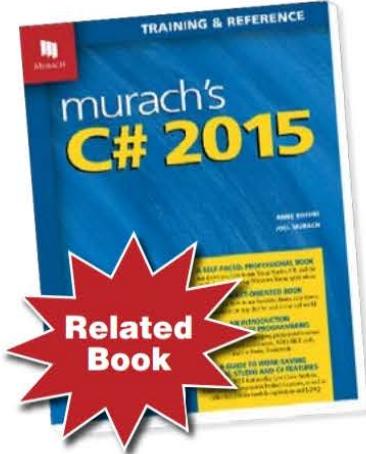
When you order directly from us, you must be satisfied. Try our books for 30 days or our eBooks for 14 days. They must work better than any other programming training you've ever used, or you can return them for a prompt refund. No questions asked!



Mike Murach, Publisher



Ben Murach, President



Become a .NET programmer

This book gives you the core language, .NET, and Visual Studio skills you need to create any C# application. You'll soon see how knowing SQL lets you code database applications more easily.

Books for .NET developers

Murach's C# 2015	\$57.50
Murach's ASP.NET 4.6 Web Programming with C# 2015	59.50
Murach's ASP.NET Core MVC	59.50

Books for database developers

Murach's SQL Server 2019 for Developers	\$59.50
Murach's MySQL (3 rd Ed.)	57.50
Murach's Oracle SQL and PL/SQL for Developers (2 nd Ed.)	54.50

Books for Python, C++, and Java developers

Murach's Python Programming	\$57.50
Murach's C++ Programming	59.50
Murach's Java Programming (5 th Ed.)	59.50
Murach's Java Servlets and JSP (3 rd Ed.)	57.50

Books for web developers

Murach's HTML5 and CSS3 (4 th Ed.)	\$59.50
Murach's JavaScript and jQuery (3 rd Ed.)	57.50
Murach's PHP and MySQL (3 rd Ed.)	57.50

*Prices and availability are subject to change. Please visit our website or call for current information.

We want to hear from you

Do you have any comments, questions, or compliments to pass on to us? It would be great to hear from you! Please share your feedback in whatever way works best.



www.murach.com



1-800-221-5528

(Weekdays, 8 am to 4 pm Pacific Time)



murachbooks@murach.com



twitter.com/MurachBooks



facebook.com/murachbooks



linkedin.com/company/mike-murach-&-associates



instagram.com/murachbooks

The software for this book

- SQL Server 2019 Express (a free download)
- SQL Server Management Studio (a free download)
- Visual Studio Community (a free download)

SQL Server 2019 only runs on Windows 10 and later. As a result, if you're using Windows 8, you'll need to upgrade your operating system before you can install SQL Server 2019 Express.

For information about downloading and installing these products, please see appendix A.

The source code for this book

- Scripts that create the databases for this book
- Scripts for the SQL statements presented throughout this book
- Solutions to the exercises that are at the end of each chapter
- C# and Visual Basic projects for the application presented in chapter 19

How to download and install the source code

1. Go to www.murach.com.
2. Navigate to the page for *Murach's SQL Server 2019 for Developers*.
3. Follow the instructions to download the exe file.
4. Double-click on the exe file to run it.

For details, please see appendix A.

How to create the databases

1. Start the SQL Server Management Studio and connect to the database server.
2. Use the Management Studio to run the three scripts in this directory:

`C:\Murach\SQL Server 2019\Datasets`

For details, please see appendix A.

What makes Murach books the best

During the last 46 years, many customers have asked me how it is that a small publisher in Fresno can make the best programming books. The short answer is that no other publisher works the way we do.

Instead of using freelance writers who get no training, we use a small staff of programmers who are trained in our proven writing and teaching methods. Instead of publishing 40+ books each year, we focus on just a few. Instead of showing pieces of code, we provide complete applications that have been checked and re-checked for accuracy. And instead of rushing books to market, we refuse to let schedule or budget interfere with quality. As I see it, that's the only way to make sure that every book we publish is the best one on its subject.

That's why people often tell me that our books are the ones that they look for *first* whenever they need to learn a new subject. Why not try this book and see for yourself!



Mike Murach
Publisher

SQL Server 2019 contents

Get started fast

- The concepts and terms you need for working with relational databases and SQL
- How to use the Management Studio to work with a SQL Server database

Master the SQL that you'll use every day

- How to write SQL statements that retrieve and update the data in a database
- How to work with inner and outer joins, summary queries, and subqueries...it's all here
- How to use data types and built-in functions to handle everyday challenges like manipulating character data, rounding numbers, and working with date/time values

Learn how to design and create a database

- How to design a database...the first step toward becoming a database administrator
- How to create a database and its tables using SQL statements or the Management Studio...valuable skills, whether you're on the DBA track or not
- How to use a sequence to generate a complex series of integers that you can assign to a column
- How to use collations and UTF-8 encoding to provide for multilingual database applications

Master advanced SQL features

- How to work with views, scripts, stored procedures, functions, table-valued parameters, triggers, cursors, transactions, locking, security, and XML...everything you need to complete your mastery of SQL
- How to use the FILESTREAM storage feature to work with large amounts of binary data, such as image, sound, and video files

What developers have said about previous editions

"Looking to get started with SQL Server? This is a fantastic place to start. Looking for a solid reference as you polish your SQL? Grab a copy."

Stephen Wynkoop, SQL Server Worldwide Users Group

"If you're new to using SQL Server in your applications, this book will save you a lot of time. It teaches you about SQL, database design, a lot about admin (which many developers have to do, not everywhere has dedicated DBAs), and the advanced SQL is excellent. Highly recommended."

David Bolton, Guide for About.com C/C++/C#

"Although I have used SQL Server on a daily basis for over 15 years, I was amazed at the number of new things that I learned. I used a couple of the ideas to create a noticeable improvement in response time for one of my client/server projects."

Brian Mishler, Orlando .NET User Group

"This is the best book on SQL Server I have seen.... I love the format...."

Andrew Katz, Programmer & Instructor, New York

"Murach's SQL Server surprised me. It is extremely easy to read, and I was impressed by its coverage of SQL fundamentals.... This is an excellent first book on SQL for those new to the art, and can provide a complete reference for people who have years of experience."

Glenn Gordon, Mgr. of New Sys. Development, New York

"This SQL Server book really saves me time and effort."

Jian Huang, Lower Alabama .NET Users Group



Printed On Recycled Paper

ISBN: 978-1-943872-57-2



55950

\$59.50 USA

Shelving: Programming/Database/SQL

9 781943 872572

