Sybase SQL Server™
Transact-SQL® User's Guide

Sybase SQL Server Release 11.0.x
Document ID: 32300-01-1100-02
Last Revised: December 15, 1995

Principal author: Server Publications Group

Document ID: 32300-01-1100

This publication pertains to Sybase SQL Server Release 11.0.x of the Sybase database management software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

Document Orders

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor.

Upgrades are provided only at regularly scheduled software release dates.

Sybase Trademarks

APT-FORMS, Data Workbench, DBA Companion, Deft, GainExposure, Gain *Momentum*, Navigation Server, PowerBuilder, Powersoft, Replication Server, SA Companion, SQL Advantage, SQL Debug, SQL Monitor, SQL SMART, SQL Solutions, SQR, SYBASE, the Sybase logo, Transact-SQL, and VQL are registered trademarks of Sybase, Inc. Adaptable Windowing Environment, ADA Workbench, AnswerBase, Application Manager, APT-Build, APT-Edit, APT-Execute, APT-Library, APT-Translator, APT Workbench, Backup Server, Bit-Wise, Client-Library, Client / Server Architecture for the Online Enterprise, Client / Server for the Real World, Client Services, Configurator, Connection Manager, Database Analyzer, DBA Companion Application Manager, DBA Companion Resource Manager, DB-Library, Deft Analyst, Deft Designer, Deft Educational, Deft Professional, Deft Trial, Developers Workbench, DirectCONNECT, Easy SQR, Embedded SQL, EMS, Enterprise Builder, Enterprise Client / Server, Enterprise CONNECT, Enterprise Manager, Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, EWA, Gain Interplay, Gateway Manager, InfoMaker, Interactive Quality Accelerator, Intermedia Server, IQ Accelerator, Maintenance Express, MAP, MDI, MDI Access Server, MDI Database Gateway, MethodSet, Movedb, Navigation Server Manager, Net-Gateway, Net-Library, New Media Studio, OmniCONNECT, OmniSQL Access Module, OmniSQL Gateway, OmniSQL Server, OmniSQL Toolkit, Open Client, Open Client / Server, Open Client / Server Interfaces, Open Gateway, Open Server, Open Solutions, PC APT-Execute,

Restricted Rights

Table of Contents

Preface

1. Introduction

Part 1: Basic Concepts

2. Queries: Selecting Data from a Table

3. Summarizing, Grouping, and Sorting Query Results

4. Joins: Retrieving Data from Several Tables

5. Subqueries: Using Queries Within Other Queries

6. Using and Creating Datatypes

7. Creating Databases and Tables

8. Adding, Changing, and Deleting Data

9. Views: Limiting Access to Data

Part 2:Advanced Topics

10. Using the Built-In Functions in Queries

11. Creating Indexes on Tables

12. Defining Defaults and Rules for Data

13. Using Batches and Control-of-Flow Language

14. Using Stored Procedures

15. Triggers: Enforcing Referential Integrity

16. Cursors: Accessing Data Row by Row

List of Figures

List of Tables

Preface

This manual, the *Transact-SQL User's Guide,* documents Transact-SQL®, an enhanced version of the SQL relational database language. The *Transact-SQL User's Guide* is intended for both beginners and those who have experience with other implementations of SQL.

Audience

Users of the Sybase SQL Server™ database management systems who are unfamiliar with SQL can treat this guide as a textbook and start at the beginning. Novice SQL users should concentrate on the first part of this book. The second part describes topics that are more advanced than those introduced in the first part.

Readers acquainted with other versions of SQL will find this manual useful both as a review and as a guide to Transact-SQL enhancements. SQL experts should study the capabilities and features that Transact-SQL has added to standard SQL, especially the material on stored procedures.

How to Use This Book

This book is a complete guide to Transact-SQL. It contains an introductory chapter which gives an overview of SQL. The remaining chapters are divided into two main parts: Basic Topics and Advanced Topics.

Chapter 1, "Introduction," describes the naming conventions used by SQL and the enhancements (also known as extensions) added by Transact-SQL. It also includes a description of how to get started with Transact-SQL using the **isql** utility. All users should read this chapter, because it prepares you for the other chapters.

"Part 1: Basic Topics" includes Chapters 2–9. These chapters introduce you to the basic functionality of SQL. Users new to SQL should become familiar with the concepts described in these chapters before moving on to Part 2. Experienced users of SQL may want to skim through these chapters to learn about the several Transact-SQL extensions introduced there, and to review the material.

"Part 2: Advanced Topics" includes Chapters 10–17. These chapters describe Transact-SQL in more detail. Most of the Transact-SQL extensions are described here. Users familiar with SQL, but not Transact-SQL, should concentrate on these chapters.

The examples in this guide—of which there are many—are based on the *pubs2* sample database. For best use of the *Transact-SQL User's Guide*, new users should work through the examples. Ask your System Administrator how to get a clean copy of *pubs2*. For a complete description of the *pubs2* database, see the *SQL Server Reference Supplement*.

You can use Transact-SQL with SQL Server's standalone program **isql**. The **isql** program is a utility program called directly from the operating system.

Related Documents

The SQL Server relational database management system documentation is designed to satisfy both the inexperienced user's preference for simplicity and the experienced user's desire for convenience and comprehensiveness. The user's guide and the reference manuals address the various needs of end users, application developers, programmers, and database administrators.

Other manuals you may find useful are:

- *What's New in Sybase SQL Server Release 11.0?*, which describes the new features in release 11.0.
- *SQL Server System Administration Guide*, which provides in-depth information about administering servers and databases. The manual includes instructions and guidelines for managing physical resources and user and system databases, and specifying character conversion, international language, and sort order settings.
- *SQL Server Reference Manual*, which contains detailed information about the commands and system procedures discussed in this manual.
- *SQL Server Reference Supplement*, which contains a list of the Transact-SQL reserved words, definitions of system tables, a description of the *pubs2* sample database, a list of SQL Server error messages, and other reference information common to all the manuals.

- *SQL Server Performance and Tuning Guide*, which gives detailed information on tuning SQL Server and your queries for maximum performance.
- SQL Server utility programs manual, which documents the Sybase utility programs such as **isql** and **bcp**, which are executed at the operating system level.
- *SQL Server Security Features User's Guide*, which is addressed to the general user and explains how to use the security features of SQL Server.
- *SQL Server Security Administration Guide*, which is addressed to the System Administrators responsible for maintaining a secure operating environment for SQL Server. The manual explains how to use the security features provided by SQL Server to control user access to data.
- The SQL Server installation and configuration guide, which describes the installation procedures for SQL Server and documents operating system-specific administration tasks.
- *Master Index for SQL Server Publications*, which combines the indexes of the *SQL Server Reference Manual*, *Transact-SQL User's Guide*, *System Administration Guide*, and *Performance and Tuning Guide*. Use it to locate various topics in different contexts throughout the documentation.

Conventions Used in This Manual

Formatting SQL Statements

SQL is a free-form language. There are no rules about the number of words you can put on a line, or where you must break a line. However, for readability, all examples and syntax statements in this manual are formatted so that each clause of a statement begins on a new line. Clauses that have more than one part extend to additional lines, which are indented.

SQL Syntax Conventions

The conventions for syntax statements in this manual are as follows:

Table 1: Syntax statement conventions

Key	Definition
command or **command**	Command names, command option names, utility names, utility flags, and other keywords are in **bold Courier** in syntax statements, and in **bold Helvetica** in paragraph text.
variable	Variables, or words that stand for values that you fill in, are in italics.
{ }	Curly braces indicate that you choose at least one of the enclosed options. Do not include braces in your option.
[ ]	Square brackets mean choosing one or more of the enclosed options is optional. Do not include brackets in your option.
( )	Parentheses are to be typed as part of the command.
|	The vertical bar means you may select only one of the options shown.
,	The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.

- Syntax statements (displaying the syntax and all options for a command) are printed like this:

  ```
  sp_dropdevice [device_name]
  ```

  or, for a command with more options:

  ```
  select column_name
      from table_name
      where search_conditions
  ```

  In syntax statements, keywords (commands) are in normal font and identifiers are in lowercase: normal font for keywords, italics for user-supplied words.

- Examples showing the use of Transact-SQL commands are printed like this:

  ```
  select * from publishers
  ```

- Examples of output from the computer are printed like this:

```
pub_id   pub_name              city          state
-------  --------------------  -----------   -----
0736     New Age Books          Boston        MA
0877     Binnet & Hardley       Washington    DC
1389     Algodata Infosystems   Berkeley      CA

(3 rows affected)
```

Case

You can disregard case when you type keywords:

SELECT is the same as **Select** is the same as **select**.

Obligatory Options {You Must Choose At Least One}

- **Curly Braces and Vertical Bars –** Choose **one and only one** option.

  {die_on_your_feet | live_on_your_knees | live_on_your_feet}

- **Curly Braces and Commas –** Choose one or more options. If you choose more than one, separate your choices with commas.

  {cash, check, credit}

Optional Options [You Don't Have to Choose Any]

- **One Item in Square Brackets –** You do not have to choose it.

  [anchovies]

- **Square Brackets and Vertical Bars –** Choose **none or only one**.

  [beans | rice | sweet_potatoes]

- **Square Brackets and Commas –** Choose **none, one, or more than one** option. If you choose more than one, separate your choices with commas.

  [extra_cheese, avocados, sour_cream]

Ellipsis: Do It Again (and Again)...

An ellipsis (three dots) means that you can **repeat** the last unit as many times as you like. In this syntax statement, **buy** is a required keyword:

```
buy thing = price [cash | check | credit]
    [, thing = price [cash | check | credit]]...
```

You must buy at least one thing and give its price. You may choose a method of payment, that is, one of the items enclosed in square brackets. You may also choose to buy additional things—as many of them as you like. For each thing you buy, give its name, its price, and (optionally) a method of payment.

Expressions

Several different types of expressions are used in SQL Server syntax statements.

Table 2: Types of expressions used in syntax statements

Usage	Definition
expression	Can include constants, literals, functions, column identifiers, variables, or parameters
logical expression	An expression that returns TRUE, FALSE, or UNKNOWN
constant expression	An expression that always returns the same value, such as “5+3” or “ABCDE”
float_expr	Any floating-point expression or expression that implicitly converts to a floating value
integer_expr	Any integer expression, or an expression that implicitly converts to an integer value
numeric_expr	Any numeric expression that returns a single value
char_expr	Any expression that returns a single character-type value
binary_expression	An expression that returns a single *binary* or *varbinary* value

If You Need Help

Help with your Sybase software is available in the form of documentation and Sybase Technical Support.

Each Sybase installation has a designated person who may contact Technical Support. If you cannot resolve your problem using the manuals, ask the designated person at your site to contact Sybase Technical Support.

1 Introduction

This chapter discusses:

- A general overview of SQL and its components
- The naming conventions used for the different parts of SQL
- The Transact-SQL enhancements (also known as extensions) added to SQL
- ANSI compatibility
- How to use Transact-SQL with the **isql** utility

Overview

SQL (Structured Query Language) is a high-level language for relational database systems. Originally developed by IBM's San Jose Research Laboratory in the late 1970s, SQL has been adopted by and adapted for many relational database management systems. It has been approved as the official relational query language standard by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). Transact-SQL is compatible with IBM SQL and most other commercial implementations of SQL, and also provides important extra capabilities and functions.

Although the "Q" in SQL stands for "Query," SQL includes commands not only for querying (retrieving data from) a database, but also for creating new databases and **database objects**, adding new data, modifying existing data, and other functions.

Queries, Data Modification, and Commands

In this manual, **query** means a request for the retrieval of data, using the **select** command. For example:

```
select au_lname, city, state
from authors
where state = 'NY'
```

Data modification refers to an addition, deletion, or change to data, using the **insert**, **delete**, or **update** command, respectively. For example:

```
insert into authors (au_lname, au_fname, au_id)
values ("Smith", "Gabriella", "999-03-2346")
```

Other SQL commands are instructions to perform administrative operations. For example:

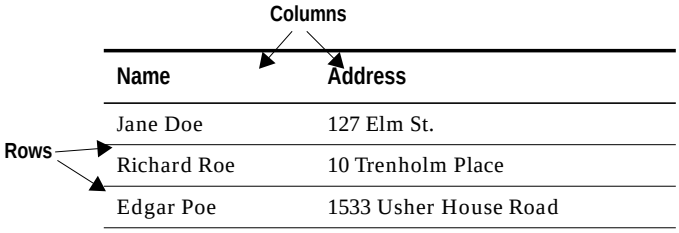
```
drop table authors
```

Each **command** or SQL **statement** begins with a **keyword**, such as **insert**, that names the basic operation performed. Many SQL commands also have one or more **keyword phrases**, or **clauses**, that tailor the command to meet a particular need. When a query is run, Transact-SQL displays the results for the user. If no data meets the criteria specified in the query, the user gets a message to that effect. Data modification statements and administrative statements do not display results, since they do not retrieve data. Transact-SQL provides a message to let the user know whether the data modification or other command has been performed.

Tables, Columns, and Rows

SQL is a database language specifically designed for the relational model of database management. In a relational database management system, users see data as tables, which are also known as relations.

Each row (synonymous with record) of a **table** describes one occurrence of an entity—a person, a company, a sale, or some other thing. Each column, or field, describes one characteristic of the entity—a person's name or address, a company's name or president, a sale's items sold or quantity or date. A database is made up of a set of related tables.



Name	Address
Jane Doe	127 Elm St.
Richard Roe	10 Trenholm Place
Edgar Poe	1533 Usher House Road

Figure 1-1: A table in a relational database

The Relational Operations

The basic query operations in a relational system are selection (also called restriction), projection, and join. All of them can be combined in the SQL **select** command.

A **selection** is a subset of the rows in a table, based on some conditions specified by the user. For example, you might want to look at the rows for all the authors who live in California.

A **projection** is a subset of the columns in a table. For example, a query can display only the name and city of all the authors, omitting the street address, the phone number, and other information.

A **join** links the rows in two or more tables by comparing the values in specified fields. For example, say you have one table containing information about authors, including the columns *au_id* (author identification number) and *au_lname* (author's last name), and another table containing title information about books, including a column (*au_id*) that gives the ID number of the book's author. You might join the *authors* table and the *titles* table, testing for equality of the values in the *au_id* columns of each table. Whenever there is a match, a new row—containing columns from both tables—is created and displayed as part of the result of the join. Joins are often combined with projections and selections so that only selected columns of selected matching rows are displayed.

Naming Conventions

A SQL statement must follow precise syntactical and structural rules, and may include only SQL keywords, identifiers (names of databases, tables, or other database objects), operators, and constants. The characters that can be used for each part of a SQL statement vary from installation to installation and are determined in part by definitions in the default character set that SQL Server uses.

For example, the characters allowed for the SQL language, such as SQL keywords, special characters, and Transact-SQL extensions, are more limited than the characters allowed for identifiers. The set of characters which may be used for data is much larger and includes all the characters that can be used for the SQL language or for identifiers.

Figure 1-2 shows the relationship among the sets of characters allowed for SQL keywords, identifiers, and data.

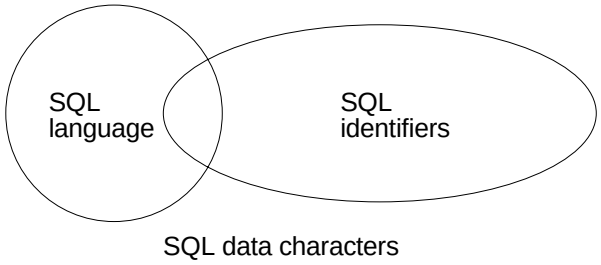


Figure 1-2: Characters used for different parts of SQL statements

The sections that follow describe the sets of characters that can be used for each part of a statement. The section on identifiers also describes naming conventions for database objects.

SQL Data Characters

The set of SQL data characters is the larger set from which both SQL language characters and identifier characters are taken. Any character in SQL Server's character set, including both single-byte and multibyte characters, may be used for data values.

SQL Language Characters

SQL keywords, Transact-SQL extensions, and special characters such as the **comparison operators** ">" and "<", can be represented only by 7-bit ASCII values A-Z, a-z, 0-9, and the following ASCII characters:

Table 1-1: ASCII characters used in SQL

;	(semicolon)	(	(open parenthesis)	)	(close parenthesis)
,	(comma)	:	(colon)	%	(percent sign)
-	(minus sign)	?	(question mark)	'	(single quote)
"	(double quote)	+	(plus sign)	_	(underscore)
*	(asterisk)	/	(slash)		(space)
<	(less than operator)	>	(greater than operator)	=	(equals operator)
&	(ampersand)	|	(vertical bar)	^	(circumflex)
[	(left bracket)	]	(right bracket)	\	(backslash)
@	(at sign)	~	(tilde)	!	(exclamation point)
$	(dollar sign)	#	(number sign)	.	(period)

Identifiers

Conventions for naming database objects apply throughout SQL Server software and documentation. Identifiers can be up to 30 bytes in length, whether or not multibyte characters are used. The first character of an identifier must be declared as an alphabetic character in the character set definition in use on SQL Server.

➤ ***Note***

In multibyte character sets, a wider range of characters is available for use in identifiers. For example, on a server with the Japanese language installed, the following types of characters can be used as the first character of an identifier: Zenkaku or Hankaku Katakana, Hiragana, Kanji, Romaji, Cyrillic, Greek, or ASCII.

The symbols @ or _(underscore) may also be used. The @ symbol as the first character of an identifier indicates a local variable.

Temporary table names must either begin with # (the number sign) if they are created outside *tempdb*, or be preceded by "*tempdb*..". Table names for temporary tables which exist outside *tempdb* should not exceed 13 bytes in length, including the number sign, since SQL Server gives them an internal numeric suffix.

After the first character, identifiers can include characters declared as alphabetic, numeric, or the symbols $, #, @, _, ¥ (yen), or £ (pound sterling).

The case-sensitivity of a SQL Server is set when the server is installed, and can be changed by a System Administrator. To see the setting for your server, execute this command:

```
sp_helpsort
```

On a server that is not case-sensitive, the identifiers *MYOBJECT*, *myobject*, and *MyObject* (and all combinations of case) are considered identical. You can only create one of these objects, and using any of these combinations of case references that object.

No embedded spaces are allowed in identifiers, and none of the SQL reserved keywords can be used. The reserved words are listed in the *SQL Server Reference Supplement*.

You can use the function **valid_name** to determine if an identifier you have created is acceptable to SQL Server. Here is the syntax:

```
select valid_name ("string")
```

where *string* is the identifier you wish to check. If *string* is not valid as an identifier, SQL Server returns a 0 (zero). If *string* is a valid identifier, SQL Server returns a nonzero number. SQL Server returns a 0 if the characters used are illegal or if *string* is more than 30 bytes long.

Delimited Identifiers

Delimited identifiers are object names enclosed in double quotes. Using delimited identifiers allows you to avoid certain restrictions on object names. You can use double quotes to delimit table, view, and column names; you cannot use them for other database objects.

Delimited identifiers can be reserved words, can begin with non-alphabetic characters, and can include characters that would not otherwise be allowed. They cannot exceed 28 bytes.

Before creating or referencing a delimited identifier, you must execute:

```
set quoted_identifier on
```

This option allows SQL Server to recognize delimited identifiers. Each time you use the quoted identifier in a statement, you must enclose it in double quotes. For example:

```
create table "1one"(col1 char(3))
select * from "1one"
create table "include spaces" (col1 int)
```

➤ ***Note***

Delimited identifiers cannot be used as parameters to system procedures or with **bcp**, and may not be supported by all front-end products.

Naming Conventions

The names of database objects need not be unique in a database. However, column names and index names must be unique within a table, and other object names must be unique for each owner within a database. Database names must be unique on SQL Server.

If you try to create a column using a name that is not unique in the table, or to create another **database object** such as a table, a view, or a stored procedure, with a name that you've already used in the same database, SQL Server responds with an error message.

You can uniquely identify a table or column by adding other names that qualify it, that is, the database name, the owner's name, and, for a column, the table name or view name. Each of these qualifiers is separated from the next by a period:

```
database.owner.table_name.column_name
```

```
database.owner.view_name.column_name
```

For example, if the user "sharon" owns the *authors* table in the *pubs2* database, the unique identifier of the *city* column in that table is:

```
pubs2.sharon.authors.city
```

The same naming syntax applies to other database objects. You can refer to any object in a similar fashion:

```
pubs2.dbo.titleview
```

```
dbo.postalcoderule
```

If the **quoted_identifier** option is set **on**, you can use double quotes around individual parts of a qualified object name. Use a separate pair of quotes for each qualifier that requires quotes. For example, use:

```
database.owner."table_name"."column_name"
```

rather than:

```
database.owner."table_name.column_name"
```

The full naming syntax is not always allowed in **create** statements because you cannot create a view, procedure, rule, default, or trigger in a database other than the one you are currently in. The naming conventions are indicated in the syntax as:

```
[[database.]owner.]object_name
```

or:

```
[owner.]object_name
```

The default value for *owner* is the current user, and the default value for *database* is the current database. When you reference an object in SQL statements, other than **create** statements, without qualifying it with the database name and owner name, SQL Server first looks at all the objects you own, and then at the objects owned by the **Database Owner**, whose name in the database is "dbo." As long as SQL Server is given enough information to identify an object, you need not type every element of its name. Intermediate elements can be omitted and their positions indicated by periods:

```
database..table_name
```

When qualifying a column name and a table name in the same statement, be sure to use the same naming abbreviations for each; they are evaluated as strings and must match or an error is returned. Here are two examples with different entries for the column name. The second example does not run because the syntax for the column name does not match the syntax for the table name.

```
select pubs2.dbo.publishers.city
from pubs2.dbo.publishers

city
----------------------
Boston
Washington
Berkeley


select pubs2.dbo.publishers.city
from pubs2..publishers

The column prefix "pubs2.dbo.publishers" does not
match a table name or alias name used in the query.
```

Identifying Remote Servers

Stored procedures can be executed on a remote SQL Server, with the results from the stored procedure printed on the terminal that called the procedure. The syntax for identifying a remote server and the stored procedure is:

[execute] ***server*.[*database*].[*owner*].*procedure_name***

You can omit the **execute** keyword when the remote procedure call is the first statement in a batch. If other SQL statements precede the remote procedure call, you must use **execute** or **exec**. You must give the server name and the stored procedure name. If you omit the database name, SQL Server looks for *procedure_name* in your default database. If you give the database name, you must also give the procedure owner's name, unless you own the procedure, or the procedure is owned by the Database Owner.

The following statements all execute the stored procedure **byroyalty** in the *pubs2* database located on the GATEWAY server:

Statement	Notes
GATEWAY.pubs2.dbo.byroyalty **GATEWAY.pubs2..byroyalty**	**byroyalty** is owned by the Database Owner.

Statement	Notes
`GATEWAY...byroyalty`	Use if *pubs2* is the default database.
`declare @var int` `exec GATEWAY...byroyalty`	Use when the statement is not the first statement in a batch.

See the *SQL Server System Administration Guide* for information on setting up SQL Server for remote access. A remote server name (GATEWAY in the previous example) must match a server name in your local SQL Server's *interfaces* file. If the server name in *interfaces* is in all caps, you must use it in all caps in the remote procedure call.

Transact-SQL Extensions

Transact-SQL has been designed to enhance the power of SQL and to minimize—if not eliminate—the occasions on which users must resort to a programming language to accomplish a desired task. Transact-SQL goes beyond the ISO standards and the many commercial versions of SQL.

Most of the Transact-SQL enhancements (known as extensions) are summarized here. Other extensions, such as the Transact-SQL administration tools, are described in their respective manuals.

The *compute* Clause

The **compute** clause is an important Transact-SQL extension that is used with the row aggregate functions, **sum**, **max**, **min**, **avg**, and **count**, to calculate summary values. The results of a query that includes a **compute** clause are displayed with both detail and summary rows, and look like a report that most DBMSs can produce only with a report generator. **compute** displays summary values as additional rows in the results, instead of as new columns. The **compute** clause is covered in Chapter 3, "Summarizing, Grouping, and Sorting Query Results."

Control-of-Flow Language

Transact-SQL provides control-of-flow language that can be used as part of any SQL statement or batch. These constructs are available: **begin...end**, **break**, **continue**, **declare**, **goto label**, **if...else**, **print**, **raiserror**, **return**,

waitfor, and **while**. Local variables can be defined with **declare** and assigned values. Several predefined global variables are supplied by the system.

Stored Procedures

One of the most important Transact-SQL extensions is the ability to create stored procedures. Stored procedures can combine almost any SQL statements using control-of-flow language. The creator of a stored procedure can also define parameters to be supplied when the stored procedure is executed.

The ability to write your own stored procedures greatly enhances the power, efficiency, and flexibility of the SQL database language. Since the execution plan is saved after stored procedures are run, stored procedures can subsequently run much faster than standalone statements.

SQL Server-supplied stored procedures, called **system procedures,** are provided for your use in SQL Server system administration. Chapter 14, "Using Stored Procedures," discusses the system procedures and explains how to create stored procedures. The system procedures are discussed in detail in the *SQL Server Reference Manual*.

Users can execute stored procedures on remote servers. Other Transact-SQL extensions support return values from stored procedures, user-defined return status from stored procedures, and the ability to pass parameters from a procedure to its caller.

Triggers

A **trigger** is a special kind of stored procedure that is used to protect referential integrity—to enforce rules about the relationships among data in different tables. Triggers go into effect when a user attempts to modify data with an **insert**, **delete**, or **update** command.

A trigger can instruct the system to take any number of actions when a specified change is attempted. By preventing incorrect, unauthorized, or inconsistent changes to data, triggers help maintain the integrity of a database.

Triggers can call local or remote stored procedures, and triggers can call other triggers. Triggers can nest to a depth of 16 levels.

Rules and Defaults

Transact-SQL provides keywords for helping maintain entity integrity (to ensure that a value is supplied for every column that requires a value) and domain integrity (to ensure that each value in a column belongs to the set of legal values for that column). Triggers, described earlier, help maintain referential integrity. Defaults and rules define integrity constraints that come into play during the entry and modification of data.

A default is a value linked to a particular column or datatype, and inserted by the system if no value is provided during data entry. Rules are user-defined integrity constraints linked to a particular column or datatype, and enforced at data entry time. Rules and defaults are discussed in Chapter 12, "Defining Defaults and Rules for Data."

Error Handling and Set Options

A number of error handling techniques are available to the Transact-SQL programmer, including the ability to capture return status from stored procedures, define customized return values from stored procedures, pass parameters from a procedure to its caller, and get reports from global variables such as *@@error*. The **raiserror** and **print** statements, in combination with the control-of-flow language, can direct error messages to the user of a Transact-SQL application. Developers can localize **print** and **raiserror** to use different languages.

set options can customize the display of results, show processing statistics, and provide other diagnostic aids for debugging your Transact-SQL programs.

Additional SQL Server Extensions to SQL

Other unique or unusual features of Transact-SQL include:

- Fewer restrictions on the **group by** clause and the **order by** clause. See Chapter 3, "Summarizing, Grouping, and Sorting Query Results."
- Subqueries, which can be used almost anywhere an expression is allowed. See Chapter 5, "Subqueries: Using Queries Within Other Queries."
- Temporary tables and other temporary database objects, which exist only for the duration of the current work session, and

disappear thereafter. See Chapter 7, “Creating Databases and Tables.”

- User-defined datatypes built on SQL Server-supplied datatypes. See Chapter 7 and Chapter 12, “Defining Defaults and Rules for Data.”
- The ability to **insert** data from a table into that same table. See Chapter 8, “Adding, Changing, and Deleting Data.”
- The ability to extract data from one table and put it into another with the **update** command. See Chapter 8.
- The ability to remove data based on data in other tables using the join in a **delete** statement. See Chapter 8.
- A fast way to delete all rows in a specified table and reclaim the space they took up with the **truncate table** command. See Chapter 8.
- Updates and selections through views. Unlike most other versions of SQL, Transact-SQL places no restrictions at all on retrieving data through views, and relatively few restrictions on updating data through views. See Chapter 9, “Views: Limiting Access to Data.”
- Dozens of built-in functions. See Chapter 10, “Using the Built-In Functions in Queries.”
- Options to the **create index** command for fine-tuning aspects of performance determined by indexes, and controlling the treatment of duplicate keys and rows. See Chapter 11, “Creating Indexes on Tables.”
- User control over what happens when you attempt to enter duplicate keys in a unique index, or duplicate rows in a table. See Chapter 11.
- Bitwise operators for use with *integer* and *bit* type columns. See the *SQL Server Reference Manual*.
- Support for *text* and *image* datatypes. See the *SQL Server Reference Manual*.

Compliance to Standards

The progression of standards for relational database management systems is ongoing. These standards have been and are being adopted by ISO and several national standards bodies. SQL86 was the first of these standards. This was replaced by SQL89, which in turn was replaced by SQL92, which is the current standard. SQL92

defines three levels of conformance: Entry, Intermediate, and Full. In the United States, the National Institute for Standards and Technology (NIST) has defined the Transitional level, which falls between the Entry and Intermediate levels.

Certain behaviors defined by the standards are not compatible with existing SQL Server applications. Transact-SQL provides **set** options that allow you to toggle these behaviors.

Compliant behavior is enabled by default for all Embedded SQL™ precompiler applications. Other applications needing to match SQL standard behavior can use option settings in Table 1-2 for entry level SQL92 compliance. For more information on setting these options, see **set** in the *SQL Server Reference Manual*.

Table 1-2: set options for ANSI compliance

Option	Setting
ansi_permissions	on
ansinull	on
arithabort	off
arithabort numeric_truncation	on
arithignore	off
chained	on
close on endtran	on
fipsflagger	on
quoted_identifier	on
string_rtruncation	on
transaction isolation level	3

The following sections describe the differences between standard behavior and the default Transact-SQL behavior.

FIPS Flagger

For customers writing applications that must conform to the standard, SQL Server provides a **set fipsflagger** option. When this option is turned on, all commands containing Transact-SQL extensions that are not allowed in entry–level SQL92 generate an informational message.

Chained Transactions and Isolation Levels

SQL Server now provides SQL standard-compliant "chained" transaction behavior as an option. In chained mode, all data retrieval and modification commands (**delete**, **insert**, **open**, **fetch**, **select**, and **update**) implicitly begin a **transaction**. Since such behavior is incompatible with many Transact-SQL applications, Transact-SQL style (or "unchained") transactions remain the default.

Chained transaction mode can be initiated with the new **set chained** option. The new **set transaction isolation level** option controls transaction isolation levels. See Chapter 17, "Transactions: Maintaining Data Consistency and Recovery," for more information.

Delimited Identifiers

SQL Server now supports delimited identifiers for table, view, and column names. Delimited identifiers are object names enclosed within double quotation marks. Using them allows you to avoid certain restrictions on object names.

Use the new **set quoted_identifier** option to recognize delimited identifiers. When this option is on, all characters enclosed within double quotes are treated as identifiers. Because this behavior is incompatible with many existing applications, the default setting for this option is **off**.

SQL Standard-Style Comments

In Transact-SQL, comments are delimited by /**/ pairs, and can be nested.Transact-SQL now also supports SQL standard-style comments, which consist of any string beginning with two connected minus signs, a comment, and a terminating newline:

```
select "hello" -- this is a comment
```

The Transact-SQL /**/ comments are still fully supported, and "--" within Transact-SQL comments is still not recognized.

Right Truncation of Character Strings

A new **set** option, **string_rtruncation**, controls silent truncation of character strings for SQL standard compatibility. Set this option **on** to prohibit silent truncation and enforce SQL standard behavior.

Permissions Required for *update* and *delete* Statements

A new **set** option, **ansi_permissions**, determines what permissions are required for **delete** and **update** statements. When this option is **on**, SQL Server uses SQL92's more stringent permissions requirements for these statements. Because this behavior is incompatible with many existing applications, the default setting for this option is **off**.

Arithmetic Errors

The **arithabort** and **arithignore set** options have been redefined to allow compliance with the SQL92 standard:

- **arithabort arith_overflow** specifies behavior following a divide-by-zero error or a loss of precision. The default setting, **on**, rolls back the entire transaction or batch in which the error occurs. If you set **arithabort arith_overflow off**, SQL Server aborts the statement that causes the error but continues to process other statements in the transaction or batch. For compliance to the SQL92 standard, **set arithabort arith_overflow off**.
- **arithabort numeric_truncation** specifies behavior following a loss of scale by an exact numeric type. The default setting, **on**, aborts the statement that causes the error but continues to process other statements in the transaction or batch. If you set **arithabort numeric_truncation off**, SQL Server truncates the query results and continues processing.For compliance to the SQL92 standard, **set arithabort numeric_truncation on**.
- **arithignore arith_overflow** determines whether SQL Server displays a message after a divide-by-zero error or a loss of precision. The default setting, **off**, displays a warning message after these errors. Setting **arithignore arith_overflow on** suppresses warning messages after these errors. For compliance to the SQL92 standard, **set arithignore off**.

Synonymous Keywords

Several keywords have been added for SQL standard compatibility that are synonymous with existing Transact-SQL keywords.

Table 1-3: ANSI-compatible keyword synonyms

Current Syntax	Additional Syntax
tran transaction	work
any	**some**
grant all	**grant all privileges**
revoke all	**revoke all privileges**
max (*expression*)	**max** ([**all** | **distinct**]) *expression*
min (*expression*)	**min** ([**all** | **distinct**]) *expression*
user_name *built-in function*	**user** *keyword*

Treatment of Nulls

A new **set** option, **ansinull**, determines whether or not evaluation of null-valued operands in SQL equality (=) or inequality (!=) comparisons and aggregate functions is SQL standard-compliant. This option does not affect how SQL Server evaluates null values in other kinds of SQL statements, such as **create table**.

How to Use Transact-SQL with the *isql* Utility

You can use SQL directly from the operating system, with the stand-alone utility program **isql**.

To use Transact-SQL, you must set up an account, or login, on SQL Server. When using **isql**, here is what you type at your operating system prompt:

```
isql
```

On your screen, you then see the prompt:

```
Password:
```

Type your password at the prompt and press the Return key. The password is not shown on the screen as you type. Note that login names and passwords are case-sensitive. This is what you will see:

```
1>
```

At this point, you can start issuing Transact-SQL commands.

For detailed information about using **isql**, see the *SQL Server Utility Programs* manual for your operating system.

Choosing a Password

Once you have logged in, you can change your password at any time with the system procedure **sp_password**. Here is how to change the password "terrible2" to "3blindmice":

```
1> sp_password terrible2, 3blindmice
2> go
```

Notice that the word "go" appears on a line by itself and must not be preceded by blanks or tabs. It is the command terminator; it lets SQL Server know that you are done typing, and you are ready for your command to be executed.

Your password is the first line of defense against SQL Server access by unauthorized people. SQL Server passwords must be at least six bytes long and can contain any printable characters. When you are creating your own password, choose one that cannot be guessed. Do not use personal information, names of pets or loved ones, or words that appear in the dictionary.

The most difficult passwords to guess are ones that combine upper- and lowercase characters or numbers and letters. Once you have selected a password, protecting it is your responsibility. Never give anyone your password and never write it down where anyone can see it.

For more information on **sp_password**, see the *SQL Server Reference Manual*. When you execute any stored procedure, a return status displays at the end of execution. A return status of "0" means that execution was successful.

Default Databases

When your SQL Server account was created, you may have been assigned a default database, to which you are connected when you log in. For example, your default database might be *pubs2*, the sample database. If you were not assigned a default database, you are connected to the **master database**.

You can change your default database to any database that you have permission to use, or to any database that allows guests. Any user with a SQL Server login, that is, listed in *master..syslogins*, can be a guest. To change your default database, use the system procedure **sp_modifylogin**. For information on this system procedure, see the *SQL Server Reference Manual*.

In any case, you can make sure you are in *pubs2* by giving this command:

```
1> use pubs2
2> go
```

Now you are ready to start following the examples given in Chapter 2, "Queries: Selecting Data from a Table."

With a couple of exceptions, the examples of Transact-SQL statements shown in the remainder of this manual do not include the line prompts used by the **isql** utility, nor do they include the terminator **go**. For more details on the **isql** utility, see the *SQL Server Utility Programs* manual for your operating system.

Using the *pubs2* Sample Database

The *pubs2* sample database is used for just about every example in this manual. You can try any of the examples on your own workstation.

The query results you see on your screen may not look exactly as they do in this manual. That is because some of the examples here have been reformatted (for example, the columns realigned) for visual clarity or to take up less space on the page.

You may need to get additional permissions to change the sample database using **create** or data modification statements. These permissions can be granted by a System Administrator. If you do change the sample database, be sure to return it to its original state for the sake of future users and uses. Ask for help from a System Administrator if you need help restoring the sample database.

What Is in the Sample Database?

The sample database, *pubs2*, consists of these tables: *publishers, authors, titles, titleauthor, roysched, sales, salesdetail, stores, discounts, au_pix*, and *blurbs*. Most of the examples are drawn from the first four of these tables. The following briefly describes each table:

- *publishers* contains the identification numbers, names, cities, and states of three publishing companies.
- *authors* contains an identification number, first and last name, address information, and contract status for each author.

  For each book that has been or is about to be published, the *titles* table contains its identification number, name, type, identification number of the publisher, price, advance, royalty, year-to-date sales, comments, and publication date.
- *titleauthor* links the *titles* and *authors* tables together. For each book, it contains the title ID, the author ID, the author order, and the royalty split among the authors of a book.
- *roysched* lists the unit sales ranges and the royalty connected with each range. The royalty is some percentage of the net receipts from sales.
- *sales* records the store ID, order number and date of book sales. It acts as the master table for the detail rows in *salesdetail.*
- *salesdetail* records the bookstore sales of titles in the *titles* table.
- *stores* lists bookstores by store ID.
- *discounts* lists three types of discounts for bookstores.
- *au_pix* contains pictures of the authors in binary form using the *image* datatype.
- *blurbs* contains long book descriptions in the *text* datatype.

The sample database is illustrated in the *SQL Server Reference Supplement*.

Part 1: Basic Concepts

2 Queries: Selecting Data from a Table

The **select** command is used to query data from the database. You can use it to retrieve a subset of the rows in one or more tables and to retrieve a subset of the columns in one or more tables.

This chapter discusses:

- Selecting all the columns in a table
- Selecting specified columns in a table
- Changing **select** statement result formats by renaming column heads and adding character strings
- Including simple computed values in a **select** statement
- Eliminating duplicate rows with **distinct**
- Using the **from** clause to specify tables and views
- Using the **where** clause with comparison operators, logical operators, **between**, **in**, **any**, and **like**
- Using **null** and **not null**

This chapter focuses on basic single-table **select** statements. Information on advanced uses of **select** is available in later chapters of this book.

What Are Queries?

A query is the process of requesting data from the database and receiving back the results. This process is also known as **data retrieval**. All SQL queries are expressed using the **select** statement. You can use it for **selections**, which retrieve a subset of the rows in one or more tables, and you can use it for **projections**, which retrieve a subset of the columns in one or more tables

A simplified version of the **select** statement is:

```
select select_list
from table_list
where search_conditions
```

The **select** clause specifies the *columns* you want to retrieve. The **from** clause specifies the *tables* to pull the columns from. The **where** clause specifies which *rows* in the tables you want to see. For example, the following **select** statement finds the first and the last names of writers living in Oakland from the *authors* table.

```
select au_fname, au_lname
from authors
where city = "Oakland"
```

select statement results appear in a columnar format, like this:

```
 au_fname       au_lname
 -------------  -----------
 Marjorie       Green
 Dick           Straight
 Dirk           Stringer
 Stearns        MacFeather
 Livia          Karsen

(5 rows affected)
```

select Syntax

The **select** syntax is both simpler and more complex than the example shown earlier. It is simpler in that the **select** clause is the only required clause in a **select** statement. The **from** clause is almost always included, but technically it is necessary only in **select** statements that retrieve data from tables. The **where** clause is optional, as are all other clauses. On the other hand, the full syntax of the **select** statement includes these phrases and keywords:

```
select [all | distinct] select_list
   [into [[database.]owner.]table_name]
   [from [[database.]owner.]{view_name|table_name
     [(index index_name [ prefetch size ][lru|mru])]}
           [holdlock | noholdlock] [shared]
       [,[[database.]owner.]{view_name|table_name
     [(index index_name [ prefetch size ][lru|mru])]}
           [holdlock | noholdlock] [shared]]... ]

   [where search_conditions]

   [group by [all] aggregate_free_expression
       [, aggregate_free_expression]... ]
   [having search_conditions]
```

```
[order by
{[[[database.]owner.]{table_name.|view_name.}]
    column_name | select_list_number | expression}
        [asc | desc]
[,{[[[database.]owner.]{table_name|view_name.}]
    column_name | select_list_number | expression}
        [asc | desc]]...]

[compute row_aggregate(column_name)
        [, row_aggregate(column_name)]...
    [by column_name [, column_name]...]]

[for {read only | update [of column_name_list]}]

[at isolation {read uncommitted | read committed |
     serialize}]

[for browse]
```

The clauses in a **select** statement must be used in the order shown here. That is, if the statement includes a **group by** clause and an **order by** clause, the **group by** clause must come before the **order by** clause.

As the section "Identifiers" on page 1-5 explains, the names of database objects must be qualified if there is ambiguity about which object is being referred to. For example, if there are several columns called *name*, you may have to qualify *name* with the database name, owner name, or table name.

Since the examples in this chapter involve single-table queries, column names in syntax models and examples are usually not qualified with the names of the tables, owners, or databases to which they belong. These elements are left out for readability; it is never wrong to include qualifiers. The following sections of this chapter analyze the syntax of the **select** statement in more detail.

This chapter describes only some of the clauses and keywords included in the syntax of the **select** command. The **group by**, **having**, **order by**, and **compute** clauses are described in Chapter 3, "Summarizing, Grouping, and Sorting Query Results." The **into** clause is described in Chapter 7, "Creating Databases and Tables." The **at isolation** clause is described in Chapter 17, "Transactions: Maintaining Data Consistency and Recovery."

The **holdlock**, **noholdlock**, and **shared** keywords (which deal with locking in SQL Server) and the **index** clause are described in the *SQL Server*

Performance and Tuning Guide. For information about the **for read only** and **for update** clauses, see the *SQL Server Reference Manual*.

➤ ***Note***

The **for browse** clause is not covered by this manual; it is used only in DB-Library™ applications. See the *Open Client DB-Library/C Reference Manual* for details.

Choosing Columns in a Query

The **select** list frequently consists of a series of column names separated by commas, or an asterisk to represent all columns in **create table** order.

However, it can include one or more expressions, separated by commas, where an expression is a constant, column name, function, subquery, or any combination of these connected by arithmetic or bitwise operators and parentheses. The general syntax for the select list looks like this:

```
select expression [, expression]...
   from table_list
```

If any table or column name in the list does not conform to the rules for valid identifiers, be sure to **set quoted_identifier on** and enclose the identifier in double quotes.

Choosing All Columns: *select* *

The asterisk (*) has a special meaning in **select** statements. It stands for **all** the column names in **all** the tables specified by the **from** clause. Use it to save typing time and errors when you want to see all the columns in a table.

The general syntax for selecting all the columns in a table is:

```
select *
from table_list
```

Because **select *** finds all the columns currently in a table, changes in the structure of a table such as adding, removing, or renaming columns automatically modify the results of a **select ***. Listing the columns individually gives you more precise control over the results.

The following statement retrieves all columns in the *publishers* table and displays them in the order in which they were defined when the publishers table was created. No **where** clause is included, therefore this statement also retrieves every row.

```
select *
from publishers
```

The results look like this:

```
pub_id  pub_name              city          state
-----   --------------        ---------     -----
0736    New Age Books         Boston        WA
0877    Binnet & Hardley      Washington    DC
1389    Algodata Infosystems  Berkeley      CA

(3 rows affected)
```

You get exactly the same results by listing all the column names in the table in order after the **select** keyword:

```
select pub_id, pub_name, city, state
from publishers
```

You can also use * more than once in a query:

```
select *, *
from publishers
```

The effect is to display each column name and each piece of column data twice. Like a column name, * can be qualified with a table name, as in the following query:

```
select publishers.*
from publishers
```

Choosing Some Columns

To select some, but not necessarily all, of the columns in a table, use this syntax:

```
select column_name[, column_name]...
   from table_name
```

Each column name must be separated from the following column name by a comma.

Rearranging the Order of Columns

The order in which you list the column names determines the order in which the columns are displayed. The two following examples show how you specify column order in a display. Both of them find and display the publisher names and identification numbers from all three of the rows in the *publishers* table. The first one prints *pub_id* first, followed by *pub_name*. The second reverses that order. The information is exactly the same; only its organization changes.

```
select pub_id, pub_name
from publishers
pub_id   pub_name
-----    ---------------
0736     New Age Books
0877     Binnet & Hardley
1389     Algodata Infosystems

(3 rows affected)
select pub_name, pub_id
from publishers
pub_name                    pub_id
--------------------        ------
New Age Books               0736
Binnet & Hardley            0877
Algodata Infosystems        1389

(3 rows affected)
```

Renaming Columns in Query Results

When query results are displayed, each column's default heading is the name given to it when it was created. You can specify a column heading by using:

column_heading* = *column_name

or:

column_name column_heading

or:

column_name* as *column_heading

instead of just the column name in a select list. This provides a substitute name for the column. When this name is displayed in the results it functions as a column heading, which can produce more

readable results. For example, to change *pub_name* to "Publisher" in the previous query, type any of the following statements:

```
select Publisher = pub_name, pub_id
from publishers

select pub_name Publisher, pub_id
from publishers

select pub_name as Publisher, pub_id
from publishers
```

The results of these statements look like this:

```
Publisher                pub_id
----------------------   ------
New Age Books            0736
Binnet & Hardley         0877
Algodata Infosystems     1389

(3 rows affected)
```

Quoted Strings in Column Headings

You can include any characters—even blanks—in a column heading if you enclose the entire heading in quotation marks. You do not need to set the **quoted_identifier** option **on**. If the column heading is not enclosed in quotation marks, it must conform to the rules for identifiers. Both of these queries:

```
select "Publisher's Name" = pub_name from
publishers
```

and:

```
select pub_name "Publisher's Name" from publishers
```

produce this result:

```
Publisher's Name
----------------
New Age Books
Binnet & Hardley
Algodata Infosystems
```

In addition, you can use Transact-SQL reserved words in quoted column headings. For example, the following query, using the reserved word **sum** as a column heading, is valid:

```
select "sum" = sum(total_sales) from titles
```

Quoted column headings cannot be more than 30 bytes long.

➤ ***Note***

Before using quotes around a column name in a **create table**, **alter table**, **select into, or create view** statement, you must **set quoted_identifier on**.

Character Strings in Query Results

The **select** statements you have seen so far produce results that consist of data from the tables in the **from** clause. Strings of characters can also be displayed in query results.

Enclose the entire string in single or double quotation marks and separate it from other elements in the select list with commas. Use double quotation marks if there is an apostrophe in the string—otherwise the apostrophe is interpreted as a single quotation mark.

An example statement with a character string is shown here, followed by its results.

```
select "The publisher's name is", Publisher =
pub_name
from publishers

                              Publisher
------------------------      --------------------
The publisher's name is       New Age Books
The publisher's name is       Binnet & Hardley
The publisher's name is       Algodata Infosystems

(3 rows affected)
```

Computed Values in the Select List

You can perform computations with data from numeric columns or on numeric constants in a select list.

Arithmetic Operators

The following table shows the arithmetic operators that are available. For information on bitwise operators, see the *SQL Server Reference Manual*.

Table 2-1: Arithmetic operators

Symbol	Operation
+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulo

The arithmetic operators—addition, subtraction, division, and multiplication—can be used on any numeric column—*int*, *smallint*, *tinyint*, *numeric*, *decimal*, *float*, or *money*. The modulo operator cannot be used on *money* columns. A modulo is the integer remainder after a division operation on two integers. For example, 21 % 9 = 3 because 21 divided by 9 equals 2, with a remainder of 3.

Certain arithmetic operations can also be performed on *datetime* columns, using the date functions. See Chapter 10, "Using the Built-In Functions in Queries," for information on the date functions. All of these operators can be used in the select list with column names and numeric constants in any combination. For example, to see what a projected sales increase of 100 percent for all the books in the *titles* table looks like, type:

```
select title_id, total_sales, total_sales * 2
from titles
```

Here are the results:

```
title_id     total_sales
--------     -----------       ---------
BU1032              4095            8190
BU1111              3876            7752
BU2075             18722           37444
BU7832              4095            8190
MC2222              2032            4064
MC3021             22246           44492
MC3026              NULL            NULL
PC1035              8780           17560
PC8888              4095            8190
PC9999             NULL            NULL
```

```
PS1372              375            750
PS2091             2045           4090
PS2106              111            222
PS3333             4072           8144
PS7777             3336           6672
TC3218              375            750
TC4203            15096          30192
TC7777             4095           8190

(18 rows affected)
```

Notice the null values in the *total_sales* column and the computed column. Null values have no explicitly assigned values. When you perform any arithmetic operation on a null value, the result is NULL. Give the computed column a heading, say "proj_sales", by typing:

```
select title_id, total_sales,
        proj_sales = total_sales * 2
from titles
```

For an even fancier display, try adding character strings such as "Current sales =" and "Projected sales are" to the **select** statement. The column from which the computed column is generated does not have to appear in the select list. The *total_sales* column, for example, is shown in these sample queries only for comparison of its values with the values from the total_sales * 2 column. To see just the computed values, type:

```
select title_id, total_sales * 2
from titles
```

Arithmetic operators also work directly with the data values in specified columns, when no constants are involved. Here's an example:

```
select title_id, total_sales * price
from titles
title_id
--------     ----------
BU1032        81,859.05
BU1111        46,318.20
BU2075        55,978.78
BU7832        81,859.05
MC2222        40,619.68
MC3021        66,515.54
MC3026             NULL
PC1035       201,501.00
PC8888        81,900.00
PC9999             NULL
```

```
PS1372         8,096.25
PS2091        22,392.75
PS2106           777.00
PS3333        81,399.28
PS7777        26,654.64
TC3218         7,856.25
TC4203       180,397.20
TC7777        61,384.05

(18 rows affected)
```

Finally, computed columns can come from more than one table. The chapters on joining and subqueries give information on how to work with multi-table queries.

This query calculates the product of the number of copies of a psychology book sold by an outlet (the *qty* column from the *salesdetail* table) and the price of the book (the *price* column from the *titles* table).

```
select salesdetail.title_id, stor_id, qty * price
from titles, salesdetail
where titles.title_id = salesdetail.title_id
and titles.title_id = "PS2106"

title_id         stor_id
---------------- ------------- ------
PS2106           8042         210.00
PS2106           8042         350.00
PS2106           8042         217.00

(3 rows affected)
```

Arithmetic Operator Precedence

When there is more than one arithmetic operator in an expression, multiplication, division, and modulo are calculated first, followed by subtraction and addition. When all arithmetic operators in an expression have the same level of precedence, the order of execution is left to right. Expressions within parentheses take precedence over all other operations.

For example, the following **select** statement multiplies the total sales of a book by its price to calculate a total dollar amount, then subtracts from that the author's advance divided in half.

The product of *total_sales* and *price* is calculated first, because the operator is multiplication. Next, the advance is divided by 2, and the result is subtracted from *total_sales*.

```
select title_id, total_sales * price - advance / 2
from titles
```

To avoid misunderstandings, use parentheses. The following query has the same meaning and gives the same results as the previous one, but some may find it easier to understand:

```
select title_id,(total_sales * price)- (advance /2)
from titles
```

```
title_id
--------     ----------
 BU1032       79,359.05
 BU1111       43,818.20
 BU2075       50,916.28
 BU7832       79,359.05
 MC2222       40,619.68
 MC3021       59,015.54
 MC3026            NULL
 PC1035      198,001.00
 PC8888       77,900.00
 PC9999            NULL
 PS1372        4,596.25
 PS2091        1,255.25
 PS2106       -2,223.00
 PS3333       80,399.28
 PS7777       24,654.64
 TC3218        4,356.25
 TC4203      178,397.20
 TC7777       57,384.05

(18 rows affected)
```

Use parentheses to change the order of execution; calculations inside parentheses are handled first. If parentheses are nested, the most deeply nested calculation has precedence. For example, the result and meaning of the preceding can be changed if you use parentheses to force evaluation of the subtraction before the division:

```
select title_id, (total_sales * price - advance) /2
from titles
```

```
title_id
--------   ----------------------
BU1032                  38,429.53
BU1111                  20,659.10
BU2075                  22,926.89
BU7832                  38,429.53
MC2222                  20,309.84
MC3021                  25,757.77
MC3026                       NULL
PC1035                  97,250.50
PC8888                  36,950.00
PC9999                       NULL
PS1372                     548.13
PS2091                  10,058.88
PS2106                  -2,611.50
PS3333                  39,699.64
PS7777                  11,327.32
TC3218                     428.13
TC4203                  88,198.60
TC7777                  26,692.03

(18 rows affected)
```

Selecting *text* and *image* Values

When the select list includes *text* and *image* values, the limit on the length of the data returned depends on the setting of the *@@textsize* **global variable**. The default setting for *@@textsize* depends on the software used to access SQL Server; the default value is 32K for **isql**. The value is changed with the **set** command:

```
set textsize 25
```

With this setting of *@@textsize*, a **select** statement that includes a *text* column displays only the first 25 bytes of the data.

➤ ***Note***

When you are selecting image data, the returned value includes the characters "0x", which indicates that the data is hexadecimal. These two characters are counted as part of *@@textsize*.

To reset *@@textsize* to its default value, use:

```
set textsize 0
```

The default display is the actual length of the data when its size is less than *textsize*. For more information about *text* and *image* datatypes, see Chapter 6, "Using and Creating Datatypes."

Using *readtext*

The **readtext** command provides another way to retrieve *text* and *image* values. The **readtext** command needs the name of the table and column, the text pointer, a starting offset within the column, and the number of characters or bytes to retrieve. This example finds 6 characters in the *copy* column in the *blurbs* table:

```
declare @val varbinary(16)
select @val = textptr(copy) from blurbs
where au_id = "648-92-1872"
readtext blurbs.copy @val 2 6 using chars
```

In the example, **readtext** displays characters 3 through 8 of the *copy* column, since the offset was 2. The full syntax of the **readtext** command is:

```
readtext [[database.]owner.]table_name.column_name
   text_ptr offset size [holdlock]
   [using {bytes|chars|characters}]
   [at isolation {read uncommitted | read committed |
       serializable}]
```

The **textptr** function returns a 16-byte binary string. Declare a local variable to hold the text pointer, and then use the variable with **readtext**. The **holdlock** flag causes the text value to be locked for reads until the end of the transaction. Other users can read the value but they cannot modify it. The **at isolation** clause is described in Chapter 17, "Transactions: Maintaining Data Consistency and Recovery."

If you are using a multibyte character set, the **using** option allows you to choose whether you want **readtext** to interpret the offset and size as bytes or as characters. Both **chars** and **characters** are used to specify characters. This option has no effect when used with a single-byte character set or with *image* values (**readtext** reads *image* values only on a byte-by-byte basis). If the **using** option is not given, **readtext** returns the value as if bytes were specified.

SQL Server has to determine the number of bytes to send to the client in response to a **readtext** command. When the offset and size are in bytes, determining the number of bytes in the returned text is simple. When the offset and size are in characters, SQL Server must take an extra step to calculate the number of bytes being returned to the client. As a result, performance may be slower when using characters

as the offset and size. **using characters** is only useful when SQL Server is using a multibyte character set. This option ensures that **readtext** does not return partial characters.

When using bytes as the offset, SQL Server may find partial characters at the beginning or end of the *text* data to be returned. If it does, the server replaces each partial character with question marks before returning the text to the client.

You cannot use **readtext** on *text* and *image* columns in views.

Select List Summary

The **select** list can include * (all columns in create-table order), a list of column names in any order, character strings, column headings, and expressions including arithmetic operators. You can also include aggregate functions, which are discussed in the section on **group by** in this chapter, and in Chapter 3, "Summarizing, Grouping, and Sorting Query Results." Here are some select lists to try with the tables in the *pubs2* sample database:

```
1. select titles.*
   from titles

2. select Name = au_fname, Surname = au_lname
   from authors

3. select Sales = total_sales * price,
   ToAuthor = advance,
   ToPublisher = (total_sales * price) - advance
   from titles

4. select 'Social security #', au_id
   from authors

5. select this_year = advance, next_year = advance
      + advance/10, third_year = advance/2,
       'for book title #', title_id
   from titles

6. select 'Total income is',
   Revenue = price * total_sales,
   'for', Book# = title_id
   from titles
```

Eliminating Duplicate Query Results with *distinct*

The optional **distinct** keyword eliminates duplicate rows from the results of a select statement.

If you don't specify **distinct**, you get all rows, including duplicates. You can optionally specify **all** before the select list, in which case you get all rows. **all** is the default.

For example, if you search for all the author identification codes in the *titleauthor* table without **distinct,** you get these rows:

```
select au_id
from titleauthor

au_id
-----------
172-32-1176
213-46-8915
213-46-8915
238-95-7766
267-41-2394
267-41-2394
274-80-9391
409-56-7008
427-17-2319
472-27-2349
486-29-1786
486-29-1786
648-92-1872
672-71-3249
712-45-1867
722-51-5454
724-80-9391
724-80-9391
756-30-7391
807-91-6654
846-92-7186
899-46-2035
899-46-2035
998-72-3567
998-72-3567

(25 rows affected)
```

Looking at the results, you'll see that there are some duplicate listings. You can eliminate them, and see only the unique *au_ids*, with **distinct**.

```
select distinct au_id
from titleauthor
```

```
au_id
-----------
172-32-1176
213-46-8915
238-95-7766
267-41-2394
274-80-9391
409-56-7008
427-17-2319
472-27-2349
486-29-1786
648-92-1872
672-71-3249
712-45-1867
722-51-5454
724-80-9391
756-30-7391
807-91-6654
846-92-7186
899-46-2035
998-72-3567

(19 rows affected)
```

➤ ***Note***

For compatibility with other implementations of SQL, SQL Server syntax allows the use of the keyword **all** to explicitly ask for all rows. However, there is no reason to use **all**, because "all rows" is the default.

The **distinct** keyword treats null values as duplicates of each other. In other words, when **distinct** is included in a **select** statement, only one NULL is returned in the results, no matter how many null values are encountered.

Specifying Tables: The *from* Clause

The **from** clause is required in every **select** statement involving data from tables or views. Use it to list all the tables and views containing columns included in the select list and in the **where** clause. If the **from** clause includes more than one table or view, separate them with commas.

The maximum number of tables and views allowed in a query is 16. This total includes tables listed in the **from** clause, base tables

referenced by a view definition, any tables referenced in subqueries, and any tables referenced as part of referential integrity constraints..

The **from** syntax looks like this:

```
select select_list
   [from [[database.]owner.]{table_name |view_name}
            [holdlock | noholdlock] [shared]
         [,[[database.]owner.]{table_name |view_name}
            [holdlock | noholdlock] [shared]]... ]
```

Table names can be from 1–30 bytes long. You can use a letter, @, #, or _ as the first character. The following characters can be digits, letters, @, #, $, _, ¥, or £. Temporary table names must either begin with # (number sign) if they are created outside *tempdb*, or with "*tempdb*..". If you create a temporary table outside *tempdb*, its name must be no longer than 13 bytes, since SQL Server attaches an internal numeric suffix to the name to ensure that the name is unique. For more information, see Chapter 7, "Creating Databases and Tables." In the **from** clause, the full naming syntax for tables and views is always permitted, such as:

```
database.owner.table_name
   database.owner.view_name
```

This is only necessary when there might be some confusion about the name. Table names can be given correlation names to save typing. **Correlation names** are assigned in the **from** clause by giving the correlation name after the table name, like this:

```
select p.pub_id, p.pub_name
from publishers p
```

All other references to that table, for example in a **where** clause, must use the correlation name. Correlation names may not begin with a numeral.

Selecting Rows: The *where* Clause

The **where** clause in a **select** statement specifies the criteria for exactly which rows are retrieved. The general format is:

```
select select_list
   from table_list
   where search_conditions
```

Search conditions, or qualifications, in the **where** clause include:

- Comparison operators (=, <, >, and so on)

```
where advance * 2 > total_sales * price
```

- Ranges (**between** and **not between**)

  ```
  where total_sales between 4095 and 12000
  ```

- Lists (**in, not in**)

  ```
  where state in ("CA", "IN", "MD")
  ```

- Character matches (**like** and **not like**)

  ```
  where phone not like "415%"
  ```

- Unknown values (**is null** and **is not null**)

  ```
  where advance is null
  ```

- Combinations of these (**and**, **or**)

  ```
  where advance < 5000 or total_sales between 2000
     and 2500
  ```

In addition, the **where** keyword can introduce:

- Join conditions (see Chapter 4, "Joins: Retrieving Data from Several Tables")
- Subqueries (see Chapter 5, "Subqueries: Using Queries Within Other Queries")

➤ ***Note***

The only **where** condition that you can use on *text* columns is **like** (or **not like**).

For a complete list of the possible search conditions, including a few not mentioned here, see the "Search Conditions" or "**where** Clause" section in the *SQL Server Reference Manual.*

Comparison Operators

Transact-SQL uses the following comparison operators:

Table 2-2: SQL comparison operators

Operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal
!=	Not equal to
<>	Not equal to
!>	Not greater than

Table 2-2: SQL comparison operators

Operator	Meaning
!<	Not less than

The operators are used in the syntax:

where expression comparison_operator expression

where an *expression* is a constant, column name, function, subquery, or any combination of them connected by arithmetic or bitwise operators. In comparing character data, < means earlier in the sort order and > means later in the sort order. (Use the system procedure **sp_helpsort** to see the sort order for your SQL Server.)

Trailing blanks are ignored for the purposes of comparison. So, for example, "Dirk" is the same as "Dirk ". In comparing dates, < means earlier and > means later. Be sure to put apostrophes or quotation marks around all *char*, *nchar*, *varchar*, *nvarchar*, *text*, and *datetime* data. For more information on entering *datetime* data, see Chapter 8, "Adding, Changing, and Deleting Data."

Some sample **select** statements using comparison operators follow:

```
select *
from titleauthor
where royaltyper < 50

select authors.au_lname, authors.au_fname
from authors
where au_lname >'McBadden'

select au_id, phone
from authors
where phone !='415 658-9932'

select title_id, newprice = price * $1.15
from  pubs2..titles
where advance > 5000
```

not negates an expression. Either of the two following queries will find all business and psychology books that do not have an advance of more than $5,500. However, note the difference in position between the negative logical operator (**not**) and the negative comparison operator (**!>**).

```
select title_id, type, advance
from titles
where (type = "business" or type = "psychology")
and not advance >5500

select title_id, type, advance
from titles
where (type = "business" or type = "psychology")
and advance !>5500
title_id  type             advance
--------  ------------    --------
BU1032    business        5,000.00
BU1111    business        5,000.00
BU7832    business        5,000.00
PS2091    psychology      2,275.00
PS3333    psychology      2,000.00
PS7777    psychology      4,000.00

(6 rows affected)
```

Ranges (*between* and *not between*)

Use the **between** keyword to specify an inclusive range, in which the lower value and the upper value are searched for as well as the values they bracket.

For example, to find all the books with sales between and including 4,095 and 12,000, you can write this query:

```
select title_id, total_sales
from titles
where total_sales between 4095 and 12000
title_id  total_sales
------    -----------
BU1032           4095
BU7832           4095
PC1035           8780
PC8888           4095
TC7777           4095

(5 rows affected)
```

Notice that books with sales of 4,095 are included in the results. If there are any with sales of 12,000, they are included, too. You can specify an exclusive range with the greater-than (>) and less-than (<) operators. The same query using the greater-than and less-than

operators returns the following results, because these operators are not inclusive:

```
select title_id, total_sales
from titles
where total_sales > 4095 and total_sales < 12000

title_id  total_sales
------    -----------
PC1035           8780

(1 row affected)
```

not between finds all the rows that are not inside the range. To find all the books with sales outside the 4,095 to 12,000 range, type:

```
select title_id, total_sales
from titles
where total_sales not between 4095 and 12000

title_id     total_sales
--------     -----------
BU1111              3876
BU2075             18722
MC2222              2032
MC3021             22246
PS1372               375
PS2091              2045
PS2106               111
PS3333              4072
PS7777              3336
TC3218               375
TC4203             15096

(11 rows affected)
```

Lists (*in* and *not in*)

The **in** keyword allows you to select values that match any one of a list of values. For example, without **in**, if you want a list of the names and states of all the authors who live in California, Indiana, or Maryland, you can type this query:

```
select au_lname, state
from authors
where state = 'CA' or state = 'IN' or state = 'MD'
```

However, you get the same results with less typing if you use **in**. The items following the **in** keyword must be separated by commas and enclosed in parentheses.

```
select au_lname, state
from authors
where state in('CA', 'IN', 'MD')
```

This is what results from either query:

```
au_lname         state
-----------      -----
White            CA
Green            CA
Carson           CA
O'Leary          CA
Straight         CA
Bennet           CA
Dull             CA
Gringlesby       CA
Locksley         CA
Yokomoto         CA
DeFrance         IN
Stringer         CA
MacFeather       CA
Karsen           CA
Panteley         MD
Hunter           CA
McBadden         CA

(17 rows affected)
```

Perhaps the most important use for the **in** keyword is in nested queries, also referred to as subqueries. For a full discussion of subqueries, see Chapter 5, "Subqueries: Using Queries Within Other Queries." However, the following example gives some idea of what you can do with nested queries and the **in** keyword.

Suppose you want to know the names of the authors who receive less than 50 percent of the total royalties on the books they co-author. The *authors* table gives author names and the *titleauthor* table gives royalty information. By putting the two together using **in**, but without listing the two tables in the same **from** clause, you can extract the information you need. The following query translates as: Find all the *au_id*s in the *titleauthor* table in which the authors make less than 50 percent of the royalty on any one book. Then select from the *authors* table all the author names with *au_id*s that match the results from the *titleauthor* query. The results show that several authors fall into the less than 50 percent category.

```
select au_lname, au_fname
from authors
where au_id in
  (select au_id
   from titleauthor
   where royaltyper <50)
```

```
au_lname          au_fname
--------------    ------------
Green             Marjorie
O'Leary           Michael
O'Leary           Michael
Gringlesby        Burt
Yokomoto          Akiko
MacFeather        Stearns
Ringer            Anne

(7 rows affected)
```

not in finds the authors that do not match the items in the list. The following query finds the names of authors who do not make less than 50 percent of the royalties on at least one book.

```
select au_lname, au_fname
from authors
where au_id not in
  (select au_id
   from titleauthor
   where royaltyper <50)
```

```
au_lname          au_fname
---------------   ------------
White             Johnson
Carson            Cheryl
Straight          Dick
Smith             Meander
Bennet            Abraham
Dull              Ann
Locksley          Chastity
Greene            Morningstar
Blotchet-Halls    Reginald
```

```
del Castillo      Innes
DeFrance          Michel
Stringer          Dirk
Karsen            Livia
Panteley          Sylvia
Hunter            Sheryl
McBadden          Heather
Ringer            Albert

(17 rows affected)
```

Matching Character Strings: *like*

Use the **like** keyword to select rows containing fields that match specified portions of character strings. **like** is used with *char*, *varchar*, *nchar*, *nvarchar*, *binary*, *varbinary*, *text*, and *datetime* data.

The column data is compared to a "match string" that can include these special symbols:

Table 2-3: Special symbols for matching character strings

Symbols	Meaning
%	Matches any string of zero or more characters
_	Matches any one character
[*specifier*]	Brackets enclose ranges or sets, such as [a-f] or [abcdef]. *specifier* may take two forms: *rangespec1* - *rangespec2*: *rangespec1* indicates the start of a range of characters. - is a special character, indicating a range *rangespec2* indicates the end of a range of characters *set*: can be comprised of any discrete set of values, in any order, such as [a2bR]. Note that the range [a-f], and the sets [abcdef] and [fcbdae] will return the same set of values.
[^*specifier*]	caret (^) preceding a specifier indicates non-inclusion. [^a-f] means "not in the range a-f"; [^a2bR] means "not a, 2, b, or R."

The column data can be matched to constants, variables, or other columns which contain these **wildcard** symbols. When using constants, enclose the match strings and character strings in

quotation marks. For example, using **like** with the data in the *authors* table:

- **like** "Mc%" searches for every name that begins with the letters ''Mc'' (McBadden).
- **like** "%inger" searches for every name that ends with ''inger'' (Ringer, Stringer).
- **like** "%en%" searches for every name that has the letters ''en'' in it (Bennet, Green, McBadden).
- **like** "_heryl" searches for every six-letter name ending with ''heryl'' (Cheryl).
- **like** "[CK]ars[eo]n" searches for ''Carsen,'' '' Karsen,'' ''Carson,'' and ''Karson'' (Carson).
- **like** "[M-Z]inger" searches for all the names ending with ''inger'' that begin with any single letter from M to Z (Ringer).
- **like** "M[^c]%" searches for all names beginning with ''M'' not having ''c'' as the second letter.

This query finds all the phone numbers in the *authors* table that have 415 as the area code:

```
select phone
from authors
where phone like "415%"
```

You can use **not like** with the same wildcard characters. To find all the phone numbers in the *authors* table that do *not* have 415 as the area code, you could use either of these queries:

```
select phone
from authors
where phone not like "415%"
```

```
select phone
from authors
where not phone like "415%"
```

The only **where** condition that you can use on *text* columns is **like.** This query will find all the rows in the *blurbs* table where the *copy* column mentions the word "computer":

```
select * from blurbs
where copy like "%computer%"
```

Wildcard characters used without **like** are interpreted as literals rather than as a pattern; they represent exactly their own values. The following query attempts to find any phone numbers that consist of

the four characters “415%” only. It does not find phone numbers that start with 415.

```
select phone
from authors
where phone = "415%"
```

Using Wildcard Characters As Literal Characters

You can search for wildcard characters by escaping them and searching for them as literals. There are two ways to use the wildcard characters as literals in a **like** match string: square brackets and the **escape** clause.

The match string can also be a variable or a value in a table that contains a wildcard character. For more information about **like** and the wildcard characters (including using **like** with multibyte character sets and sort orders that are not case-sensitive), see the *SQL Server Reference Manual*.

Square Brackets (Transact-SQL Extension)

Use square brackets as characters for the percent sign, the underscore, and the open bracket. The close bracket does not need an escape character; use it by itself. To search for a dash, rather than using it to specify a range for which to search, use the dash as the first character inside a set of brackets.

Table 2-4: Using square brackets to search for wildcard characters

like Predicate	Meaning
`like "5%"`	5 followed by any string of 0 or more characters
`like "5[%]"`	5%
`like "_n"`	an, in, on, and so forth
`like "[_]n"`	_n
`like "[a-cdf]"`	a, b, c, d, or f
`like "[-acdf]"`	-, a, c, d, or f
`like "[[]"`	[
`like "]"`	]

escape Clause (SQL Standard Compliant)

Use the **escape** clause to specify an escape character in the **like** predicate:

Table 2-5: Using the escape clause

like Predicate	Meaning
`like "5@%" escape "@"`	5%
`like "*_n" escape "*"`	_n
`like "%80@%%" escape "@"`	string containing 80%
`like "*_sql**%" escape "*"`	string containing _sql*
`like "%#####_#%%" escape "#"`	string containing ##_%

- An escape character must be a single character string. Any character in the server's default character set can be used. Specifying more than one escape character raises a SQLSTATE error condition, and SQL Server returns an error message.

  For example, the following escape clauses cause this error condition:

  ```
  like "%XX_%" escape "XX"
  like "%XX%X_%" escape "XX"
  ```

- An escape character is valid only within its **like** predicate and has no effect on other **like** predicates contained in the same statement.
- The only characters that are valid following an escape character are the wildcard characters (_ , % , [,] , or [^]), and the escape character itself. The escape character affects only the character following it, and subsequent characters are not affected by it. If the pattern contains two literal occurrences of a character that happens to be an escape character, the string must contain four consecutive escape characters (see the fifth example in *Table 2-5: Using the escape clause*). If not, SQL Server raises a SQLSTATE error condition and returns an error message.

  For example, the following escape clauses cause this error condition:

  ```
  like "P%X%%X" escape "X"
  like "%X%%Xd_%" escape "X"
  like "%?X%" escape "?"
  like "_e%&u%" escape "&"
  ```

Interaction of Square Brackets and the *escape* Clause

An escape character retains its special meaning within square brackets, unlike the wildcard characters such as the underscore, the percent sign, and the open bracket.

It is recommended that you not use existing wildcard characters as escape characters for these reasons:

- If you specify the underscore (_) or percent sign (%) as escape characters, they lose their special meaning within that **like** predicate, and act only as escape characters.
- If you specify the open or close bracket ([or]) as escape characters, the Transact-SQL meaning of the brackets is disabled within that **like** predicate.
- If you specify - or [^] as escape characters, they lose the special meaning that they normally have within square brackets, and act only as escape characters.

Trailing Blanks and %

Trailing blanks following "%" in a **like** clause are truncated to a single trailing blank. **like** "% " (percent followed by 2 spaces) matches "X " (one space); "X " (two spaces); "X " (three spaces), or any number of trailing spaces.

Using Wildcard Characters in Columns

Wildcard characters can be used in columns, and the column names in **like** clauses. A table called *special_discounts* in the *pubs2* database could be created to run a price projection for a special sale:

```
id_type discount
------- -----------
BU%              10
PS%              12
MC%              15
```

The following query uses wildcard characters in *id_type* in the **where** clause:

```
select title_id, discount, price, price -
(price*discount/100)
from special_discounts, titles
where title_id like id_type
```

Here are the results of that query:

```
title_id  discount   price
--------- ---------- -------------- --------------
BU1032            10          19.99          17.99
BU1111            10          11.95          10.76
BU2075            10           2.99           2.69
BU7832            10          19.99          17.99
PS1372            12          21.59          19.00
PS2091            12          10.95           9.64
PS2106            12           7.00           6.16
PS3333            12          19.99          17.59
PS7777            12           7.99           7.03
MC2222            15          19.99          16.99
MC3021            15           2.99           2.54
MC3026            15           NULL           NULL

(12 rows affected)
```

This permits sophisticated pattern matching without having to construct a series of **or** clauses.

Character Strings and Quotation Marks

When you enter or search for character and date data (*char*, *nchar*, *varchar*, *nvarchar*, *datetime*, and *smalldatetime* datatypes), you must enclose it in single or double quotation marks.

➤ ***Note***

If the **quoted_identifier** option is set to **on**, do not use double quotes around character or date data. You must use single quotes, or SQL Server will treat the data as though it is an identifier.

There are two ways to specify literal quotations within a character entry. The first method is to use two consecutive quotation marks. For example, if you have begun a character entry with a single quotation mark and wish to include a single quotation mark as part of the entry, use two single quotation marks:

```
'I don''t understand.'
```

With double quotation marks:

```
"He said, ""It is not really confusing."""
```

The second method is to enclose a quotation in the other kind of quotation mark. In other words, surround an entry containing

double quotation marks with single quotation marks, or vice versa. Here are some examples:

```
'George said, "There must be a better way."'
"Isn't there a better way?"
'George asked, "Isn''t there a better way?"'
```

To continue a character string that would go off the end of one line on your screen, enter a backslash (\) before going to the following line.

"Unknown" Values: NULL

When you see NULL in a column, it means that the user or application has made no entry in that column. A data value for the column is "unknown" or "not available."

NULL is **not** synonymous with "zero" (numerical values) or "blank" (character values). Rather, null values allow you to distinguish between a deliberate entry of zero for numeric columns or blank for character columns and a non-entry which is NULL for both numeric and character columns.

NULL can be entered in a column for which null values are permitted, as specified in the **create table** statement, in two ways:

- If no data is entered, SQL Server automatically enters the value NULL.
- The user can explicitly enter the value NULL by typing the word "NULL" or "null" **without** single or double quotation marks.

If the word "NULL" is typed into a character column **with** single or double quotation marks, it is treated as data, not as a null value.

When null values are retrieved, displays of query results show the word NULL in the appropriate position. For example, the *advance* column of the *titles* table allows null values. By inspecting the data in that column you can tell whether a book had **no** advance payment by agreement (zero in the advance column as in the row for MC2222) or whether the advance amount was **not known** when the data was entered (NULL in the advance column, as in the row for MC3026).

```
select title_id, type, advance
from titles
where pub_id = "0877"
```

```
title_id     type            advance
--------     ----------      ---------
MC2222       mod_cook             0.00
MC3021       mod_cook        15,000.00
MC3026       UNDECIDED            NULL
PS1372       psychology       7,000.00
TC3218       trad_cook        7,000.00
TC4203       trad_cook        4,000.00
TC7777       trad_cook        8,000.00

(7 rows affected)
```

Transact-SQL treats null values in different ways, depending on the **operators** that you use and the type of values you are comparing. These operators will return results when used with a NULL:

- = returns all rows that contain NULL.
- != or <> returns all rows that do **not** contain NULL.

However, when **set ansinull** is **on** for compliance with the SQL standard, the = and != operators will not return results when used with a NULL. Regardless of the **set ansinull** option value, the following operators will never return values when used with a NULL: <, <=, !<, >, >=, !>.

SQL Server can determine that a column value is NULL, thus

```
column1 =  NULL
```

can be considered to be true. However, the comparison

```
where column1 > null
```

can never be determined, since NULL means “having an unknown value.” There is no reason to assume that two unknown values are the same.

This logic also applies when using two column names in a **where** clause, that is, when joining two tables. With a clause like “where column1 = column2” never returns any rows where the columns contain null values.

You can also find null values or non-null values in the database with this pattern:

```
where column_name is [not] null
```

If you try to find null values in columns defined as NOT NULL, SQL Server displays an error message.

Some of the rows in the *titles* table contain some incomplete data. For example, a book called *The Psychology of Computer Cooking* has been

proposed and its title, title identification number, and probable publisher entered. However, since the author has no contract as yet and details are still up in the air, null values appear in the *price*, *advance*, *royalty*, *total_sales*, and *notes* columns. Because null values don't match anything in a comparison, a query for all the title identification numbers and advances for books with moderate advances (under $5,000) will not find the row for *The Psychology of Computer Cooking*, title identification number MC3026.

```
select title_id, advance
from titles
where advance < $5000
```

```
title_id  advance
--------  ----------
MC2222         0.00
PS2091     2,275.00
PS3333     2,000.00
PS7777     4,000.00
TC4203     4,000.00

(5 rows affected)
```

Here is a query for books with an advance under $5,000 **or** a null value in the *advance* column:

```
select title_id, advance
from titles
where advance < $5000
  or advance is null
```

```
title_id  advance
--------  ----------
MC2222         0.00
MC3026         NULL
PC9999         NULL
PS2091      2,275.00
PS3333      2,000.00
PS7777      4,000.00
TC4203      4,000.00

(7 rows affected)
```

See Chapter 7, "Creating Databases and Tables," for information on NULL in the **create table** statement and for information on the relationship between NULL and defaults. See Chapter 8, "Adding, Changing, and Deleting Data," for information on inserting null values into a table. See the "Null Values" section in the *SQL Server Reference Manual* for more information.

Connecting Conditions with Logical Operators

The **logical operators** **and**, **or**, and **not** are used to connect search conditions in **where** clauses.

and joins two or more conditions and returns results only when **all** of the conditions are true. For example, the following query finds only the rows in which the author's last name is Ringer and the author's first name is Anne. It does not find the row for Albert Ringer.

```
select *
from authors
where au_lname = 'Ringer' and au_fname = 'Anne'
```

or also connects two or more conditions, but it returns results when **any** of the conditions is true. The following query searches for rows containing Anne or Ann in the *au_fname* column.

```
select *
from authors
where au_fname = 'Anne' or au_fname = 'Ann'
```

not negates the expression that follows it. The following query selects all the authors who do not live in California:

```
select * from authors
where not state = "CA"
```

Logical Operator Precedence

Arithmetic and bitwise operators are handled before logical operators. When more than one logical operator is used in a statement, **not** is evaluated first, then **and**, and finally **or**. See the *SQL Server Reference Manual* for information on bitwise operators.

For example, the following query finds **all** the business books in the *titles* table, no matter what their advances are, as well as all psychology books that have an advance greater than $5,500. The advance condition pertains to psychology books and not to business books because the **and** is handled before the **or**.

```
select title_id, type, advance
from titles
where type = "business" or type = "psychology"
  and advance >5500
```

```
title_id  type        advance
--------  ----------  ----------

BU1032    business     5,000.00
BU1111    business     5,000.00
BU2075    business    10,125.00
BU7832    business     5,000.00
PS1372    psychology   7,000.00
PS2106    psychology   6,000.00

(6 rows affected)
```

You can change the meaning of the query by adding parentheses to force evaluation of the **or** first. This query finds all business and psychology books that have advances over $5,500:

```
select title_id, type, advance
from titles
where (type = "business" or type = "psychology")
  and advance >5500
title_id  type         advance
--------  ----------   ---------
BU2075    business     10,125.00
PS1372    psychology    7,000.00
PS2106    psychology    6,000.00

(3 rows affected)
```

3 Summarizing, Grouping, and Sorting Query Results

You can summarize, group, and sort the results of queries using aggregate functions, the **group by** clause, the **having** clause, and the **order by** clause with the **select** statement. You can also use the **compute** clause (a Transact-SQL extension) with aggregate functions to produce a report with detail and summary rows. The **union** operator allows you to combine the results of queries.

This chapter discusses:

- How to summarize query results using aggregate functions
- How to organize query results into groups
- How to select groups of data
- How to sort query results
- How to summarize groups of data
- How to combine the results of queries

If your SQL Server is not case-sensitive, see **group by** and **compute** in the *SQL Server Reference Manual* for examples on how case sensitivity affects the data returned by these clauses.

Summarizing Query Results Using Aggregate Functions

The aggregate functions calculate summary values from the data in a particular column.

Aggregate functions can be applied to all the rows in a table, to a subset of the table specified by a **where** clause, or to one or more groups of rows in the table. From each set of rows to which an **aggregate function** is applied, a single value is generated.

This example calculates the sum of year-to-date sales for all books in the *titles* table:

```
select sum(total_sales)
from titles

-------------
        97446

(1 row affected)
```

Notice that to use the aggregate functions, you give the function name followed by the name of the column on whose values it will

operate. Put the column name, which is the function's argument, in parentheses.

The general syntax of the aggregate functions is:

aggregate_function **([all|distinct]** ***expression*****)**

The aggregate operators are **sum**, **avg**, **max**, **min**, **count**, and **count(*)**. The optional keyword **distinct** can be used with **sum**, **avg**, and **count** to eliminate duplicate values before the aggregate function is applied. **distinct** is not allowed with **max**, **min**, or **count**(*). For **sum**, **avg**, and **count**, the default is **all** which performs the operation on all rows. The keyword **all** is optional.

The "expression" to which the syntax statement refers is usually a column name. It can also be a constant, a function, or any combination of column names, constants, and functions connected by arithmetic or bitwise operators. An expression can also be a subquery.

For example, you can find what the average price of all books would be if the prices were doubled with this statement:

```
select avg(price * 2)
from titles

-------------
        29.53

(1 row affected)
```

The syntax of the aggregate functions and the results they produce are:

Table 3-1: Syntax and results of aggregate functions

Aggregate Function	Result
sum([all | distinct] ***expression*****)**	The total of the (distinct) values in the expression
avg([all | distinct] ***expression*****)**	The average of the (distinct) values in the expression
count([all | distinct] ***expression*****)**	The number of (distinct) non-null values in the expression
count(*)	The number of selected rows
max(*expression*)	The highest value in the expression
min(*expression*)	The lowest value in the expression

The aggregate functions can be used in a select list, as in the previous examples, or in the **having** clause of a **select** statement that includes a **group by** clause. For information about the **having** clause, see "Selecting Groups of Data: The having Clause" on page 3-19.

Aggregate functions **cannot** be used in a **where** clause.

However, a **select** statement with aggregate functions in its select list often includes a **where** clause that restricts the rows to which the aggregate is applied. In the examples given earlier, each aggregate function produced a single summary value for the whole table.

If a **select** statement includes a **where** clause, but not a **group by** clause, an aggregate function produces a single value for the subset of rows that the **where** clause specifies. However, a **select** may also include a column in its select list (a Transact-SQL extension), which would repeat the single value for each row in the result table. In that case, you can qualify the rows using the **having** clause, which is described in "Selecting Groups of Data: The having Clause" on page 3-19.

This query returns the average advance and the sum of year-to-date sales for business books only:

```
select avg(advance), sum(total_sales)
from titles
where type = "business"

------------ ---------
     6,281.25     30788

(1 row affected)
```

Whenever an aggregate function is used in a **select** statement that does not include a **group by** clause, it produces a single value. This is true whether it is operating on all the rows in a table or on a subset of rows defined by a **where** clause. It is called a **scalar aggregate**.

Note that you can use more than one aggregate function in the same select list, and produce more than one scalar aggregate in a single **select** statement.

Aggregate Functions and Datatypes

sum and **avg** can be used with numeric columns only—*int*, *smallint*, *tinyint*, *decimal*, *numeric*, *float*, and *money*.

min and **max** cannot be used with *bit* datatypes.

Aggregate functions other than **count(*)** cannot be used with *text* and *image* datatypes.

With these exceptions, the aggregate functions can be used with any type of column. For example, you can use **min** (minimum) to find the lowest value—the one closest to the beginning of the alphabet—in a character type column:

```
select min(au_lname)
from authors

--------------------------
Bennet

(1 row affected)
```

Using *count(*)*

count(*) does not require an expression as an argument because, by definition, it does not use information about any particular column. It is used to find the total number of rows in a table. This statement finds the total number of books:

```
select count(*)
from titles

-----------------
                18

(1 row affected)
```

count(*) returns the number of rows in the specified table without eliminating duplicates. It counts each row separately, including rows that contain null values.

Like other aggregate functions, **count(*)** can be combined with other aggregates in the select list, with **where** clauses, and so on:

```
select count(*), avg(price)
from titles
where advance > 1000

---------- ---------
        15     14.42
(1 row affected)
```

Using Aggregate Functions with *distinct*

The **distinct** keyword is optional with **sum**, **avg**, and **count**. It is not allowed with **min**, **max**, or **count(*)**. When **distinct** is used, duplicate values are eliminated before the sum, average, or count is calculated.

If you use **distinct**, the argument cannot include an arithmetic expression. It must consist of a column name only.

When **distinct** is used, it appears inside the parentheses and before the column name. For example, to find the number of different cities in which there are authors, type:

```
select count(distinct city)
from authors

-------------
           16

(1 row affected)
```

The following statement returns the average of the distinct prices of business books:

```
select avg(distinct price)
from titles
where type = "business"

-------------
        11.64

(1 row affected)
```

If two or more books have the same price and you use the **distinct** keyword, the shared price is included only once in the calculation, For an accurate calculation of the average price of business books, you would omit **distinct**:

```
select avg(price)
from titles
where type = "business"


-------------
        13.73

(1 row affected)
```

Null Values and the Aggregate Functions

Any null values in the column on which the aggregate function is operating are ignored for the purposes of the function. If you have set **ansinull** to **on**, the SQL Server returns an error message whenever a null value is ignored. For more information, see the **set** command in the *SQL Server Reference Manual*.

If all the values in a column are null, **count(*column_name*)** returns zero. For example, if you ask for the **count** of advances in the *titles* table, your answer is not the same as if you ask for the **count** of title names, because of the null values in the *advance* column:

```
select count(advance)
from titles

-------------
           16

(1 row affected)
select count(title)
from titles

-------------
           18

(1 row affected)
```

The exception to this rule is **count(*)**, which counts each row, even if every field in it is NULL.

If no rows meet the conditions specified in the **where** clause, **count** returns a value of zero. The other functions all return NULL. Here are examples:

```
select count(distinct title)
from titles
where type = "poetry"

-------------
            0

(1 row affected)
select avg(advance)
from titles
where type = "poetry"

-------------
         NULL

 (1 row affected)
```

Organizing Query Results into Groups: The *group by* Clause

The **group by** clause is used in **select** statements to divide the output of a table into groups. You can group by one or more column names, or by the results of computed columns using numeric datatypes in an expression. For **group by**, the maximum number of columns or expressions is 16.

➤ ***Note***

You cannot group by columns of *text* or *image* datatypes.

A **group by** clause almost always appears in statements that also include aggregate functions, in which case the aggregate produces a value for each group. These values are called **vector aggregates**. Remember, a **scalar aggregate** is a single value produced by an aggregate function without a **group by** clause.

In this example of a vector aggregate, the statement finds the average advance and sum of year-to-date sales for each type of book:

```
select type, avg(advance), sum(total_sales)
from titles
group by type

type
------------     ---------   -------
UNDECIDED        NULL        NULL
business         6,281.25    30788
mod_cook         7,500.00    24278
popular_comp     7,500.00    12875
psychology       4,255.00     9939
trad_cook        6,333.33    19566

(6 rows affected)
```

The summary values (vector aggregates) produced by **select** statements with aggregates and a **group by** appear as columns in each row of the results. By contrast, the summary values (scalar aggregates) produced by **select** statements with aggregates and no **group by** also appear as columns, but with only one row. For example:

```
select avg(advance), sum(total_sales)
from titles

---------   -------
5,962.50    97446

(1 row affected)
```

While it is possible to use **group by** without aggregates, such a construction has very limited functionality, and sometimes produces confusing results. The following example attempts to group the results by title type:

```
select type, advance
from titles
group by type
type            advance
------------    ---------
business         5,000.00
business         5,000.00
business        10,125.00
business         5,000.00
mod_cook             0.00
mod_cook        15,000.00
UNDECIDED            NULL
popular_comp     7,000.00
popular_comp     8,000.00
popular_comp         NULL
psychology       7,000.00
psychology       2,275.00
psychology       6,000.00
psychology       2,000.00
psychology       4,000.00
trad_cook        7,000.00
trad_cook        4,000.00
trad_cook        8,000.00

(18 rows affected)
```

Without an aggregate for the *advance* column, the query returns values for every row in the table.

group by Syntax

The complete syntax of the **select** statement is repeated here so that you can see the **group by** clause in context:

```
select [all | distinct] select_list
   [into [[database.]owner.]table_name]
   [from [[database.]owner.]{view_name|table_name
     [(index index_name [ prefetch size ][lru|mru])]}
           [holdlock | noholdlock] [shared]
       [,[[database.]owner.]{view_name|table_name
     [(index index_name [ prefetch size ][lru|mru])]}
           [holdlock | noholdlock] [shared]]... ]
```

```
[where search_conditions]

[group by [all] aggregate_free_expression
    [, aggregate_free_expression]... ]
[having search_conditions]

[order by
{[[[database.]owner.]{table_name.|view_name.}]
    column_name | select_list_number | expression}
        [asc | desc]
[,{[[[database.]owner.]{table_name|view_name.}]
    column_name | select_list_number | expression}
        [asc | desc]]...]

[compute row_aggregate(column_name)
        [, row_aggregate(column_name)]...
    [by column_name [, column_name]...]]

[for {read only | update [of column_name_list]}]

[at isolation {read uncommitted | read committed |
     serializable}]

[for browse]
```

Remember that the order of the clauses in the **select** statement **is** significant. You can omit any of the optional clauses, but when you use them they must appear in the order shown here.

The SQL standards for **group by** are more restrictive than what is shown in the above syntax. The standard requires that:

- The columns in a select list must also be in the **group by** expression or they must be arguments of aggregate functions.
- A **group by** expression can only contain column names that are in the select list, but not those used only as arguments for vector aggregates.

The results of a standard **group by** with vector aggregate functions produce one row and one summary value per group. Several Transact-SQL extensions (described in the following sections) relax these restrictions, but at the expense of more complex results. If you want to refrain from using the extensions, you can set the **fipsflagger** option as follows:

```
set fipsflagger on
```

This option displays a warning message whenever Transact-SQL extensions are used. For more information about the **fipsflagger** option and the **set** command, see the *SQL Server Reference Manual.*

You can list more than one column in the **group by** clause in order to nest groups—that is, you can group a table by any combination of columns. For example, here is the statement that finds the average price and the sum of the year-to-date sales, grouped first by publisher identification number and then by type:

```
select pub_id, type, avg(price), sum(total_sales)
from titles
group by pub_id, type
```

```
pub_id  type
------  ------------  ------  -------
0736    business       2.99   18722
0736    psychology    11.48    9564
0877    UNDECIDED      NULL    NULL
0877    mod_cook      11.49   24278
0877    psychology    21.59     375
0877    trad_cook     15.96   19566
1389    business      17.31   12066
1389    popular_comp  21.48   12875

(8 rows affected)
```

You can nest many groups within groups, up to the maximum of 16 columns or expressions specified with **group by.**

Referencing Other Columns in Queries Using *group by*

Through extensions to the SQL standards, Transact-SQL does not place restrictions on what you can include or omit in the select list of a **select** statement that includes **group by**:

1. The columns in the select list are **not** limited to the grouping columns and columns used with the vector aggregates.
2. The columns specified by **group by** are **not** limited to those non-aggregate columns in the select list.

A vector aggregate requires that one or more columns appear with the **group by** clause. The SQL standards require that the non-aggregate columns in the select list match the **group by** columns. However, the first extension described above allows you to specify additional "extended" columns in the select list of the query.

For example, the inclusion of the extended *title_id* column in the select list would not be allowed in many versions of SQL, but is perfectly legal in Transact-SQL:

```
select type, title_id, avg(price), avg(advance)
from titles
group by type
type           title_id
------------   --------    -----   ---------
business       BU1032      13.73   6,281.25
business       BU1111      13.73   6,281.25
business       BU2075      13.73   6,281.25
business       BU7832      13.73   6,281.25
mod_cook       MC2222      11.49   7,500.00
mod_cook       MC3021      11.49   7,500.00
UNDECIDED      MC3026      NULL       NULL
popular_comp   PC1035      21.48   7,500.00
popular_comp   PC8888      21.48   7,500.00
popular_comp   PC9999      21.48   7,500.00
psychology     PS1372      13.50   4,255.00
psychology     PS2091      13.50   4,255.00
psychology     PS2106      13.50   4,255.00
psychology     PS3333      13.50   4,255.00
psychology     PS7777      13.50   4,255.00
trad_cook      TC3218      15.96   6,333.33
trad_cook      TC4203      15.96   6,333.33
trad_cook      TC7777      15.96   6,333.33

(18 rows affected)
```

The above example still aggregates the *price* and *advance* columns based on the *type* column, but its results also display the *title_id* for the books included in each group.

The second extension described above allows you to group columns that are not specified as columns in the select list of the query. These columns do not appear in the results, but the vector aggregates still compute their summary values. For example:

```
select state, count(au_id)
from authors
group by state, city
```

```
state
-----  --------
AU            1
CA            2
CA            1
CA            5
CA            2
CA            1
CA            1
CA            1
CA            1
IN            1
KS            1
MD            1
MI            1
OR            1
TN            1
UT            2

(16 rows affected)
```

This example groups the vector aggregate results by both *state* and *city*, even though it does not display which city belongs to each group.

As you can see, the results from such queries using these extensions are more complex. The queries require you to know how SQL Server treats these extensions for you to understand the results. For example, you may think the following query should produce similar results to the previous query, since only the vector aggregate seems to tally the number of each city for each row:

```
select state, count(au_id)
from authors
group by city
```

However, its results are much different (and misleading). By not using **group by** with both the *state* and *city* columns, the query tallies the number of each city, but it displays the tally for each row of that city in *authors* rather than group them into one result row per city.

When you use the Transact-SQL extensions in complex queries that include the **where** clause or joins, the results may become even more difficult to comprehend. To avoid confusing or misleading results with **group by**, you should not use the extensions frivolously. Use the **fipsflagger** option (see page 3-10) to identify queries that use these extensions.

For more information about these Transact-SQL extensions to **group by** and how they work, see the *SQL Server Reference Manual.*

Expressions and *group by*

Another Transact-SQL extension to SQL is that you can group by an expression that does not include aggregate functions. With standard SQL, you can group only by column names. For example:

```
select avg(total_sales), total_sales * price
from titles
group by total_sales * price

---------      -----------------
      111              777.00
      375            7,856.25
      375            8,096.25
     2045           22,392.75
     3336           26,654.64
     2032           40,619.68
     3876           46,318.20
    18722           55,978.78
     4095           61,384.05
    22246           66,515.54
     4072           81,399.28
     4095           81,859.05
     4095           81,900.00
    15096          180,397.20
     8780          201,501.00

(15 rows affected)
```

You cannot **group by** a column heading or **alias**, although you can still use aliases in your select list. This statement produces an error message:

```
select Category = type, title_id, avg(price),
avg(advance)
from titles
group by Category /* Wrong use of column heading */
```

Use **group by** *type* to correct this query.

Nesting Aggregates with *group by*

Another kind of nesting—nesting a vector aggregate inside a scalar aggregate—is a Transact-SQL extension. For example, to find the average price of all the types of books, the query is:

```
select avg(price)
from titles
group by type

---------------
NULL
13.73
11.49
21.48
13.50
15.96

(6 rows affected)
```

You can find the highest average price of a group of books, grouped by type, in a single query by nesting the average price inside the **max** function:

```
select max(avg(price))
from titles
group by type

-------------
        21.48

(1 row affected)
```

By definition, the **group by** clause applies to the innermost aggregate—in this case, **avg**.

Null Values and *group by*

If the grouping column contains a null value, that row becomes a group in the results. If the grouping column contains more than one null value, the null values are put into a single group.

The *advance* column in the *titles* table contains some null values. Here's an example that uses **group by** and the *advance* column:

```
select advance, avg(price * 2)
from titles
group by advance
```

```
advance
------------------  -----------------
              NULL               NULL
              0.00              39.98
         2,000.00              39.98
         2,275.00              21.90
         4,000.00              19.94
         5,000.00              34.62
         6,000.00              14.00
         7,000.00              43.66
         8,000.00              34.99
        10,125.00               5.98
        15,000.00               5.98

(11 rows affected)
```

If you are using the **count(*column_name*)** aggregate function, grouping by a column that contains null values will return a count of zero for the grouping row, since **count(*column_name*)** does not count null values. In most cases, you want to use **count(*)** instead. This example groups and counts on the *price* column from the *titles* table, which contains null values, and shows **count(*)** for comparison:

```
select price, count(price), count(*)
from titles
group by price
```

```
price
------------- ----- -----
        NULL     0     2
        2.99     2     2
        7.00     1     1
        7.99     1     1
       10.95     1     1
       11.95     2     2
       14.99     1     1
       19.99     4     4
       20.00     1     1
       20.95     1     1
       21.59     1     1
       22.95     1     1

(12 rows affected)
```

where Clause and *group by*

You can use a **where** clause in a statement with **group by**. Rows that do not satisfy the conditions in the **where** clause are eliminated before any grouping is done. Here is an example:

```
select type, avg(price)
from titles
where advance > 5000
group by type
```

```
type
-------------    --------
business            2.99
mod_cook            2.99
popular_comp       21.48
psychology         14.30
trad_cook          17.97

(5 rows affected)
```

Only the rows with advances greater than $5,000 are included in the groups that are used to produce the query results. The values are very different when you run the query without the **where** clause.

However, the way that SQL Server handles extra columns in the select list and the **where** clause may seem contradictory. For example:

```
select type, advance, avg(price)
from titles
where advance > 5000
group by type
```

```
type           advance
-------------  ---------  --------
business        5,000.00      2.99
business        5,000.00      2.99
business       10,125.00      2.99
business        5,000.00      2.99
mod_cook            0.00      2.99
mod_cook       15,000.00      2.99
popular_comp    7,000.00     21.48
popular_comp    8,000.00     21.48
popular_comp        NULL     21.48
```

```
psychology      7,000.00     14.30
psychology      2,275.00     14.30
psychology      6,000.00     14.30
psychology      2,000.00     14.30
psychology      4,000.00     14.30
trad_cook       7,000.00     17.97
trad_cook       4,000.00     17.97
trad_cook       8,000.00     17.97

(17 rows affected)
```

It only seems as if the query is ignoring the **where** clause when you look at the results for the *advance* (extended) column. SQL Server still computes the vector aggregate using only those rows that satisfy the **where** clause, but it also displays all rows for any extended columns that you include in the select list. To further restrict these rows from the results, you must use a **having** clause (described later in this chapter).

For more information about how SQL Server handles the **where** clause and **group by**, see the *SQL Server Reference Manual*.

group by and *all*

The keyword **all** in the **group by** clause is a Transact-SQL enhancement to SQL. It is meaningful only if the **select** statement in which it is used also includes a **where** clause.

If you use **all**, the query results will include all the groups produced by the **group by** clause, even if some of the groups don't have any rows that meet the search conditions. Without **all**, a **select** statement that includes **group by** does not show groups for which no rows qualify.

Here is an example:

```
select type, avg(advance)
from titles
where advance > 1000 and advance < 10000
group by type

type
------------  ------------------------
business                    5,000.00
popular_comp                7,500.00
psychology                  4,255.00
trad_cook                   6,333.33

(4 rows affected)
```

```
select type, avg(advance)
from titles
where advance > 1000 and advance < 10000
group by all type
```

```
type
------------  ------------------------
UNDECIDED                         NULL
business                     5,000.00
mod_cook                          NULL
popular_comp                 7,500.00
psychology                   4,255.00
trad_cook                    6,333.33

(6 rows affected)
```

The first statement produces groups only for those books which commanded advances greater than $1,000 but less than $10,000. Since no modern cooking books have an advance within that range, there is no group in the results for the *mod_cook* type.

The second statement produces groups for all types, including modern cooking and "UNDECIDED," even though the modern cooking group doesn't include any rows that meet the qualification specified in the **where** clause. SQL Server returns a NULL result for these rows,

The column that holds the aggregate value (the average advance) is for groups that lack qualifying rows.

Using Aggregates Without *group by*

By definition, scalar aggregates apply to all rows in a table, producing a single value for the whole table for each function. The Transact-SQL extension that allows you include extended columns with vector aggregates, also allows you to include extended columns with scalar aggregates. For example:

```
select pub_id, count(pub_id)
from publishers
```

```
pub_id
---------- ---------
0736               3
0877               3
1389               3

(3 rows affected)
```

SQL Server treats *publishers* as a single group, and the scalar aggregate applies to the (single-group) table. The results display every row of the table for each column you include in the select list in addition to the scalar aggregate.

The **where** clause behaves the same way for scalar aggregates as with vector aggregates. The **where** clause restricts the columns included in the aggregate summary values, but it does not affect the rows that appear in the results for each extended column you specify in the select list. For example:

```
select pub_id, count(pub_id)
from publishers
where pub_id < "1000"

pub_id
-------------- -----------
0736                     2
0877                     2
1389                     2

(3 rows affected)
```

Like the other Transact-SQL extensions to **group by**, this extension to scalar aggregates provides results that may be difficult to comprehend, especially for queries on large tables or queries with multitable joins.

Selecting Groups of Data: The *having* Clause

The **having** clause sets conditions for the **group by** clause similar to the way in which **where** sets conditions for the **select** clause.

having search conditions are identical to **where** search conditions, with one exception: **where** search conditions cannot include aggregates, while **having** search conditions often do. **having** clauses can reference any of the items that appear in the select list. There is a limit of 128 conditions that can be included in a **having** clause.

This statement is an example of a **having** clause with an aggregate function. It groups the rows in the *titles* table by type, but eliminates the groups that include only one book:

```
select type
from titles
group by type
having count(*) > 1
```

```
type
----------------
business
mod_cook
popular_comp
psychology
trad_cook

(5 rows affected)
```

Here is an example of a **having** clause without aggregates. It groups the *titles* table by type and eliminates those types that do not start with the letter "p":

```
select type
from titles
group by type
having type like 'p%'
```

```
type
------------
popular_comp
psychology

(2 rows affected)
```

When more than one condition is included in the **having** clause, they are combined with **and**, **or**, or **not**. For example, to group the *titles* table by publisher, and to include only those publishers with identification numbers greater than 0800, who have paid more than $15,000 in total advances, and whose books average less than $18 in price, the statement is:

```
select pub_id, sum(advance), avg(price)
from titles
group by pub_id
having sum(advance) > 15000
  and avg(price) < 18
  and pub_id > "0800"
```

```
pub_id
------  ---------------- ----------------
0877          41,000.00            15.41

(1 row affected)
```

How the *having*, *group by*, and *where* Clauses Interact

When you include the **having**, **group by**, and **where** clauses in a query, the sequence in which each clause affects the rows in the table is significant when determining the final results:

- The **where** clause excludes rows that do not meet its search conditions.
- The **group by** clause collects the remaining rows into one group for each unique value in the **group by** expression.
- Aggregate functions specified in the select list calculate summary values for each group.
- The **having** clause excludes rows from the final results that do not meet its search conditions.

The following query illustrates the use of **where**, **group by**, and **having** clauses in one **select** statement:

```
select stor_id, title_id, sum(qty)
from salesdetail
where title_id like "PS%"
group by stor_id, title_id
having sum(qty) > 200
```

```
stor_id  title_id
-------  --------  -----------
5023     PS1372            375
5023     PS2091           1845
5023     PS3333           3437
5023     PS7777           2206
6380     PS7777            500
7067     PS3333            345
7067     PS7777            250

(7 rows affected)
```

The **where** clause includes only rows that have a *title_id* beginning with "PS" (psychology books), before **group by** collects the rows by common *stor_id* and *title_id*. The **sum** aggregate calculates the total number of books sold for each group, and then the **having** clause excludes the groups whose totals do not exceed 200 books from the final results.

All of the previous **having** examples adhere to the SQL standards, which specify that columns in a **having** expression must have a single value, and must be in the select list or **group by** clause. However, the Transact-SQL extensions to **having** allow columns or expressions not in the select list and not in the **group by** clause.

The following example uses this extension. It determines the average price for each title type, but it excludes those types that do not have over 10,000 total sales even though the **sum** aggregate does not appear in the results.

```
select type, avg(price)
from titles
group by type
having sum(total_sales) > 10000
```

```
type
------------  ----------
business           13.73
mod_cook           11.49
popular_comp       21.48
trad_cook          15.96

(4 rows affected)
```

The extension behaves as if the column or expression were part of the select list but not part of the displayed results. If you include an unaggregated column with **having**, but it is not part of the select list or the **group by** clause, the query produces results similar to the "extended" column extension described earlier in this chapter. For example:

```
select type, avg(price)
from titles
group by type
having total_sales > 4000
```

```
type
------------  ----------
business           13.73
business           13.73
business           13.73
mod_cook           11.49
popular_comp       21.48
popular_comp       21.48
psychology         13.50
trad_cook          15.96
trad_cook          15.96

(9 rows affected)
```

Unlike an extended column, the *total_sales* column does not appear in the final results, yet the number of displayed rows for each type depends on the *total_sales* for each title. The query indicates that 3 business, 1 mod_cook, 2 popular_comp, 1 psychology, and 2 trad_cook titles exceed 4000 total sales.

As mentioned earlier, the way SQL Server handles extended columns may seem as if the query is ignoring the **where** clause in the final results. To make the **where** conditions affect the results for the extended column, you should repeat the conditions in the **having** clause. For example:

```
select type, advance, avg(price)
from titles
where advance > 5000
group by type
having advance > 5000
type           advance
-------------  ---------  --------
business       10,125.00      2.99
mod_cook       15,000.00      2.99
popular_comp    7,000.00     21.48
popular_comp    8,000.00     21.48
psychology      7,000.00     14.30
psychology      6,000.00     14.30
trad_cook       7,000.00     17.97
trad_cook       8,000.00     17.97

(8 rows affected)
```

Using *having* Without *group by*

A query with a **having** clause should also have a **group by** clause. If it is omitted, all the rows not excluded by the **where** clause are considered to be a single group.

Because no grouping is done between the **where** and **having** clauses, they cannot act independently of each other. **having** acts like **where** because it affects the rows in a single group rather than groups, except the **having** clause can still use aggregates.

The following example uses the **having** clause to exclude from the results those rows in the single group table *titles* whose price does not exceed the average price of all titles, after the **where** clause excludes the titles with advance greater than $4,000 from the computation of the average price:

```
select title_id, advance, price
from titles
where advance < 4000
having price > avg(price)
```

```
title_id        advance    price
-------------  ---------  --------
BU1032          5,000.00     19.99
BU7832          5,000.00     19.99
MC2222              0.00     19.99
PC1035          7,000.00     22.95
PC8888          8,000.00     20.00
PS1372          7,000.00     21.59
PS3333          2,000.00     19.99
TC3218          7,000.00     20.95

(8 rows affected)
```

You can also use the **having** clause with the Transact-SQL extension that allows you to omit the **group by** clause from a query that includes an aggregate in its select list. These scalar aggregate functions calculate values for the table as a single group, not for groups within the table.

In this example, omitting the **group by** clause makes the aggregate function calculate a value for the whole table. The **having** clause excludes rows from the result group—rows of the single table.

```
select pub_id, count(pub_id)
from publishers
having pub_id < "1000"

pub_id
------ ----------------
0736                 3
0877                 3

(2 rows affected)
```

For more information about queries that use **having** and omit **group by**, see the *SQL Server Reference Manual.*

Sorting Query Results: The *order by* Clause

The **order by** clause allows sorting of query results by one or more columns. The maximum number of columns is 16. Each sort can be either ascending (**asc**) or descending (**desc**). If neither is specified, **asc** is assumed. The following query returns results ordered by *pub_id*:

```
select pub_id, type, title_id
from titles
order by pub_id
```

```
pub_id  type            title_id
------  ------------    --------
0736    business        BU2075
0736    psychology      PS2091
0736    psychology      PS2106
0736    psychology      PS3333
0736    psychology      PS7777
0877    UNDECIDED       MC3026
0877    mod_cook        MC2222
0877    mod_cook        MC3021
0877    psychology      PS1372
0877    trad_cook       TC3218
0877    trad_cook       TC4203
0877    trad_cook       TC7777
1389    business        BU1032
1389    business        BU1111
1389    business        BU7832
1389    popular_comp    PC1035
1389    popular_comp    PC8888
1389    popular_comp    PC9999

(18 rows affected)
```

If more than one column is named in the **order by** clause, sorts are nested. The following statement sorts the rows in the *titles* table first by publisher in descending order, then by type (ascending) within each publisher, and finally by title number (also ascending, since **desc** is not specified). Null values are sorted first within any group.

```
select pub_id, type, title_id
from titles
order by pub_id desc, type, title_id
```

```
pub_id    type          title_id
------    ----------    --------
1389      business      BU1032
1389      business      BU1111
1389      business      BU7832
1389      popular_comp  PC1035
1389      popular_comp  PC8888
1389      popular_comp  PC9999
0877      UNDECIDED     MC3026
0877      mod_cook      MC2222
0877      mod_cook      MC3021
0877      psychology    PS1372
```

```
0877      trad_cook      TC3218
0877      trad_cook      TC4203
0877      trad_cook      TC7777
0736      business       BU2075
0736      psychology     PS2091
0736      psychology     PS2106
0736      psychology     PS3333
0736      psychology     PS7777

(18 rows affected)
```

The position number of a column in a select list can be used instead of the column name. Column names and select list numbers can be mixed. Both of the following statements produce the same results as the preceding one.

```
select pub_id, type, title_id
from titles
order by 1 desc, 2, 3

select pub_id, type, title_id
from titles
order by 1 desc, type, 3
```

Most versions of SQL require that **order by** items appear in the select list, but Transact-SQL has no such restriction. You could order the results of the preceding query by *title*, although that column does not appear in the select list.

➤ ***Note***

You cannot use **order by** on *text* or *image* columns.

Subqueries, aggregates, variables and constant expressions are not allowed in the **order by** list.

The effects of an **order by** clause on mixed-case data depend on the sort order installed on your SQL Server. The basic choices are binary, dictionary order, and case-insensitive. The system procedure **sp_helpsort** displays the sort order for your Server. See **order by** in the *SQL Server Reference Manual* for full information on sort orders.

order by and *group by*

You can use an **order by** clause when you want to order the results of a **group by** in a particular way.

Put the **order by** clause after the **group by** clause. For example, to find the average price of each type of book and order the results by average price, the statement is:

```
select type, avg(price)
from titles
group by type
order by avg(price)
```

```
type
---------- ------------
UNDECIDED           NULL
mod_cook          11.49
psychology        13.50
business          13.73
trad_cook         15.96
popular_comp      21.48

(6 rows affected)
```

Summarizing Groups of Data: The *compute* Clause

The **compute** clause is a Transact-SQL extension of SQL. Use it with row aggregates to produce reports that summarize values whenever the value in a specified column changes. Such reports, usually produced by a report generator, are called control-break reports, since summary values appear in the report under the control of the groupings ("breaks") you specify in the **compute** clause.

These summary values appear as additional rows in the query results, unlike the aggregate results of a **group by** clause, which appear as new columns.

A **compute** clause allows you to see detail and summary rows with one **select** statement. You can calculate summary values for subgroups and you can calculate more than one row aggregate for the same group.

The general syntax for **compute** is:

```
compute row_aggregate(column_name)
     [, row_aggregate(column_name)]...
     [by column_name [, column_name]...]
```

The row aggregates you can use with **compute** are **sum**, **avg**, **min**, **max**, and **count**. **sum** and **avg** are used with numeric columns only. Unlike the **order by** clause, you cannot use the positional number of a column from the select list instead of the column name.

➤ ***Note***

You cannot use *text* or *image* columns in a **compute** clause.

Following are two queries and their results. The first one uses **group by** and aggregates. The second uses **compute** and row aggregates. Notice the difference in the displays.

```
select type, sum(price), sum(advance)
from titles
group by type
```

```
type
------------   -------    ----------
UNDECIDED         NULL         NULL
business         54.92    25,125.00
mod_cook         22.98    15,000.00
popular_comp     42.95    15,000.00
psychology       67.52    21,275.00
trad_cook        47.89    19,000.00

(6 rows affected)
```

```
select type, price, advance
from titles
order by type
compute sum(price), sum(advance) by type
```

```
type             price            advance
-------------    ------------     ----------
UNDECIDED               NULL          NULL
                sum               sum
                ------------      ----------
                        NULL           NULL

type             price            advance
-------------    ------------     ----------
business                 2.99      10,125.00
business                11.95       5,000.00
business                19.99       5,000.00
business                19.99       5,000.00
                 sum              sum
                 ------------     ---------
                        54.92      25,125.00
```

```
type            price           advance
-------------   ------------    ---------
mod_cook               2.99     15,000.00
mod_cook              19.99          0.00
                sum             sum
                ------------    ---------
                      22.98     15,000.00

type            price           advance
-------------   ------------    ---------
popular_comp            NULL         NULL
popular_comp          20.00       8,000.00
popular_comp          22.95       7,000.00
                sum             sum
                ------------    ---------
                      42.95     15,000.00

type            price           advance
-------------   ------------    ---------
psychology             7.00       6,000.00
psychology             7.99       4,000.00
psychology            10.95       2,275.00
psychology            19.99       2,000.00
psychology            21.59       7,000.00
                sum             sum
                ------------    ---------
                      67.52     21,275.00

type            price           advance
-------------   ------------    ---------
trad_cook             11.95       4,000.00
trad_cook             14.99       8,000.00
trad_cook             20.95       7,000.00
                sum             sum
                ------------    ---------
                      47.89     19,000.00

(24 rows affected)
```

The summary values are treated as new rows, which is why SQL Server's message says "24 rows affected."

Row Aggregates and *compute*

The row aggregates used with compute are listed in the following table:

Table 3-2: Row aggregates used with compute statement

Row Aggregate	Result
sum	Total of the values in the expression
avg	Average of the values in the expression
max	Highest value in the expression
min	Lowest value in the expression
count	Number of selected rows

These row aggregates are the same aggregates that can be used with **group by**, except that there is no **row aggregate function** that is the equivalent of **count(*)**. To find the summary information produced by **group by** and **count(*)**, use a **compute** clause without **by**.

Rules for *compute* Clauses

- The **distinct** keyword is not allowed with the row aggregates.
- The columns in a **compute** clause must appear in the statement's select list.
- You cannot use **select into** in the same statement as a **compute** clause because statements that include **compute** do not generate normal rows.
- If you use **compute** with the **by** keyword, you must also use an **order by** clause. The columns listed after **by** must be identical to, or a subset of, those listed after **order by**, and must be in the same left-to-right order, start with the same expression, and not skip any expressions.

  For example, say the **order by** clause is:

  ```
  order by a, b, c
  ```

  The **compute** clause can be any or all of these:

  ```
  compute row_aggregate (column_name) by a, b, c
  compute row_aggregate (column_name) by a, b
  compute row_aggregate (column_name) by a
  ```

  The **compute** clause cannot be any of these:

  ```
  compute row_aggregate (column_name) by b, c
  ```

```
compute row_aggregate (column_name) by a, c
compute row_aggregate (column_name) by c
```

You must use a column name or an expression in the **order by** clause; you cannot sort by a column heading.

- The **compute** keyword can be used without **by** to generate grand totals, grand counts, and so on. **order by** is optional if you use the **compute** keyword without **by**. **compute** without **by** is discussed later.

Specifying More Than One Column After *compute*

Listing more than one column after the keyword **by** breaks a group into subgroups and applies the specified row aggregate at each level of grouping. For example, here is a query that finds the sum of the prices of psychology books from each publisher:

```
select type, pub_id, price
from titles
where type = "psychology"
order by type, pub_id, price
compute sum(price) by type, pub_id
type        pub_id  price
----------- ------- -------------
psychology    0736           7.00
psychology    0736           7.99
psychology    0736          10.95
psychology    0736          19.99
                    sum
                    ------------
                           45.93
type        pub_id  price
----------- ------- -------------
psychology    0877          21.59
                    sum
                    ------------
                           21.59

(7 rows affected)
```

Using More Than One *compute* Clause

You can use different aggregates in the same statement by including more than one **compute** clause. The following is a query similar to the preceding one. It finds the sum of the prices of all psychology books, as well as the sum of the prices of psychology books by publisher:

```
select type, pub_id, price
from titles
where type = "psychology"
order by type, pub_id, price
compute sum(price) by type, pub_id
compute sum(price) by type
```

```
type        pub_id  price
----------- ------- --------------
psychology    0736           7.00
psychology    0736           7.99
psychology    0736          10.95
psychology    0736          19.99
                    sum
                    -------------
                           45.93
type        pub_id  price
 ---------- ------- --------------
 psychology    0877          21.59
                    sum
                    -------------
                           21.59
                    sum
                    -------------
                           67.52

(8 rows affected)
```

Applying an Aggregate to More Than One Column

One **compute** clause can apply the same aggregate to several columns. This query finds the sum of the prices and advances for each type of cookbook:

```
select type, price, advance
from titles
where type like "%cook"
order by type
compute sum(price), sum(advance) by type
```

```
type      price            advance
--------- ---------------- ---------------
mod_cook             2.99       15,000.00
mod_cook            19.99            0.00
          sum              sum
          ---------------- ---------------
                    22.98       15,000.00
type      price            advance
--------- ---------------- ---------------
trad_cook           11.95        4,000.00
trad_cook           14.99        8,000.00
trad_cook           20.95        7,000.00
          sum              sum
          ---------------- ---------------
                    47.89       19,000.00

(7 rows affected)
```

Remember, the columns to which the aggregates apply must also be in the select list.

Using Different Aggregates in the Same *compute* Clause

You can use different aggregates in the same **compute** clause:

```
select type, pub_id, price
from titles
where type like "%cook"
order by type, pub_id
compute sum(price), max(pub_id) by type
```

```
type        pub_id  price
----------- ------- --------------
mod_cook    0877             2.99
mod_cook    0877            19.99
                    sum
                    --------------
                            22.98
            max
            -----
            0877

type        pub_id  price
----------- ------- --------------
trad_cook   0877            11.95
```

```
trad_cook   0877            14.99
trad_cook   0877            20.95
                       sum
                       --------------
                               47.89
            max
            -----
            0877

(7 rows affected)
```

Grand Values: *compute* Without *by*

The **compute** keyword can be used without **by** to generate grand totals, grand counts, and so on.

This statement finds the grand total of the prices and advances of all types of books over $20:

```
select type, price, advance
from titles
where price > $20
compute sum(price), sum(advance)
```

```
type         price            advance
------------ ---------------  -------------
popular_comp            22.95      7,000.00
psychology              21.59      7,000.00
trad_cook               20.95      7,000.00
             sum                   sum
             ================ ==============
                       65.49      21,000.00

(4 rows affected)
```

You can use a **compute** with **by** and a **compute** without **by** in the same query. The following query finds the sum of prices and advances by type, and then computes the grand total of prices and advances for all types of books.

```
select type, price, advance
from titles
where type like "%cook"
order by type
compute sum(price), sum(advance) by type
compute sum(price), sum(advance)
```

```
type         price                advance
-----------  -----------------    ------------
mod_ cook                 2.99       15,000.00
mod_cook                 19.99            0.00
             sum                  sum
             -----------------    ------------
                          22.98       15,000.00

type         price                advance
-----------  -----------------    ------------
trad_cook                11.95        4,000.00
trad_cook                14.99        8,000.00
trad_cook                20.95        7,000.00
             sum                  sum
             -----------------    ------------
                          47.89       19,000.00
             sum                  sum
             =================    ============
                          70.87       34,000.00

(8 rows affected)
```

Combining Queries: The *union* Operator

The Transact-SQL **union** operator allows you to manipulate the results of two or more queries by combining the results of each query into a single result set. The syntax is as follows:

```
query1
   [union [all] queryN] ...
   [order by clause]
   [compute clause]
```

where *query1* is:

```
   select select_list
   [into clause]
   [from clause]
   [where clause]
   [group by clause]
   [having clause]
```

and *queryN* is:

```
   select select_list
   [from clause]
   [where clause]
   [group by clause]
   [having clause]
```

For example, suppose you have the following two tables containing the data shown:

Table T1	
a	**b**
char(4)	**int**
abc	1
def	2
ghi	3

Table T2	
a	**b**
char(4)	**int**
ghi	3
jkl	4
mno	5

The following query creates a **union** between the two tables:

```
select * from T1
union
select * from T2
```

The result set is as follows:

a	**b**
char(4)	**int**
abc	1
def	2
ghi	3
jkl	4
mno	5

Notice that by default, the **union** operator removes duplicate rows from the result set. If you use the **all** option, all rows are included in the results; duplicates are not removed. Notice also that the columns in the result set have the same names as the columns in *T1*. Any number of **union** operators may appear in a Transact-SQL statement. For example:

```
x union y union z
```

By default, SQL Server evaluates a statement containing **union** operators from left to right. Parentheses may be used to specify the order of evaluation. For example, the expressions:

```
x union all (y union z)
```

and:

```
(x union all y) union z
```

are not equivalent. In the first example, duplicates are eliminated in the union between *y* and *z*. Then, in the union between that set and *x*,

duplicates are **not** eliminated. In the second example, duplicates are included in the union between *x* and *y*, but are then eliminated in the subsequent union with *z*; **all** has no effect in the final result of this statement.

Guidelines for *union* Queries

The following are guidelines to observe when you use **union** statements:

- All select lists in the **union** statement must have the same number of expressions (such as column names, arithmetic expressions, and aggregate functions). The following statement is invalid because the first select list is longer than the second:

  ```
  select stor_id, date, ord_num from stores
  union
  select stor_id, ord_num from stores_east
  ```

- Corresponding columns in all tables, or any subset of columns used in the individual queries, must be of the same datatype, or an implicit data conversion must be possible between the two datatypes, or an explicit conversion should be supplied. For example, a **union** is not possible between a column of the *char* datatype and one of the *int* datatype, unless an explicit conversion is supplied. However, a **union** is possible between a column of the *money* datatype and one of the *int* datatype. See **union** and "Datatype Conversion Functions" in the *SQL Server Reference Manual* for more information about comparing datatypes in a **union** statement.
- Corresponding columns in the individual queries of a **union** statement must occur in the same order, because **union** compares the columns one-to-one in the order given in the individual queries. For example, suppose you have the following tables:

Table T3

a	b	c
int	char(4)	char(4)
1	abc	jkl
2	def	mno
3	ghi	pqr

Table T4

a	b
char(4)	int
abc	1
def	2
ghi	3

The following query:

```
select a, b from T3
union
select b, a from T4
```

produces this result set:

a	b
1	abc
2	def
3	ghi

The following query:

```
select a, b from T3
union
select a, b from T4
```

results in an error message, because the datatypes of corresponding columns are not compatible. When different (but compatible) datatypes such as *float* and *int* are combined in a **union** statement, they are converted to the datatype with the most precision.

- The column names in the table resulting from a **union** are taken from the **first** individual query in the **union** statement. Therefore, if you want to define a new column heading for the result set, you must do so in the first query. In addition, if you want to refer to a column in the result set by a new name, for example in an **order by** statement, it must be referred to in that way in the first **select** statement.

  The following query is correct:

  ```
  select Cities = city from stores
  union
  select city from authors
  order by Cities
  ```

Using *union* with Other Transact-SQL Commands

Following are some guidelines to follow when you use **union** statements with other Transact-SQL commands:

- The first query in the **union** statement may contain an **into** clause that creates a table to hold the final result set. For example, the following statement creates a table called *results* that contains the union of tables *publishers*, *stores*, and *salesdetail*:

```
select pub_id, pub_name, city into results from
     publishers
union
select stor_id, stor_name, city from stores
union
select stor_id, title_id, ord_num from salesdetail
```

The **into** clause can be used only in the first query; if it appears anywhere else you get an error message.

- **order by** and **compute** clauses can be used only at the end of the **union** statement, to define the order of the final results or to compute summary values. They cannot be used within the individual queries that make up the **union** statement.
- **group by** and **having** clauses can be used within individual queries only; they cannot be used to affect the final result set.
- The **union** operator can also be used within an **insert** statement. For example:

  ```
  insert into tour
      select city, state from stores
      union
      select city, state from authors
  ```

- The **union** operator cannot be used within a **create view** statement.
- The **for browse** clause cannot be used in statements involving the **union** operator.

4 Joins: Retrieving Data from Several Tables

This chapter begins the discussion of retrieval operations that involve data from two or more tables. These tables can be located in the same database or in different databases. Up to this point, the discussion has been restricted to examples of retrieving data from a single table.

The multitable operation discussed in this chapter is the join. Subqueries, which also can involve two or more tables, are covered in Chapter 5, "Subqueries: Using Queries Within Other Queries." Many joins can be stated as subqueries.

This chapter discusses:

- A general overview of join operations
- How to join tables in a query
- How SQL Server processes joins
- How null values affect joins
- How to determine which columns to join

What Are Joins?

Joining two or more tables is a process of comparing the data in specified fields and using the comparison results to form a new table from the rows that qualify. A **join** statement specifies a column from each table, compares the values in those columns row by row, and combines rows with qualifying values into new rows. The comparison is usually for equality—values that match exactly—but other types of joins can be specified. If a join is to have meaningful results, the columns being compared must have similar values—values that are comparable because they have the same or similar datatypes.

The join operation has a jargon all its own. The word "join" is used both as a verb and as a noun, referring to the operation itself, to the query, or to its results.

There are several varieties of joins—equijoins, natural joins, outer joins, and so on.

The most common variety is the join based on equality. Here is an example of a join that finds the names of authors and publishers located in the same city:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city = publishers.city
au_fname      au_lname      pub_name
--------      --------      --------------------
Cheryl        Carson        Algodata Infosystems
Abraham       Bennet        Algodata Infosystems

(2 rows affected)
```

Since the query draws on information contained in two separate tables, *publishers* and *authors*, a join is required to retrieve the requested information.

Joins and the Relational Model

The join operation is the hallmark of the relational model of database management. More than any other feature, the join distinguishes relational database management systems from other types of database management systems.

In structured database management systems, often known as network and hierarchical systems, relationships between data values are predefined. Once a database has been set up, it is difficult to make queries about unanticipated relationships among the data.

In a relational database management system, on the other hand, relationships among data values are left unstated in the definition of a database. They become explicit when the data is manipulated—when you **query** the database, not when you create it. You can ask any question that comes to mind about the data stored in the database, regardless of what was intended when the database was set up.

According to the rules of good database design, called **normalization** rules, each table should describe one kind of entity—a person, place, event, or thing. That is why, when you want to compare information about two or more kinds of entities, you need the join operation. Relationships among data stored in different tables are discovered by joining them.

A corollary of this rule is that the join operation gives you unlimited flexibility in adding new kinds of data to your database. You can always create a new table that contains data about a different kind of entity. If the new table has a field with values similar to those in some field of an existing table or tables, it can be linked to those other tables by joining.

Joining Tables in Queries

A join statement, like a selection statement, starts with the keyword **select**. The columns named after the **select** keyword are the columns to be included in the query results, in their desired order. The previous example specified the columns that contained the authors' and publishers' names.

The columns *pub_name*, *au_lname*, and *au_fname* did not have to be qualified by a table name, since there is no ambiguity about the table to which they belong. But the *city* column used for the join comparison did have to be qualified, since there are columns of that name in both the *publishers* and *authors* tables. Though in this example neither of the *city* columns is printed in the results, SQL Server needs the table name in order to perform the comparison.

As in a select statement, you can specify that all the columns of the tables involved in the query be included in the results with the abbreviation "*". For example, to include all the columns in *publishers* and *authors* in the preceding join query, the statement is:

```
select *
from authors, publishers
where authors.city = publishers.city
```

```
au_id       au_lname au_fname phone        address
city       state postalcode contract pub_id pub_name
city       state
----------- -------- -------- ------------ --------------------
---------- ----- ---------- -------- ------ --------------------
---------- -----
238-95-7766 Carson   Cheryl   415 548-7723 589 Darwin Ln.
Berkeley   CA    94705      1        1389   Algodata Infosystems
Berkeley   CA
409-56-7008 Bennet   Abraham  415 658-9932 223 Bateman St
Berkeley   CA    94705      1        1389   Algodata Infosystems
Berkeley   CA

(2 rows affected)
```

The display shows a total of two rows with thirteen columns each. Because of the length of the rows, each takes up multiple horizontal lines in this display. Whenever "*" is used, the columns in the results are displayed in their order in the **create** statement for the table.

The select list and the results of a join need not include columns from both of the tables being joined. For example, to find the names of the authors that live in the same city as one of the publishers, your query need not include any columns from *publishers*:

```
select au_lname, au_fname
from authors, publishers
where authors.city = publishers.city
```

Remember, just as in any **select** statement, column names in the select list and table names in the **from** clause must be separated by commas.

The *from* Clause

The **from** clause of a join statement names all the tables or views involved in the join. This is actually the clause that indicates to SQL Server that a join is desired. You can list the tables or views in any order. The order of tables affects the results displayed only when you use **select *** to specify the select list.

More than two tables or views can be named in the **from** clause. At most, a query can reference 16 tables. This maximum includes:

- Tables (or views on tables) listed in the **from** clause
- Each instance of multiple references to the same table (self-joins)
- Tables referenced in subqueries
- Base tables referenced by the views listed in the **from** clause

Joins involving more than two tables or views are discussed later in this chapter under "Joining More Than Two Tables" on page 4-13.

As explained in Chapter 2, "Queries: Selecting Data from a Table," table or view names can be qualified by the names of the owner and database, and can be given correlation names for convenience.

Views can be joined in exactly the same way as tables and used wherever tables are used. Chapter 9 discusses views; this chapter uses only tables in its examples.

The *where* Clause

The **where** clause specifies the connection between the tables named in the **from** clause, restricting the rows to be included in the results. It gives the names of the columns to be joined, qualified by table names if necessary, and the join operator—often equality, sometimes "greater than" or "less than." For details of **where** clause syntax, see Chapter 2 of this guide, and the "where Clause" section of the *SQL Server Reference Manual*.

➤ ***Note***

You will get unexpected results if you leave off the **where** clause of a join. Without a **where** clause, any of the join queries discussed so far will produce 27 rows instead of 2. The next section explains why that happens.

Joins that match columns on the basis of equality are called **equijoins**. A more precise definition of an **equijoin** is given later in this chapter, along with examples of joins not based on equality.

The join operators that determine the basis on which columns will be matched are the relational operators:

Table 4-1: Join operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
!=	Not equal to
!>	Less than or equal to
!<	Greater than or equal to

Joins that use the relational operators are collectively called **theta joins**. Another set of join operators is used for **outer joins,** also discussed in detail later in this chapter. The outer join operators are:

Table 4-2: Outer join operators

Operator	Action
*=	Include in the results all the rows from the first table, not just the ones where the joined columns match.
=*	Include in the results all the rows from the second table, not just the ones where the joined columns match.

Columns being joined need not have the same name, although they often do. Also, they need not be the same datatype (see Chapter 7).

However, if the datatypes are not identical they must be **compatible**—types that SQL Server automatically converts. For example, SQL Server automatically converts among any of the numeric type columns—*int*, *smallint*, *tinyint*, *decimal*, or *float*, and among any of the character type and date columns—*char*, *varchar*,

nchar, *nvarchar*, and *datetime*. For details on datatype conversion, see Chapter 10, "Using the Built-In Functions in Queries," and the "Datatype Conversion Functions" section of the *SQL Server Reference Manual*.

➤ ***Note***

Tables cannot be joined on *text* or *image* columns. You can, however, compare the lengths of text columns from two tables with a **where** clause such as:

```
where datalength(textab_1.textcol) >
datalength(textab_2.textcol)
```

The **where** clause of a join statement can include other conditions in addition to the one that links columns from different tables. In other words, you can include a join operation and a **select** operation in the same SQL statement. An example is given later in this chapter.

How Joins Are Processed

Knowing how joins are processed helps to understand them—and to figure out why, when you incorrectly state a join, you sometimes get unexpected results. This section describes the processing of joins in conceptual terms. SQL Server's actual procedure is more sophisticated.

Conceptually speaking, the first step in processing a join is to form the **Cartesian product** of the tables—all the possible combinations of the rows from each of the tables. The number of rows in a Cartesian product of two tables is equal to the number of rows in the first table times the number of rows in the second table.

The Cartesian product of the *authors* table and the *publishers* table is 69 (23 authors multiplied by 3 publishers). You can have a look at a Cartesian product with any query that includes columns from more than one table in the select list, more than one table in the **from** clause, and no **where** clause. For example, if you leave the **where** clause off the join used in previous examples, SQL Server combines each of the 23 authors with each of the 3 publishers, and returns all 69 rows.

This Cartesian product does not contain any particularly useful information. In fact, it is downright misleading, since it seems to imply that every author in the database has a relationship with every publisher in the database—which is not true at all.

That is why a join must include a **where** clause, which specifies the columns to be matched and the basis on which to match them. It may also include other restrictions. Once the Cartesian product has been formed, the rows that do not satisfy the join are eliminated using the conditions in the **where** clause.

The **where** clause included in the previous example eliminates from the results all the rows in which the author's city is not the same as the publisher's city.

Equijoins and Natural Joins

An **equijoin** is a join in which the values in the columns being joined are compared for equality, and all the columns in the tables being joined are included in the results.

The earlier query:

```
select *
from authors, publishers
where authors.city = publishers.city
```

is an example of an equijoin. In the results of that statement, the *city* column appears twice. By definition, the results of an equijoin contain two identical columns. Since there is usually no point in repeating the same information, one of these columns can be eliminated by restating the query. The result is called a **natural join**.

The query that results in the natural join of *publishers* and *authors* on the *city* column is:

```
select publishers.pub_id, publishers.pub_name,
  publishers.state, authors.*
from publishers, authors
where publishers.city = authors.city
```

The column *publishers.city* does not appear in the results.

Joins with Additional Conditions

The **where** clause of a join query can include selection criteria as well as specifying the join condition. For example, to retrieve the names and publishers of all the books for which advances greater than $7,500 were paid, the statement is:

```
select title, pub_name, advance
from titles, publishers
where titles.pub_id = publishers.pub_id
and advance > $7500
```

```
title                          pub_name              advance
-----------------------------  --------------------  ---------
You Can Combat Computer Stress!  New Age Books        10,125.00
The Gourmet Microwave            Binnet & Hardley     15,000.00
Secrets of Silicon Valley        Algodata Infosystems  8,000.00
Sushi, Anyone?                   Binnet & Hardley      8,000.00

(4 rows affected)
```

Notice that the columns being joined do not need to appear in the select list and therefore do not show up in the results.

As many selection criteria as desired can be included in a join statement. The order of the selection criteria and the join condition is not important.

Joins Not Based on Equality

The condition for joining the values in two columns need not be equality. Any of the other comparison operators can be used: not equal (!=), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=). Transact-SQL also provides the operators !> and !<, which are equivalent to <= and >=, respectively.

This example of a greater-than join finds New Age authors who live in states that come after New Age Books' state, Massachusetts, in alphabetical order.

```
select pub_name, publishers.state,
   au_lname, au_fname, authors.state
from publishers, authors
where authors.state > publishers.state
and pub_name = "New Age Books"
```

```
pub_name         state   au_lname       au_fname         state
-------------    ------  -------------- -----------      -----
New Age Books    MA      Greene         Morningstar      TN
New Age Books    MA      Blotchet-Halls Reginald         OR
New Age Books    MA      del Castillo   Innes            MI
New Age Books    MA      Panteley       Sylvia           MD
New Age Books    MA      Ringer         Anne             UT
New Age Books    MA      Ringer         Albert           UT

(6 rows affected)
```

The following example uses a ">=" join and a "<" join to look up the correct *royalty* from the *roysched* table, based on the book's total sales.

```
select t.title_id, t.total_sales, r.royalty
from titles t, roysched r
where t.title_id = r.title_id
  and t.total_sales >= r.lorange and
  t.total_sales < r.hirange
```

```
title_id       total_sales      royalty
--------       -----------      -------
BU1032               4095           10
BU1111               3876           10
BU2075               1872           24
BU7832               4095           10
MC2222               2032           12
MC3021              22246           24
PC1035               8780           16
PC8888               4095           10
PS1372                375           10
PS2091               2045           12
PS2106                111           10
PS3333               4072           10
PS7777               3336           10
TC3218                375           10
TC4203              15096           14
TC7777               4095           10

(16 rows affected)
```

Self-Joins and Correlation Names

You can compare values within a column of a table with the **self-join**. For example, you can use a self-join to find out which authors in Oakland, California live in the same postal code area.

Since this query involves a join of the *authors* table with itself, the *authors* table appears in two roles. To distinguish these roles, you can temporarily and arbitrarily give the *authors* table two different correlation names—such as *au1* and *au2*—in the **from** clause. These correlation names are used to qualify the column names in the rest of the query. The self-join statement looks like this:

```
select au1.au_fname, au1.au_lname,
au2.au_fname, au2.au_lname
from authors au1, authors au2
where au1.city = "Oakland" and au2.city = "Oakland"
and au1.state = "CA" and au2.state = "CA"
and au1.postalcode = au2.postalcode
```

```
au_fname     au_lname    au_fname    au_lname
---------    ----------- --------    --------
Marjorie     Green       Marjorie    Green
Dick         Straight    Dick        Straight
Dick         Straight    Dirk        Stringer
Dick         Straight    Livia       Karsen
Dirk         Stringer    Dick        Straight
Dirk         Stringer    Dirk        Stringer
Dirk         Stringer    Livia       Karsen
Stearns      MacFeather  Stearns     MacFeather
Livia        Karsen      Dick        Straight
Livia        Karsen      Dirk        Stringer
Livia        Karsen      Livia       Karsen

(11 rows affected)
```

To eliminate the rows in the results where the authors match themselves, and to eliminate rows that are identical except that the order of the authors is reversed, you can make this addition to the self-join query:

```
select au1.au_fname, au1.au_lname,
au2.au_fname, au2.au_lname
from authors au1, authors au2
where au1.city = "Oakland" and au2.city = "Oakland"
and au1.state = "CA" and au2.state = "CA"
and au1.postalcode = au2.postalcode
and au1.au_id < au2.au_id
```

```
au_fname    au_lname     au_fname    au_lname
---------   -----------  ---------   ---------
Dick        Straight     Dirk        Stringer
Dick        Straight     Livia       Karsen
Dirk        Stringer     Livia       Karsen

(3 rows affected)
```

It is now clear that Dick Straight, Dirk Stringer, and Livia Karsen all have the same postal code.

The Not-Equal Join

The not-equal join is particularly useful in restricting the rows returned by a self-join. For example, a not-equal join and a self-join are used to find the categories in which there are two or more inexpensive (less than $15) books of different prices:

```
select distinct t1.type, t1.price
from titles t1, titles t2
where t1.price <$15 and t2.price <$15
and t1.type = t2.type
and t1.price != t2.price
```

```
type        price
----------  -----
business     2.99
business    11.95
psychology   7.00
psychology   7.99
psychology  10.95
trad_cook   11.95
trad_cook   14.99

(7 rows affected)
```

➤ ***Note***

The expression “not column_name = column_name” is equivalent to “column_name != column_name.”

The following example uses a not-equal join, combined with a self-join. It finds all the rows in the *titleauthor* table where there are two or more rows with the same *title_id*, but different *au_id* numbers that is, books which have more than one author.

```
select distinct t1.au_id, t1.title_id
from titleauthor t1, titleauthor t2
where t1.title_id = t2.title_id
and t1.au_id != t2.au_id
order by t1.title_id
```

```
au_id           title_id
-----------     --------
213-46-8915     BU1032
409-56-7008     BU1032
267-41-2394     BU1111
724-80-9391     BU1111
722-51-5454     MC3021
899-46-2035     MC3021
427-17-2319     PC8888
846-92-7186     PC8888
724-80-9391     PS1372
756-30-7391     PS1372
899-46-2035     PS2091
998-72-3567     PS2091
267-41-2394     TC7777
472-27-2349     TC7777
672-71-3249     TC7777

(15 rows affected)
```

Not-Equal Joins and Subqueries

Sometimes a not-equal join query is not sufficiently restrictive and needs to be replaced by a subquery. For example, suppose you wanted to list the names of authors who live in a city where no publisher is located. For the sake of clarity, let's also restrict this query to authors whose last names begin with "A", "B", or "C". A not-equal join query might be:

```
select distinct au_lname, authors.city
from publishers, authors
where au_lname like "[ABC]%"
and publishers.city != authors.city
```

But here are the results—not an answer to the question that was asked!

```
au_lname            city
----------------    ------------
Bennet              Berkeley
Carson              Berkeley
Blotchet-Halls      Corvallis

(3 rows affected)
```

The system interprets this version of the SQL statement to mean: "find the names of authors who live in a city where **some** publisher is not located." All the excluded authors qualify, including the

authors who live in Berkeley, home of the publisher Algodata Infosystems.

In this case, the way that the system handles joins (first finding every eligible combination before evaluating other conditions) causes this query to return undesirable results. In cases like this, you must use a subquery to get the results you want. A subquery can eliminate the noneligible rows first, and then perform the remaining restrictions.

Here is the correct statement:

```
select distinct au_lname, city
from authors
where au_lname like "[ABC]%"
and city not in
(select city from publishers
where authors.city = publishers.city)
```

Now the results are what we want:

```
au_lname             city
-------------        ------------
Blotchet-Halls       Corvallis

(1 row affected)
```

Subqueries are covered in greater detail in Chapter 5.

Joining More Than Two Tables

The *titleauthor* table of *pubs2* offers a good example of a situation in which joining more than two tables is helpful. To find the titles of all the books of a particular type and the names of their authors, the query is:

```
select au_lname, au_fname, title
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
and titles.type = "trad_cook"
```

```
au_lname       au_fname     title
-------------- -----------  ------------------------
Panteley       Sylvia       Onions, Leeks, and Garlic: Cooking
                            Secrets of the Mediterranean
Blotchet-Halls Reginald     Fifty Years in Buckingham Palace
                            Kitchens
O'Leary        Michael      Sushi, Anyone?
Gringlesby     Burt         Sushi, Anyone?
Yokomoto       Akiko        Sushi, Anyone?

(5 rows affected)
```

Notice that one of the tables in the **from** clause, *titleauthor*, does not contribute any columns to the results. Nor do any of the columns that are joined—*au_id* and *title_id*—appear in the results. Nonetheless, this join is possible only by using *titleauthor* as an intermediate table.

You can also join more than two pairs of columns in the same statement. For example, here's a query that shows the *title_id*, its total sales and the range in which they fall, and the resulting royalty.

```
select titles.title_id, total_sales, lorange,
hirange, royalty
from titles, roysched
where titles.title_id = roysched.title_id
  and total_sales >= lorange and total_sales <
hirange
```

```
title_id    total_sales   lorange hirange  royalty
--------    -----------   ------- -------  -------
 BU1032            4095         0    5000       10
 BU1111            3876         0    4000       10
 BU2075           18722     14001   50000       24
 BU7832            4095         0    5000       10
 MC2222            2032      2001    4000       12
 MC3021            2224     12001   50000       24
 PC1035            8780      4001   10000       16
 PC8888            4095         0    5000       10
 PS1372             375         0   10000       10
 PS2091            2045      1001    5000       12
 PS2106             111         0    2000       10
 PS3333            4072         0    5000       10
 PS7777            3336         0    5000       10
 TC3218             375         0    2000       10
 TC4203           15096      8001   16000       14
 TC7777            4095         0    5000       10

(16 rows affected)
```

When there is more than one join operator in the same statement, either to join more than two tables or to join more than two pairs of columns, the "join expressions" are almost always connected with **and**, as in the earlier examples. However, it is also legal to connect them with **or**.

Outer Joins

In the joins we have discussed so far, only matching rows, that is, rows with values in the specified columns that satisfy the join condition, are included in the results. In a sense, these join operations eliminate the information contained in the rows that do not match.

Sometimes it is desirable to retain that information by including non-matching rows in the results of a join. On such occasions, the outer join is the operation of choice. Transact-SQL is one of a few versions of SQL that supports the outer join.

The outer join operators that Transact-SQL provides are:

Table 4-3: Summary of outer join operators

Operator	Action
*=	Include all rows from the first-named table
=*	Include all rows from the second-named table

Recall that the query for authors who live in the same city as a publisher returns two names: Abraham Bennett and Cheryl Carson. To include all the authors in the results, regardless of whether a publisher is located in the same city, use an outer join. Here's what the query and the results of the outer join look like:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city *= publishers.city
```

```
au_fname     au_lname         pub_name
---------    --------------   ---------------
Johnson      White            NULL
Marjorie     Green            NULL
Cheryl       Carson           Algodata Infosystems
Michael      O'Leary          NULL
Dick         Straight         NULL
Meander      Smith            NULL
Abraham      Bennet           Algodata Infosystems
Ann          Dull             NULL
Burt         Gringlesby       NULL
Chastity     Locksley         NULL
```

```
Morningstar  Greene           NULL
Reginald     Blotche-Halls    NULL
Akiko        Yokomoto         NULL
Innes        del Castillo     NULL
Michel       DeFrance         NULL
Dirk         Stringer         NULL
Stearns      MacFeather       NULL
Livia        Karsen           NULL
Sylvia       Panteley         NULL
Sheryl       Hunter           NULL
Heather      McBadden         NULL
Anne         Ringer           NULL
Albert       Ringer           NULL

(23 rows affected)
```

The comparison operator "*=" distinguishes the outer join from an ordinary join. This "left" outer join tells SQL Server to include all the rows in the *authors* table in the results, whether or not there is a match on the *city* column in the *publishers* table. Notice that in the results, there is no matching data for most of the authors listed, so these rows contain NULL in the *pub_name* column.

➤ ***Note***

Since *bit* columns do not permit null values, a value of "0" appears in an outer join when there is no match for a bit column that is in the inner table.

The "right" outer join is specified with the comparison operator "=*", which indicates that all the rows in the second table are to be included in the results, regardless of whether there is matching data in the first table.

Substituting this operator in the outer join query shown earlier gives this result:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city =* publishers.city
```

```
au_fname     au_lname    pub_name
---------    ---------   ---------------
NULL         NULL        New Age Books
NULL         NULL        Binnet & Hardley
Cheryl       Carson      Algodata Infosystems
Abraham      Bennet      Algodata Infosystems

(4 rows affected)
```

You can further restrict an outer join by comparing it to a constant. This means that you can zoom in on precisely the value or values you really want to see, and use the outer join to list the rows that didn't make the cut. Let's look at the equijoin first, and then compare it to the outer join. For example, if you wanted to find out which titles had a sale of more than 500 copies from any store, use this query:

```
select distinct salesdetail.stor_id, title
from titles, salesdetail
where qty > 500
and salesdetail.title_id = titles.title_id
```

```
stor_id    title
-------    ----------------------------------------
5023       Sushi, Anyone?
5023       Is Anger the Enemy?
5023       The Gourmet Microwave
5023       But Is It User Friendly?
5023       Secrets of Silicon Valley
5023       Straight Talk About Computers
5023       You Can Combat Computer Stress!
5023       Silicon Valley Gastronomic Treats
5023       Emotional Security: A New Algorithm
5023       The Busy Executive's Database Guide
5023       Fifty Years in Buckingham Palace Kitchens
5023       Prolonged Data Deprivation: Four Case Studies
5023       Cooking with Computers: Surreptitious Balance Sheets
7067       Fifty Years in Buckingham Palace Kitchens

(14 rows affected)
```

To show, in addition, the titles that did not have a sale of more than 500 copies in any store, you can use an outer join query:

```
select distinct salesdetail.stor_id, title
from titles, salesdetail
where qty > 500
and salesdetail.title_id =* titles.title_id
```

```
 stor_id   title
 -------   ----------------------------------------
 NULL      Net Etiquette
 NULL      Life Without Fear
 5023      Sushi, Anyone?
 5023      Is Anger the Enemy?
 5023      The Gourmet Microwave
 5023      But Is It User Friendly?
 5023      Secrets of Silicon Valley
 5023      Straight Talk About Computers
 5023      You Can Combat Computer Stress!
```

```
5023      Silicon Valley Gastronomic Treats
5023      Emotional Security: A New Algorithm
5023      The Busy Executive's Database Guide
5023      Fifty Years in Buckingham Palace Kitchens
7067      Fifty Years in Buckingham Palace Kitchens
5023      Prolonged Data Deprivation: Four Case Studies
5023      Cooking with Computers: Surreptitious Balance Sheets
NULL      Computer Phobic and Non-Phobic Individuals: Behavior
              Variations
NULL      Onions, Leeks, and Garlic: Cooking Secrets of the
              Mediterranean

(18 rows affected)
```

Outer Join Restrictions

In Transact-SQL, a table cannot participate in **both** an outer join clause and in a regular join clause. The following query fails because the *salesdetail* table is asked to do double duty:

```
select distinct sales.stor_id, stor_name, title
from sales, stores, titles, salesdetail
where qty > 500
and salesdetail.title_id =* titles.title_id
and sales.stor_id = salesdetail.stor_id
and sales.stor_id = stores.stor_id

Msg 303, Level 16, State 1:
Server 'RAW', Line 1:
The table 'salesdetail' is an inner member of an
outer-join clause. This is not allowed if the
table also participates in a regular join clause.
```

If you wanted to know the name of the store that sold more than 500 copies of some book, you would have to use a second query. If you submit a query with an outer join and a qualification on a column from the inner table of the outer join, the results may be other than what you expect. The qualification in the query does not restrict the number of rows returned, but rather affects which rows contain the null value. For rows that do not meet the qualification, a null value appears in the inner table's columns of those rows.

How Null Values Affect Joins

If there are null values in the columns of tables being joined, the null values can never match each other. Also, the result of a join of NULL with any other value is NULL. Since null values represent unknown

or inapplicable values, Transact-SQL has no reason to believe that one unknown value matches another.

You can detect the presence of null values in a column from one of the tables being joined only by using an outer join. Here are two tables, each of which has a NULL in the column that will participate in the join. A left outer join displays the NULL in the first table.

Table 1:

```
a           b
---------   ------
        1   one
     NULL   three
        4   join4
```

Table 2:

```
c           d
---------   ------
     NULL   two
        4   four
```

Left Outer Join:

```
select *
from t1, t2
where a *= c
```

```
 a           b       c           d
 ----------  ------  ----------  ------
          1  one           NULL  NULL
       NULL  three         NULL  NULL
          4  join4            4  four
```

Note that the results do not make it particularly easy to distinguish a NULL in the data from a NULL that represents a failure to join. When null values are present in data being joined, it is usually preferable to omit them from the results by using a regular join.

Determining Which Table Columns to Join

The system procedure **sp_helpjoins** lists the columns in two tables or views that are likely join candidates. Its syntax is:

```
sp_helpjoins table1, table2
```

For example, here is how to use **sp_helpjoins** to find the likely join columns between *titleauthor* and *titles*:

```
sp_helpjoins titleauthor, titles
```

The column pairs that **sp_helpjoins** displays come from two sources. First, **sp_helpjoins** checks the *syskeys* table in the current database to see if any foreign keys have been defined on the two tables with **sp_foreignkey**, and then checks to see if any common keys have been defined on the two tables with **sp_commonkey**. If it doesn't find any common keys there, the procedure applies less restrictive criteria to come up with any keys that may be reasonably joined; it checks for keys with the same user datatypes, and if that fails, for columns with the same name and datatype.

For complete information on the system procedures, see the *SQL Server Reference Manual*.

5 Subqueries: Using Queries Within Other Queries

A **subquery** is a **select** statement that is nested inside another **select**, **insert**, **update**, or **delete** statement, inside a conditional statement, or inside another subquery.

This chapter discusses:

- What subqueries are
- Types of subqueries
- Expression subqueries
- Quantified predicate subqueries
- Correlated subqueries

What Are Subqueries?

Subqueries are queries that appear within a **where** or **having** clause of another SQL statement or in the select list of a statement. You can use subqueries to handle query requests that are expressed as the results of other queries. A statement that includes a subquery operates on rows from one table, based on its evaluation of the subquery's **select** list, which can refer to the same table as the outer query, or to a different table. In Transact-SQL, a subquery can also be used almost anywhere an expression is allowed, if the subquery returns a single value.

select statements that contain one or more subqueries are sometimes called **nested queries** or **nested select statements**. The practice of nesting one **select** statement inside another is one reason for the word "structured" in "Structured Query Language."

Many SQL statements that include a subquery, also called an **inner query**, can be alternatively formulated as joins. Other questions can be posed only with subqueries. Some people prefer subqueries to alternative formulations, because they find subqueries easier to understand. Other SQL users avoid subqueries whenever possible. You can choose whichever formulation you prefer. (SQL Server converts some subqueries into joins before processing them.)

Example of Using a Subquery

If you want to find all the books that are the same price as *Straight Talk About Computers*, you can do so in two steps. First, find the price of *Straight Talk*:

```
select price
from titles
where title = "Straight Talk About Computers"

price
-------------
       $19.99

(1 row affected)
```

Now, use the result in a second query to find all the books that cost the same as *Straight Talk*:

```
select title, price
from titles
where price = $19.99

title                                          price
-----------------------------------------    -----
The Busy Executive's Database Guide           19.99
Straight Talk About Computers                 19.99
Silicon Valley Gastronomic Treats             19.99
Prolonged Data Deprivation: Four Case Studies 19.99

(4 rows affected)
```

A subquery solves the problem with a single statement:

```
select title, price
from titles
where price =
   (select price
    from titles
    where title = "Straight Talk About Computers")

title                                        price
--------------------------------------       -----
The Busy Executive's Database Guide           19.99
Straight Talk About Computers                 19.99
Silicon Valley Gastronomic Treats             19.99
Prolonged Data Deprivation: Four Case Studies 19.99

(4 rows affected)
```

Subquery Syntax and General Rules

Always enclose the **select** statement of a subquery in parentheses. The subquery's **select** statement has a **select** syntax that is somewhat restricted, as shown in the following example:

```
(select [distinct] subquery_select_list
   [from [[database.]owner.]{table_name |view_name}
    [({index index_name | prefetch size |[lru|mru]})]}
           [holdlock | noholdlock] [shared]
       [,[[database.]owner.]{table_name |view_name}
    [({index index_name | prefetch size |[lru|mru]})]}
           [holdlock | noholdlock] [shared]]... ]
   [where search_conditions]
   [group by aggregate_free_expression [,
           aggregate_free_expression]... ]
   [having search_conditions])
```

Subqueries can be nested inside the **where** or **having** clause of an outer **select**, **insert**, **update**, or **delete** statement, inside another subquery, or in a select list.

In Transact-SQL, a subquery can appear almost anywhere an expression can be used, if it returns a single value.

Subquery Restrictions

A subquery is subject to the following restrictions:

- Subqueries **cannot** be used in an **order by**, **group by**, or **compute by** list.
- A subquery cannot include **for browse** clauses or a **union**.
- The select list of an inner subquery introduced with a comparison operator can include only one expression or column name, and the subquery must return a single value. The column you name in the **where** clause of the outer statement must be join-compatible with the column you name in the subquery select list.
- *text* and *image* datatypes are not allowed in subqueries.
- Subqueries cannot manipulate their results internally, that is, a subquery cannot include the **order by** clause, the **compute** clause, or the **into** keyword.
- Correlated (repeating) subqueries are not allowed in the **select** clause of an updatable cursor defined by **declare cursor.**
- There is a limit of 16 nesting levels.

- The maximum number of subqueries on each side of a union is 16.

Qualifying Column Names

In the following example, the *pub_id* column in the **where** clause of the outer query is implicitly qualified by the table name *publishers* in the outer query's **from** clause. The reference to *pub_id* in the select list of the subquery is qualified by the subquery's **from** clause—that is, by the *titles* table:

```
select pub_name
from publishers
where pub_id in
   (select pub_id
    from titles
    where type = "business")
```

The general rule is that column names in a statement are implicitly qualified by the table referenced in the **from** clause at the same level.

This is what the query looks like with the implicit assumptions spelled out:

```
select pub_name
from publishers
where publishers.pub_id in
   (select titles.pub_id
    from titles
    where type = "business")
```

It is never wrong to state the table name explicitly, and it is always possible to override implicit assumptions about table names with explicit qualifications.

Subqueries with Correlation Names

As discussed in Chapter 4, "Joins: Retrieving Data from Several Tables," table correlation names are required in self-joins because the table being joined to itself appears in two different roles. Correlation names can also be used in nested queries that refer to the same table in both an inner query and an outer query.

For example, you can find authors who live in the same city as Livia Karsen by using the following subquery:

```
select au1.au_lname, au1.au_fname, au1.city
from authors au1
where au1.city in
   (select au2.city
    from authors au2
    where au2.au_fname = "Livia"
    and au2.au_lname = "Karsen")
```

```
au_lname     au_fname     city
-----------  ---------    -------
Green        Marjorie     Oakland
Straight     Dick         Oakland
Stringer     Dirk         Oakland
MacFeather   Stearns      Oakland
Karsen       Livia        Oakland

(5 rows affected)
```

Explicit correlation names make it clear that the reference to *authors* in the subquery does not mean the same thing as the reference to *authors* in the outer query.

Without explicit correlation, the subquery is:

```
select au_lname, au_fname, city
from authors
where city in
   (select city
    from authors
    where au_fname = "Livia"
    and au_lname = "Karsen")
```

The above query, and other statements in which the subquery and the outer query refer to the same table, can be alternatively stated as self-joins:

```
select au1.au_lname, au1.au_fname, au1.city
from authors au1, authors au2
where au1.city = au2.city
and au2.au_lname = "Karsen"
and au2.au_fname = "Livia"
```

A subquery restated as a join may not return the results in the same order, and the join may require the **distinct** keyword to eliminate duplicates.

Multiple Levels of Nesting

A subquery can include one or more subqueries. Up to 16 subqueries can be nested in a statement.

An example of a problem that can be solved using a statement with multiple levels of nested queries is: "Find the names of authors who have participated in writing at least one popular computing book."

```
select au_lname, au_fname
from authors
where au_id in
   (select au_id
    from titleauthor
    where title_id in
       (select title_id
        from titles
        where type = "popular_comp") )
```

```
au_lname               au_fname
---------------------- ------------
Carson                 Cheryl
Dull                   Ann
Hunter                 Sheryl
Locksley               Chastity

(4 rows affected)
```

The outermost query selects the names of all authors. The next query finds the authors' IDs, and the innermost query returns the title ID numbers PC1035, PC8888, and PC9999.

You can also express this query as a join:

```
select au_lname, au_fname
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
and type = "popular_comp"
```

Subqueries in *update*, *delete*, and *insert* Statements

Subqueries can be nested in **update**, **delete**, and **insert** statements as well as in **select** statements.

➤ ***Note***

Running these example queries will change the *pubs2* database. Ask a System Administrator for help in getting a clean copy of the sample database.

The following query doubles the price of all books published by New Age Books. The statement updates the *titles* table; its subquery references the *publishers* table.

```
update titles
set price = price * 2
where pub_id in
   (select pub_id
    from publishers
    where pub_name = "New Age Books")
```

An equivalent **update** statement using a join is:

```
update titles
set price = price * 2
from titles, publishers
where titles.pub_id = publishers.pub_id
and pub_name = "New Age Books"
```

You can remove all records of sales of business books with this nested **select** statement:

```
delete salesdetail
where title_id in
   (select title_id
    from titles
    where type = "business")
```

An equivalent **delete** statement using a join is:

```
delete salesdetail
from salesdetail, titles
where salesdetail.title_id = titles.title_id
and type = "business"
```

Subqueries in Conditional Statements

Subqueries can also be used in conditional statements. The preceding subquery that removed all records of sales of business books could be rewritten, as shown in the next example, to check for the records **before** deleting them:

```
if exists (select title_id
    from titles
    where type = "business")
begin
    delete salesdetail
    where title_id in
      (select title_id
       from titles
       where type = "business")
end
```

Using Subqueries in Place of an Expression

In Transact-SQL, a subquery can be substituted almost anyplace an expression can be used in a **select**, **update**, **insert**, or **delete** statement. A subquery cannot be used in an **order by** list. Following are some examples that illustrate how you might use this Transact-SQL enhancement.

The following statement finds the titles and types of books that have been written by authors living in California and that are also published there:

```
select title, type
from titles
where title in
   (select title
    from titles, titleauthor, authors
    where titles.title_id = titleauthor.title_id
    and titleauthor.au_id = authors.au_id
    and authors.state = "CA")
and title in
   (select title
    from titles, publishers
    where titles.pub_id = publishers.pub_id
    and publishers.state = "CA")
```

```
title                                type
-----------------------------------  ----------
The Busy Executive's Database Guide  business
Cooking with Computers:
     Surreptitious Balance Sheets    business
Straight Talk About Computers        business
But Is It User Friendly?             popular_comp
Secrets of Silicon Valley            popular_comp
Net Etiquette                        popular_comp

(6 rows affected)
```

The following statement selects the book titles that have more than 5,000 copies sold, lists their prices, and the price of the most expensive book:

```
select title, price,
    (select max(price) from titles)
    from titles
    where total_sales > 5000
```

```
title                                price
-----------------------------------  -----  ------
You Can Combat Computer Stress!       2.99   22.95
The Gourmet Microwave                 2.99   22.95
But Is It User Friendly?             22.95   22.95
Fifty Years in Buckingham Palace
    Kitchens                         11.95   22.95
```

Types of Subqueries

There are two basic types of subqueries:

- Subqueries that are introduced with an unmodified comparison operator, and must return a single value, are **expression subqueries**.
- Subqueries that operate on lists introduced with **in** or with a comparison operator modified by **any** or **all**; or subqueries that are an existence test, introduced with **exists**, are **quantified predicate subqueries**.

Subqueries of either type are either correlated (repeating) or noncorrelated.

- A **noncorrelated subquery** can be evaluated as if it were an independent query. Conceptually, the results of the subquery are substituted in the main statement, or outer query. This is **not** how SQL Server actually processes statements with subqueries. Noncorrelated subqueries can be alternatively stated as joins and are processed as joins by SQL Server.
- A **correlated subquery** cannot be evaluated as an independent query, but it can reference columns in a table listed in the **from** list of the outer query. Correlated subqueries are discussed in detail at the end of this chapter.

The following sections discuss the different types of subqueries.

Expression Subqueries

Expression subqueries are introduced with one of the comparison operators **=**, **!=**, **<>**, **>**, **>=**, **<**, **!>**, **!<** or **<=** and take the general form:

[Start of **select**, **insert**, **update**, **delete** statement or subquery]

where *expression comparison_operator* (*subquery*)

[End of **select**, **insert**, **update**, **delete** statement or subquery]

A subquery that is introduced with an unmodified comparison operator, that is, a comparison operator that is not followed by **any** or **all**, must resolve to a single value. If such a subquery returns more than one value, SQL Server generates an error message.

Ideally, in order to use a subquery that is introduced by an unmodified comparison operator, you must be familiar enough with your data and with the nature of the problem to know that the subquery will return one value.

For example, suppose that each publisher is located in only one city. To find the names of authors who live in the city where Algodata Infosystems is located, write a statement with a subquery that is introduced with the comparison operator **=**:

```
select au_lname, au_fname
from authors
where city =
   (select city
    from publishers
    where pub_name = "Algodata Infosystems")
au_lname       au_fname
-------------- --------------
Carson         Cheryl
Bennet         Abraham

(2 rows affected)
```

Using Scalar Aggregate Functions to Guarantee a Single Value

Subqueries that are introduced with unmodified comparison operators often include scalar aggregate functions, since these return a single value.

For example, this statement finds the names of all the books that are priced higher than the current minimum price:

```
select title
from titles
where price >
   (select min(price)
    from titles)
```

```
title
--------------------------------------------------
The Busy Executive's Database Guide
Cooking with Computers: Surreptitious Balance
     Sheets
Straight Talk About Computers
Silicon Valley Gastronomic Treats
But Is It User Friendly?
Secrets of Silicon Valley
Computer Phobic and Non-Phobic Individuals:
     Behavior Variations
Is Anger the Enemy?
Life Without Fear
Prolonged Data Deprivation: Four Case Studies
Emotional Security: A New Algorithm
Onions, Leeks, and Garlic: Cooking Secrets of the
     Mediterranean
Fifty Years in Buckingham Palace Kitchens
Sushi, Anyone?

(14 rows affected)
```

group by and *having* in Expression Subqueries

Because subqueries that are introduced by unmodified comparison operators must return a single value, they cannot include **group by** and **having** clauses unless you know that the **group by** and **having** clauses will return a single value.

For example, this query finds the books that are priced higher than the lowest priced book in the *trad_cook* category:

```
select title
from titles
where price >
   (select min(price)
    from titles
    group by type
    having type = "trad_cook")
```

Using *distinct* with Expression Subqueries

Subqueries that are introduced with unmodified comparison operators often include the **distinct** keyword to return a single value.

For example, without **distinct**, the following subquery would fail because it would return more than one value:

```
select pub_name from publishers
    where pub_id =
        (select distinct pub_id
        from titles
        where pub_id  = publishers.pub_id)
```

Quantified Predicate Subqueries

Quantified predicate subqueries, which return values of 0 or higher, are subqueries in a **where** or **having** clause that are connected by **any**, **all**, **in**, or **exists**. The **any** or **all** subquery operators modify comparison operators.

- Subqueries introduced with a modified comparison operator, which may include a **group by** or **having** clause, take the general form:

  [Start of **select**, **insert**, **update**, **delete** statement; or subquery]

  where *expression comparison_operator* [any | all]
  (*subquery*)

  [End of **select**, **insert**, **update**, **delete** statement; or subquery]

- Subqueries introduced with **in** or **not in** take the general form:

  [Start of **select**, **insert**, **update**, **delete** statement; or subquery]

  where *expression* [not] in (*subquery*)

  [End of **select**, **insert**, **update**, **delete** statement; or subquery]

- Subqueries introduced by **exists** or **not exists** are existence tests which take the general form:

  [Start of **select**, **insert**, **update**, **delete** statement; or subquery]

  where [not] exists (*subquery*)

  [End of **select**, **insert**, **update**, **delete** statement; or subquery]

Though the keyword **distinct** is allowed in quantified predicate subqueries, the subquery is always processed as if **distinct** were not included.

Subqueries with *any* and *all*

The keywords **all** and **any** modify a comparison operator that introduces a subquery.

Using the **>** comparison operator as an example:

- **>all** means greater than every value, or greater than the maximum value. For example, **>all** (1, 2, 3) means greater than 3.
- **>any** means greater than at least one value, or greater than the minimum value. Therefore, **>any** (1, 2, 3) means greater than 1.

If a subquery is introduced with **all** and a comparison operator does not return any values, the entire query fails.

The use of **all** and **any** can be tricky because computers cannot tolerate the ambiguity that these words sometimes have in English. For example, you might ask the question "Which books commanded an advance greater than any book published by New Age Books?"

This question can be paraphrased to make its SQL "translation" more clear: "Which books commanded an advance greater than the largest advance paid by New Age Books?" The **all** keyword, **not** the **any** keyword, is required here:

```
select title
from titles
where advance > all
   (select advance
    from publishers, titles
    where titles.pub_id = publishers.pub_id
    and pub_name = "New Age Books")
title
----------------------------------------
The Gourmet Microwave

(1 row affected)
```

For each title, the outer query gets the titles and advances from the *titles* table, and it compares these to the advance amounts paid by New Age Books returned from the subquery. The outer query looks at the largest value in the list and determines whether the title being considered has commanded an even greater advance.

>*all* Means Greater Than All Values

In the context of a subquery, **>all** means that in order for a row to satisfy the condition specified in the outer query, the value in the column that introduces the subquery must be greater than each of the values returned by the subquery.

For example, to find the books that are priced higher than the highest-priced book in the *mod_cook* category:

```
select title from titles where price > all
    (select price from titles
    where type =  "mod_cook")

title
--------------------------------------------------
But Is It User Friendly?
Secrets of Silicon Valley
Computer Phobic and Non-Phobic Individuals:
    Behavior Variations
Onions, Leeks, and Garlic: Cooking Secrets of
    the Mediterranean

(4 rows affected)
```

However, if the set returned by the inner query contains a NULL, the query returns 0 rows. This is because NULL stands for "value unknown," and it is impossible to tell whether the value you are comparing is greater than an unknown value.

For example, try to find the books that are priced higher than the highest-priced book in the *popular_comp* category:

```
select title from titles where price > all
    (select price from titles
    where title_id = "popular_comp")

title
--------------------------------------------------

(0 rows affected)
```

No rows were returned because the subquery found that one of the books, *Net Etiquette*, has a null price.

=*all* Means Equal to Every Value

The **=all** operator means equal to each value. For a row to satisfy the condition specified in the outer query, the value in the column that

introduces the subquery must be the same as each value in the list of values returned by the subquery.

For example, the following query finds out which authors live in the same city by looking at the postal code:

```
select au_fname, au_lname, city
from authors
where city = all
     (select city
     from authors
     where postalcode like “946%”)
```

>*any* Means Greater Than At Least One Value

>any means that in order for a row to satisfy the condition specified in the outer query, the value in the column that introduces the subquery must be greater than at least one of the values in the list of values returned by the subquery.

The following query provides an example of a subquery that is introduced with a comparison operator modified by **any**. It finds each title that has an advance larger than any advance amount paid by New Age Books.

```
select title
from titles
where advance > any
   (select advance
    from titles, publishers
    where titles.pub_id = publishers.pub_id
    and pub_name = "New Age Books")

title
--------------------------------------------------
Sushi, Anyone?
Life Without Fear
Is Anger the Enemy?
The Gourmet Microwave
But Is It User Friendly?
Secrets of Silicon Valley
Straight Talk About Computers
You Can Combat Computer Stress!
Emotional Security: A New Algorithm
```

```
The Busy Executive's Database Guide
Fifty Years in Buckingham Palace Kitchens
Cooking with Computers: Surreptitious Balance
    Sheets
Computer Phobic and Non-Phobic Individuals:
    Behavior Variations
Onions, Leeks, and Garlic: Cooking Secrets of
    the Mediterranean

(14 rows affected)
```

For each title selected by the outer query, the inner query finds a list of advance amounts paid by New Age Books. The outer query looks at all the values in the list and determines whether the title being considered has commanded an advance that is larger than any of those values. In other words, the example finds titles with advances as large or larger than the **lowest** value paid by New Age Books.

If the subquery does not return any values, the entire query fails.

=*any* Means Equal to Some Value

The **=any** operator is an existence check; it is equivalent to **in**. For example, to find authors that live in the same city as any publisher, you can use either **=any** or **in**:

```
select au_lname, au_fname
from authors
where city = any
   (select city
    from publishers)
```

```
select au_lname, au_fname
from authors
where city in
   (select city
    from publishers)
```

```
au_lname       au_fname
-------------- --------------
Carson         Cheryl
Bennet         Abraham

(2 rows affected)
```

However, the **!=any** operator is different from **not in**. The **!=any** operator means "not = a **or** not = b **or** not = c"; **not in** means "not = a **and** not = b **and** not = c".

For example, say you want to find the authors who live in a city where no publisher is located. You might try this query:

```
select au_lname, au_fname
from authors
where city != any
   (select city
    from publishers)
```

The results include all 23 authors. This is because every author lives in **some** city where no publisher is located, and each author lives in only one city.

What happens is that the inner query finds all the cities in which publishers are located, and then, for **each** city, the outer query finds the authors who do not live there.

Here is what happens when you substitute **not in** in the same query:

```
select au_lname, au_fname
from authors
where city not in
   (select city
    from publishers)
```

```
au_lname       au_fname
-------------  ------------
del Castillo   Innes
Blotchet-Halls Reginald
Gringlesby     Burt
DeFrance       Michel
Smith          Meander
White          Johnson
Greene         Morningstar
Green          Marjorie
Straight       Dick
Stringer       Dirk
MacFeather     Stearns
Karsen         Livia
Dull           Ann
Hunter         Sheryl
Panteley       Sylvia
Ringer         Anne
Ringer         Albert
Locksley       Chastity
O'Leary        Michael
McBadden       Heather
Yokomoto       Akiko

(21 rows affected)
```

These are the results you want. They include all the authors except Cheryl Carson and Abraham Bennet, who live in Berkeley, where Algodata Infosystems is located.

You get the same results as in the preceding example with the **!=all** operator, which is equivalent to **not in**:

```
select au_lname, au_fname
from authors
where city != all
   (select city
    from publishers)
```

Subqueries Used with *in*

Subqueries that are introduced with the keyword **in** return a list of 0 and higher values. For example, this query finds the names of the publishers who have published business books:

```
select pub_name
from publishers
where pub_id in
   (select pub_id
    from titles
    where type = "business")
```

```
pub_name
----------------------------------------
New Age Books
Algodata Infosystems

(2 rows affected)
```

This statement is evaluated in two steps. First, the inner query returns the identification numbers of the publishers who have published business books, 1389 and 0736. Second, these values are substituted in the outer query, which finds the names that go with the identification numbers in the *publishers* table. The query looks like this:

```
select pub_name
from publishers
where pub_id in ("1389", "0736")
```

Another way to formulate this query using a subquery is:

```
select pub_name
from publishers
where "business" in
   (select type
    from titles
    where pub_id = publishers.pub_id)
```

Note that the expression following the **where** keyword in the outer query can be a constant as well as a column name. You can use other types of expressions, such as combinations of constants and column names.

The preceding queries, like many other subqueries, can be alternatively formulated as a join query:

```
select distinct pub_name
from publishers, titles
where publishers.pub_id = titles.pub_id
and type = "business"
```

Both this query and the subquery versions find publishers who have published business books. All are equally correct and produce the same results, though you may need to use the **distinct** keyword to eliminate duplicates.

However, one advantage of using a join query rather than a subquery for this and similar problems is that a join query shows columns from more than one table in the result. For example, to include the titles of the business books in the result, you would need to use the join version:

```
select pub_name, title
from publishers, titles
where publishers.pub_id = titles.pub_id
and type = "business"
```

```
pub_name              title
--------------------  ----------------------------------------
Algodata Infosystems  The Busy Executive's Database Guide
Algodata Infosystems  Cooking with Computers: Surreptitious
                      Balance Sheets
New Age Books         You Can Combat Computer Stress!
Algodata Infosystems  Straight Talk About Computers

(4 rows affected)
```

Here is another example of a statement that can be formulated either with a subquery or with a join query. The English version of the query is: "Find the names of all second authors who live in California and receive less than 30 percent of the royalties on a book." Using a subquery the statement is:

```
select au_lname, au_fname
from authors
where state = "CA"
and au_id in
   (select au_id
    from titleauthor
    where royaltyper < 30
    and au_ord = 2)
```

```
au_lname                 au_fname
------------------------ ------------
MacFeather               Stearns

(1 row affected)
```

The outer query produces a list of the 15 authors who live in California. The inner query is then evaluated, producing a list of the IDs of the authors who meet the qualifications.

Notice that more than one condition can be included in the **where** clause of both the inner and the outer query.

Using a join, the query is expressed like this:

```
select au_lname, au_fname
from authors, titleauthor
where state = "CA"
  and authors.au_id = titleauthor.au_id
  and royaltyper < 30
  and au_ord = 2
```

A join can always be expressed as a subquery. A subquery can often be expressed as a join.

Subqueries Used with *not in*

Subqueries that are introduced with the keyword phrase **not in** also return a list of 0 and higher values. **not in** means "not = a **and** not = b **and** not = c".

This query finds the names of the publishers who have **not** published business books, the inverse of the example in "Subqueries Used with in" on page 5-18:

```
select pub_name from publishers
where pub_id not in
   (select pub_id
    from titles
    where type = "business")
```

```
pub_name
----------------------------------------
Binnet & Hardley

(1 row affected)
```

The query is exactly the same as the previous one except that **not in** is substituted for **in**. However, this statement cannot be converted to a join. The analogous "not equal" join has a different meaning—it finds the names of publishers who have published **some** book that is not a business book. The difficulties interpreting the meaning of joins that are not based on equality are discussed in more detail in Chapter 4, "Joins: Retrieving Data from Several Tables."

Subqueries Using *not in* with NULL

A subquery using **not in** returns a set of values for each row in the outer query. If the value in the outer query is not in the set returned by the inner query, the **not in** evaluates to TRUE, and the outer query puts the record being considered in the results.

However, if the set returned by the inner query contains no matching value, but it does contain a NULL, the **not in** returns UNKNOWN. This is because NULL stands for "value unknown," and it is impossible to tell whether the value you are looking for is in a set containing an unknown value. The outer query discards the row.

For example, using the *pubs2* database:

```
select pub_name
    from publishers
    where $100.00 not in
        (select price
         from titles
         where titles.pub_id = publishers.pub_id)
```

returns:

```
pub_name
------
New Age Books
```

New Age Books is the only publisher who does not publish any books that cost $100. Binnet & Handley and Algodata Infosystems were not included in the query results because each publishes a book whose price is undecided.

Subqueries Used with *exists*

When a subquery is introduced with the keyword **exists**, the subquery functions as an **existence test**. In other words, the **where** clause of the outer query tests for the existence of the rows returned by the subquery. The subquery does not actually produce any data, but returns a value of TRUE or FALSE.

For example, the following query finds the names of all the publishers who publish business books:

```
select pub_name
from publishers
where exists
   (select *
    from titles
    where pub_id = publishers.pub_id
    and type = "business")

pub_name
----------------------------------------
New Age Books
Algodata Infosystems

(2 rows affected)
```

To conceptualize the resolution of this query, consider each publisher's name in turn. Does this value cause the subquery to return at least one row? In other words, does it cause the existence test to evaluate to TRUE?

In the results of the preceding query, the second publisher's name is Algodata Infosystems, which has an identification number of 1389. Are there any rows in the *titles* table in which *pub_id* is 1389 and *type* is business? If so, "Algodata Infosystems" should be one of the values selected. The same process is repeated for each of the other publisher's names.

A subquery that is introduced with **exists** is different from other subqueries, in these ways:

- The keyword **exists** is not preceded by a column name, constant, or other expression.
- The subquery **exists** evaluates to TRUE or FALSE rather than returning any data.
- The select list of the subquery usually consists of the asterisk (*). There is no need to specify column names, since you are simply testing for the existence or nonexistence of rows that meet the conditions specified in the subquery. Otherwise, the select list

rules for a subquery introduced with **exists** are identical to those for a standard select list.

The **exists** keyword is very important, because there is often no alternative, non-subquery formulation. In practice, a subquery introduced by **exists** is always a correlated subquery (see "Using Correlated Subqueries" on page 5-26).

Although some queries formulated with **exists** cannot be expressed in any other way, all queries that use **in** or a comparison operator modified by **any** or **all** can be expressed with **exists**. Some examples of statements using **exists** and their equivalent alternatives follow.

Here are two ways to find authors that live in the same city as a publisher:

```
select au_lname, au_fname
from authors
where city =any
   (select city
    from publishers)
```

```
select au_lname, au_fname
from authors
where exists
   (select *
    from publishers
    where authors.city = publishers.city)
```

```
au_lname          au_fname
--------------    --------------
Carson            Cheryl
Bennet            Abraham

(2 rows affected)
```

Here are two queries that find titles of books published by any publisher located in a city that begins with the letter "B":

```
select title
from titles
where exists
   (select *
    from publishers
    where pub_id = titles.pub_id
    and city like "B%")
```

```
select title
from titles
where pub_id in
   (select pub_id
    from publishers
    where city like "B%")

title
--------------------------------------------------
The Busy Executive's Database Guide
Cooking with Computers: Surreptitious Balance
     Sheets
You Can Combat Computer Stress!
Straight Talk About Computers
But Is It User Friendly?
Secrets of Silicon Valley
Net Etiquette
Is Anger the Enemy?
Life Without Fear
Prolonged Data Deprivation: Four Case Studies
Emotional Security: A New Algorithm

(11 rows affected)
```

Subqueries Used with *not exists*

not exists is just like **exists** except that the **where** clause in which it is used is satisfied when no rows are returned by the subquery.

For example, to find the names of publishers who do **not** publish business books, the query is:

```
select pub_name
from publishers
where not exists
   (select *
    from titles
    where pub_id = publishers.pub_id
    and type = "business")

pub_name
----------------------------------------
Binnet & Hardley

(1 row affected)
```

This query finds the titles for which there have been no sales:

```
select title
from titles
where not exists
   (select title_id
    from salesdetail
    where title_id = titles.title_id)
```

```
title
----------------------------------------
The Psychology of Computer Cooking
Net Etiquette

(2 rows affected)
```

Finding Intersection and Difference with *exists*

Subqueries that are introduced with **exists** and **not exists** can be used for two set theory operations: intersection and difference. The intersection of two sets contains all elements that belong to both of the original sets. The difference contains the elements that belong only to the first set.

The intersection of *authors* and *publishers* over the *city* column is the set of cities in which both an author and a publisher are located:

```
select distinct city
from authors
where exists
  (select *
   from publishers
   where authors.city = publishers.city)
```

```
city
--------------------
Berkeley

(1 row affected)
```

The difference between *authors* and *publishers* over the *city* column is the set of cities where an author lives but no publisher is located, that is, all the cities except Berkeley:

```
select distinct city
from authors
where not exists
  (select *
   from publishers
   where authors.city = publishers.city)
```

```
city
--------------------
Gary
Covelo
Oakland
Lawrence
San Jose
Ann Arbor
Corvallis
Nashville
Palo Alto
Rockville
Vacaville
Menlo Park
Walnut Creek
San Francisco
Salt Lake City

(15 rows affected)
```

Using Correlated Subqueries

Many of the previous queries could be evaluated by executing the subquery once and substituting the resulting value(s) into the **where** clause of the outer query; these are noncorrelated subqueries. In queries that include a repeating subquery, or **correlated subquery**, the subquery depends on the outer query for its values. The subquery is executed repeatedly, once for each row that is selected by the outer query.

With this query, you can find the names of all authors who earn 100 percent royalty on a book:

```
select au_lname, au_fname
from authors
where 100 in
   (select royaltyper
    from titleauthor
    where au_id = authors.au_id)
```

```
au_lname        au_fname
--------------  ----------
Carson          Cheryl
Ringer          Albert
Straight        Dick
White           Johnson
Green           Marjorie
Panteley        Sylvia
Locksley        Chastity
del Castillo    Innes
Blotchet-Hall   Reginald

(9 rows affected)
```

Unlike most of the previous subqueries, the subquery in this statement cannot be resolved independently of the main query. It needs a value for *authors.au_id*, but this value is a variable—it changes as SQL Server examines different rows of the *authors* table.

This is how the preceding query is evaluated: Transact-SQL considers each row of the *authors* table for inclusion in the results, by substituting the value in each row in the inner query. For example, say that Transact-SQL first examines the row for Cheryl Carson. Then, *authors.au_id* takes the value "238-95-7766," which Transact-SQL substitutes for the inner query:

```
select royaltyper
from titleauthor
where au_id = "238-95-7766"
```

The result is 100, so the outer query evaluates to:

```
select au_lname, au_fname
from authors
where 100 in (100)
```

Since the **where** condition is true, the row for Cheryl Carson is included in the results. If you go through the same procedure with the row for Abraham Bennet, you can see how that row is not included in the results.

Correlated Subqueries with Correlation Names

A correlated subquery can be used to find the types of books that are published by more than one publisher:

```
select distinct t1.type
from titles t1
where t1.type in
   (select t2.type
    from titles t2
    where t1.pub_id != t2.pub_id)
```

```
type
--------------------
business
psychology

(2 rows affected)
```

Correlation names are required in the following query to distinguish between the two roles in which the *titles* table appears. This nested query is equivalent to the self-join query:

```
select distinct t1.type
from titles t1, titles t2
where t1.type = t2.type
and t1.pub_id != t2.pub_id
```

Correlated Subqueries with Comparison Operators

Expression subqueries can be correlated subqueries. For example, to find the sales of psychology books where the quantity is less than average for sales of that title:

```
select s1.ord_num, s1.title_id, s1.qty
from salesdetail s1
where title_id like "PS%"
and s1.qty <
   (select avg(s2.qty)
    from salesdetail s2
    where s2.title_id = s1.title_id)
```

Following are the results of this query:

```
ord_num            title_id     qty
------------------ --------     ---
91-A-7             PS3333        90
91-A-7             PS2106        30
55-V-7             PS2106        31
AX-532-FED-452-2Z7 PS7777       125
BA71224            PS7777       200
NB-3.142           PS2091       200
NB-3.142           PS7777       250
NB-3.142           PS3333       345
ZD-123-DFG-752-9G8 PS3333       750
91-A-7             PS7777       180
356921             PS3333       200

(11 rows affected)
```

The outer query selects the rows of the *sales* table (or "s1"*)* one by one. The subquery calculates the average quantity for each sale being considered for selection in the outer query. For each possible value of *s1*, Transact-SQL evaluates the subquery and puts the record being considered in the results, if the quantity is less than the calculated average.

Sometimes a correlated subquery mimics a **group by** statement. To find the titles of books that have prices higher than average for books of the same type, the query is:

```
select t1.type, t1.title
from titles t1
where t1.price >
   (select avg(t2.price)
    from titles t2
    where t1.type = t2.type)
```

```
type         title
---------    -------------------------------------
business     The Busy Executive's Database Guide
business     Straight Talk About Computers
mod_cook     Silicon Valley Gastronomic Treats
popular_comp But Is It User Friendly?
psychology   Computer Phobic and Non-Phobic
             Individuals: Behavior Variations
psychology   Prolonged Data Deprivation: Four Case
             Studies
trad_cook    Onions, Leeks, and Garlic: Cooking
             Secrets of the Mediterranean

(7 rows affected)
```

For each possible value of *t1*, Transact-SQL evaluates the subquery and includes the row in the results if the price value of that row is greater than the calculated average. It is not necessary to group by type explicitly, because the rows for which the average price is calculated are restricted by the **where** clause in the subquery.

Correlated Subqueries in a *having* Clause

Quantified predicate subqueries can be correlated subqueries.

This example of a correlated subquery in the **having** clause of an outer query finds the types of books in which the maximum advance is more than twice the average within a given group:

```
select t1.type
from titles t1
group by t1.type
having max(t1.advance) >=any
   (select 2 * avg(t2.advance)
   from titles t2
   where t1.type = t2.type)

type
----------
mod_cook

(1 row affected)
```

The subquery above is evaluated once for each group that is defined in the outer query, that is, once for each type of book.

6 Using and Creating Datatypes

You can use the SQL Server system datatypes when defining a column in the **create table** or **alter table** statements, a variable in the **declare** statement, or a parameter in the **create procedure** statement. These statements are described later in this manual. You can also create and use user-defined datatypes in these statements.

This chapter discusses:

- A general overview of datatypes
- The different system datatypes supplied by SQL Server
- How to convert between datatypes
- How mixed-mode arithmetic works in the datatype hierarchy
- How to create user-defined datatypes
- How to get information about a datatype

What Are Transact-SQL Datatypes?

In Transact-SQL, datatypes specify the type of information, size, and storage format of table columns, stored procedure parameters, and local variables. For example, the integer (*int*) datatype is used to hold whole numbers in the range of plus or minus 2^{31}, and the tiny integer (*tinyint*) datatype stores whole numbers between 0 and 255 only.

SQL Server supplies several system datatypes, and two user-defined datatypes, *timestamp* and *sysname*. You can use the **sp_addtype** system procedure to build user-defined datatypes based on the system datatypes. (User-defined datatypes are discussed in "Creating User-Defined Datatypes" on page 6-14.)

You must specify a system datatype or user-defined datatype when declaring a column, local variable, or parameter. The following example uses the system datatypes *char*, *numeric*, and *money* to define the columns in the **create table** statement:

```
create table sales_daily
    (stor_id char(4)
    ord_num numeric(10,0)
    ord_amt money)
```

The following example uses the *bit* system datatype to define the local variable in the **declare** statement:

```
declare @switch bit
```

Subsequent chapters describe in more detail how to declare columns, local variables, and parameters using the datatypes described in this chapter. You can determine which datatypes have been defined for columns of existing tables by using the system procedure **sp_help**.

Using System-Supplied Datatypes

The following table lists the system-supplied datatypes provided for various types of information, the synonyms recognized by SQL Server, and the range and storage size for each. The system datatypes are printed in lowercase characters, although SQL Server allows you to enter them in either uppercase or lowercase. (*timestamp* and *sysname*, like all user-defined datatypes, are case-sensitive.) Most SQL Server-supplied datatypes are not reserved words and can be used to name other objects.

Table 6-1: SQL Server system datatypes

Datatypes by Category	Synonyms	Range	Bytes of Storage
Exact numeric: integers			
tinyint		0 to 255	1
smallint		2^{15} -1 (32,767) to -2^{15} (-32,768)	2
int	*integer*	2^{31} -1 (2,147,483,647) to -2^{31} (-2,147,483,648)	4
Exact numeric: decimals			
numeric (p, s)		10^{38} -1 to -10^{38}	2 to 17
decimal (p, s)	*dec*	10^{38} -1 to -10^{38}	2 to 17
Approximate numeric			
float (precision)		machine dependent	4 or 8
double precision		machine dependent	8
real		machine dependent	4
Money			
smallmoney		214,748.3647 to -214,748.3648	4
money		922,337,203,685,477.5807 to -922,337,203,685,477.5808	8
Date/time			

Table 6-1: SQL Server system datatypes (continued)

Datatypes by Category	Synonyms	Range	Bytes of Storage
smalldatetime		January 1, 1900 to June 6, 2079	4
datetime		January 1, 1753 to December 31, 9999	8
Character			
char(n)	*character*	255 characters or less	*n*
varchar(n)	*character varying, char varying*	255 characters or less	actual entry length
nchar(n)	*national character, national char*	255 characters or less	*n* * *@@ncharsize*
nvarchar(n)	*nchar varying, national char varying, national character varying*	255 characters or less	*@@ncharsize* * number of characters
text		2^{31} -1 (2,147,483,647) bytes or less	0 or multiple of 2K
Binary			
binary(n)		255 bytes or less	*n*
varbinary(n)		255 bytes or less	actual entry length
image		2^{31} -1 (2,147,483,647) bytes or less	0 or multiple of 2K
Bit			
bit		0 or 1	1 (one byte holds up to 8 *bit* columns)

Descriptions of each datatype follow.

Exact Numeric Types: Integers

SQL Server provides three datatypes, *tinyint*, *smallint*, and *int*, to store integers (whole numbers). These types are exact numeric types; they preserve their accuracy during arithmetic operations.

Choose among the integer types based on the expected size of the numbers to be stored. Internal storage size varies by datatype.

Table 6-2: Integer datatypes

Datatype	Stores	Bytes of Storage
tinyint	Whole numbers between 0 and 255, inclusive. (Negative numbers are not permitted.)	1
smallint	Whole numbers between 2^{15} -1 and -2^{15} (32,767 and -32,768), inclusive.	2
int	Whole numbers between 2^{31} - 1 and -2^{31} (2,147,483,647 and -2,147,483,648), inclusive.	4

Exact Numeric Types: Decimal Numbers

SQL Server provides two other exact numeric types, *numeric* and *decimal*, for numbers that include decimal points. Data stored in *numeric* and *decimal* columns is packed to conserve disk space, and preserves its accuracy to the least significant digit after arithmetic operations. The *numeric* and *decimal* types are identical in all respects but one: Only *numeric* types with a scale of 0 can be used for the IDENTITY column.

The exact numeric types accept two optional parameters, *precision* and *scale*, enclosed within parentheses and separated by a comma:

```
datatype [(precision [, scale])]
```

SQL Server defines each combination of **precision** and scale as a distinct datatype. For example, *numeric*(10,0) and *numeric*(5,0) are two separate datatypes. The precision and scale determine the range of values that can be stored in a *decimal* or *numeric* column:

- The precision specifies the maximum number of decimal digits that can be stored in the column. It includes all digits, to the right or left of the decimal point. You can specify a precision of 1 to 38 digits, or use the default precision of 18 digits.
- The **scale** specifies the maximum number of digits that can be stored to the right of the decimal point. Note that the scale must be less than or equal to the precision. You can specify a scale of 0 to 38 digits, or use the default scale of 0 digits.

Exact numeric types with a scale of 0 are displayed without a decimal point. If you enter a value that exceeds either the precision or scale for the column, SQL Server flags the entry as an error.

The storage size for a *numeric* or *decimal* column depends on its precision. The minimum storage requirement is 2 bytes for a 1- or 2-digit column. Storage size increases by 1 byte for each additional 2 digits of precision, to a maximum of 17 bytes.

Approximate Numeric Datatypes

SQL Server provides three approximate numeric types, *float*, *double precision*, and *real*, for numeric data that can tolerate rounding during arithmetic operations. Use the approximate numeric types for data that covers a wide range of values. They support all aggregate functions and all arithmetic operations except modulo (%).

The *real* and *double precision* types are built on types supplied by the operating system. The *float* type accepts an optional precision in parentheses. *float* columns with a precision of 1–15 are stored as *real*; those with higher precision are stored as *double precision*. The range and storage precision for all three types is machine dependent.

The following table shows the range, display precision, and storage size for each approximate numeric type. Note that **isql** displays only six significant digits after the decimal point and rounds the remainder:

Table 6-3: Approximate numeric datatypes

Datatype	Bytes of Storage
float[(*default precision*)]	4 for *default precision* < 16, 8 for *default precision* >= 16
double precision	8
real	4

Character Datatypes

Use the character datatypes to store strings consisting of letters, numbers, and symbols entered within single or double quotes. You can use the **like** keyword to search these strings for particular characters and the built-in string functions to manipulate their contents. Strings consisting of numbers can be converted to exact

and approximate numeric datatypes with the **convert** function, then used for arithmetic.

The *char(n)* datatype stores fixed-length strings, and the *varchar(n)* datatype stores variable-length strings, in single-byte character sets such as English. Their national character counterparts, *nchar(n)* and *nvarchar(n),* store fixed- and variable-length strings in multibyte character sets such as Japanese. You can specify the maximum number of characters with *n,* or use the default column length of one character. For strings longer than bytes, use the *text* datatype.

Table 6-4: Character datatypes

Datatype	Stores	Bytes of Storage
char(n)	Fixed-length data, such as social security numbers or postal codes	*n*
varchar(n)	Data, such as names, that is likely to vary greatly in length	Actual number of characters entered
nchar(n)	Fixed-length data in multibyte character sets	*n * @@ncharsize*
nvarchar(n)	Variable-length data in multibyte character sets	Actual number of characters * *@@ncharsize*
text	Up to 2,147,483,647 bytes of printable characters on linked lists of data pages	0 when uninitialized; a multiple of 2K after initialization

SQL Server truncates entries to the specified column length without warning or error unless you set **string_rtruncation on**. See the **set** command in the *SQL Server Reference Manual* for more information. The empty string, “” or ‘’ , is stored as a single space rather than as NULL. Thus “abc” + “” + “def” is equivalent to “abc def”, not to “abcdef”.

Fixed- and variable-length columns behave somewhat differently:

- Data in fixed-length columns is blank-padded to the column length. For the *char* datatype, storage size is *n* bytes; for the *nchar* datatype, *n* times the average national character length (*@@ncharsize*). When you create a *char* or *nchar* column that allows nulls, SQL Server automatically converts it to a *varchar* or *nvarchar* column and uses the storage rules for those datatypes. (This is not true of *char* and *nchar* variables and parameters.)

- Data in variable-length columns is stripped of trailing blanks; storage size is the actual length of the data. For *varchar* columns, this is the number of characters; for *nvarchar* columns, the number of characters times the average character length. Variable-length character data may require less space than fixed-length data, but it is accessed somewhat more slowly.

The *text* datatype stores up to 2,147,483,647 bytes of printable characters on linked lists of separate data pages. To save storage space, define *text* columns as NULL. When you initialize a *text* column with a non-null **insert** or **update**, SQL Server assigns a text pointer and allocates an entire 2K data page to hold the value. Each page stores a maximum 1,800 bytes of data. To add data without saving large blocks of text in your transaction log, use **writetext**. See the *SQL Server Reference Manual* for details.

Binary Datatypes

The binary datatypes store raw binary data such as pictures, in a hexadecimal-like notation. Binary data begins with the characters "0x" and includes any combination of digits and the uppercase and lowercase letters A-F.

➤ ***Note***

SQL Server manipulates the binary types in a platform-specific manner. For true hexadecimal data, use the **hextoint** and **inttohex** functions. See Chapter 10, "Using the Built-In Functions in Queries."

Use the *binary(n)* and *varbinary(n)* datatypes to store data up to 255 bytes in length. Each byte of storage holds 2 binary digits. Specify the column length with *n*, or use the default length of one byte. If you enter a value longer than *n*, SQL Server truncates the entry to the specified length without warning or error.

- Use the fixed-length binary type, *binary(n)*, for data in which all entries are expected to have a similar length. Because entries in *binary* columns are zero-padded to the column length, they may require more storage space than those in *varbinary* columns, but are accessed somewhat faster.
- Use the variable-length binary type, *varbinary(n)*, for data that is expected to vary greatly in length. Storage size is the actual size of the data values entered, not the column length. Trailing zeros are truncated.

When you create a *binary* column that allows nulls, SQL Server automatically converts it to a *varbinary* column and uses the storage rules for that datatype.

You can search binary strings with the **like** keyword and operate on them with the built-in string functions. Because the exact form in which you enter a particular value depends upon the hardware you are using**, calculations involving binary data may produce different results on different platforms.**

Use the *image* datatype to store larger blocks of binary data on external data pages. An *image* column can store up to 2,147,483,647 bytes of data on linked lists of data pages separate from other data storage for the table. When you initialize an *image* column with a non-null **insert** or **update**, SQL Server assigns a text pointer and allocates an entire 2K data page to hold the value. Each page stores a maximum of 1,800 bytes.

To save storage space, define *image* columns as NULL. To add *image* data without saving large blocks of text in your transaction log, use **writetext**. See the *SQL Server Reference Manual* for details.

The following table summarizes the storage requirements for the binary datatypes:

Table 6-5: Binary datatypes

Datatype	Bytes of Storage
binary(n)	*n*
varbinary(n)	Actual length of entry
image	0 when uninitialized; a multiple of 2K after initialization

Money Datatypes

The money datatypes, *money* and *smallmoney*, store monetary data. You can use these datatypes for U.S. dollars and other decimal currencies, although SQL Server provides no means to convert from one currency to another. You can use all arithmetic operations except modulo, and all aggregate functions, with *money* and *smallmoney* data.

Both *money* and *smallmoney* are accurate to one ten-thousandth of a monetary unit, but round values up to two decimal places for display purposes. The default print format places a comma after every three digits.

The following table summarizes the range and storage requirements for money datatypes:

Table 6-6: Money datatypes

Datatype	Range	Bytes of Storage
money	Monetary values between +922,337,203,685,477.5807 and -922,337,203,685,477.5808	8
smallmoney	Monetary values between +214,748.3647 and -214,748.3648	4

Date and Time Datatypes

Use the *datetime* and *smalldatetime* datatypes to store date and time information from January 1, 1753 through December 31, 9999. Dates outside this range must be entered, stored, and manipulated as *char* or *varchar* values.

- *datetime* columns hold dates between January 1, 1753 and December 31, 9999. *datetime* values are accurate to 1/300 second on platforms that support this level of granularity. Storage size is 8 bytes: 4 bytes for the number of days since the base date of January 1, 1900 and 4 bytes for the time of day.
- *smalldatetime* columns hold dates from January 1, 1900 to June 6, 2079, with accuracy to the minute. Its storage size is 4 bytes: 2 bytes for the number of days after January 1, 1900, and 2 bytes for the number of minutes since midnight.

Date/time information must be enclosed in single or double quotes. It can be entered in either upper- or lowercase and can include spaces between data parts. SQL Server recognizes a wide variety of data entry formats, which are described in Chapter 8. Values such as zero or 00/00/00, which are not recognized as dates, are rejected.

The default display format for dates is "Apr 15 1987 10:23PM". You can use the **convert** function for even more styles of date display. You can also do some arithmetic calculations on *datetime* values with the built-in date functions.

The following table summarizes the range and storage requirements for date datatypes:

Table 6-7: Date datatypes

Datatype	Range	Bytes of Storage
datetime	January 1, 1753 to December 31, 9999	8
smalldatetime	January 1, 1900 to June 6, 2079	4

The *bit* Datatype

Use *bit* columns for true and false or yes and no types of data. *bit* columns hold either 0 or 1. Integer values other than 0 or 1 are accepted, but are always interpreted as 1. Storage size is 1 byte. Multiple *bit* datatypes in a table are collected into bytes. For example, 7 *bit* columns fit into 1 byte; 9 *bit* columns take 2 bytes.

Columns of datatype *bit* cannot be NULL and cannot have indexes on them. The *status* column in the *syscolumns* system table indicates the unique offset position for bit columns.

The *timestamp* Datatype

SQL Server also supplies the *timestamp* user-defined datatype. *timestamp* columns are necessary in tables that are to be browsed in Open Client™ DB-Library™ applications.

Every time a row containing a *timestamp* column is inserted or updated, the *timestamp* column is automatically updated. A table can have only one column of the *timestamp* datatype. A column named *timestamp* will automatically have the system datatype *timestamp*. Its definition is *varbinary(8)* NULL.

Because *timestamp* is a user-defined datatype, you cannot use it to define other user-defined datatypes. You must enter it as "timestamp", in all lowercase letters.

The *sysname* Datatype

sysname is a user-defined datatype that is distributed on the SQL Server installation tape and used in the system tables. Its definition is:

```
varchar(30) "not null"
```

You cannot use the *sysname* datatype to create a column. You can, however, create a user-defined datatype with a base type of *sysname*. You can then use the **user-defined datatype** to create columns. For more information about user-defined datatypes, see "Creating User-Defined Datatypes" on page 6-14.

Converting Between Datatypes

SQL Server automatically handles many conversions from one datatype to another. These are called implicit conversions. You can explicitly request other conversions with the **convert**, **inttohex**, and **hextoint** functions. Still other conversions cannot be done, either explicitly or automatically, because of incompatibilities between the datatypes.

For example, while SQL Server automatically converts *char* expressions to *datetime* for the purposes of the comparison, if they can be interpreted as *datetime* values; you must use the **convert** function to convert *char* to *int*. Similarly, you must use **convert** on integer data if you want SQL Server to treat it as character data so that you can use the **like** keyword with it.

The syntax for the **convert** function is:

```
convert (datatype, expression, [style])
```

Here is an example:

```
select title, total_sales
from titles
where convert (char(20), total_sales) like "2%"
```

The optional *style* parameter is used to convert *datetime* values to *char* or *varchar* datatypes in order to get a wide variety of date display formats.

See Chapter 10 for details on the **convert**, **inttohex**, and **hextoint** functions.

Mixed-Mode Arithmetic and Datatype Hierarchy

When you perform arithmetic on values with different datatypes, SQL Server must determine the datatype and, in some cases, the length and precision, of the result.

Each system datatype has a **datatype hierarchy**, which is stored in the *systypes* system table. User-defined datatypes inherit the hierarchy of the system type on which they are based.

The following query ranks the datatypes in a database by hierarchy. In addition to the information shown below, your query results will include information about any user-defined datatypes in the database:

```
select name,hierarchy
from systypes
order by hierarchy
```

```
name                          hierarchy
----------------------------- -------
floatn                              1
float                               2
datetimn                            3
datetime                            4
real                                5
numericn                            6
numeric                             7
decimaln                            8
decimal                             9
moneyn                             10
money                              11
smallmoney                         12
smalldatetime                      13
intn                               14
int                                15
smallint                           16
tinyint                            17
bit                                18
varchar                            19
```

```
sysname                         19
nvarchar                        19
char                            20
nchar                           20
varbinary                       21
timestamp                       21
binary                          22
text                            23
image                           24
(28 rows affected)
```

The datatype hierarchy determines the results of computations using values of different datatypes. The result value is assigned the datatype that is closest to the top of the list.

In the following example, *qty* from the *sales* table is multiplied by *royalty* from the *roysched* table. *qty* is a *smallint*, which has a hierarchy of 16; *royalty* is an *int*, which has a hierarchy of 15. Therefore, the datatype of the result is an *int*.

```
smallint(qty) * int(royalty) = int
```

Working with *money* Datatypes

If you are combining *money* and literals or variables, and need results of *money* type, use *money* literals or variables:

```
select moneycol * $2.5 from mytable
```

If you are combining money with a *float* or *numeric* datatype from column values, use the **convert** function:

```
select convert (money, moneycol * percentcol)
    from debts, interest
```

Determining Precision and Scale

For the *numeric* and *decimal* types, each combination of precision and scale is a distinct SQL Server datatype. If you perform arithmetic on two *numeric* or *decimal* values, *n1* with precision *p1* and scale *s1*, and *n2* with precision *p2* and scale *s2*, SQL Server determines the precision and scale of the results as follows:

Table 6-8: Precision and scale after arithmetic operations

Operation	Precision	Scale
n1 + n2	max(s1, s2) + max(p1 -s1, p2 - s2) + 1	max(s1, s2)

Table 6-8: Precision and scale after arithmetic operations

Operation	Precision	Scale
n1 - n2	max(s1, s2) + max(p1 -s1, p2 - s2) + 1	max(s1, s2)
n1 * n2	s1 + s2 + (p1 - s1) + (p2 - s2) + 1	s1 + s2
n1 / n2	max(s1 + p2 + 1, 6) + p1 - s1 + p2	max(s1 + p2 -s2 + 1, 6)

Creating User-Defined Datatypes

A Transact-SQL enhancement to SQL allows you to name and design your own datatypes to supplement the system datatypes. A user-defined datatype is defined in terms of system datatypes. You can give one name to a frequently used datatype definition. This makes it easy for you to custom fit datatypes to columns.

➤ ***Note***

To use a user-defined datatype in more than one database, create it in the *model* database. The user-defined datatype definition then becomes known to all new databases you create.

Once you define a datatype, it can be used as the datatype for any column in the database. For example, *tid* is used as the datatype for columns in several *pubs2* tables: *titles.title_id*, *titleauthor.title_id*, *sales.title_id*, and *roysched.title_id*.

The advantage of user-defined datatypes is that you can bind rules and defaults to them, for use in several tables. For more about this topic, see Chapter 12.

The system procedure **sp_addtype** is used to create user datatypes. It takes as parameters the name of the user datatype being created, the SQL Server-supplied datatype from which it is being built, and an optional NULL, NOT NULL, or IDENTITY specification.

You can build a user-defined datatype using any system datatype other than *timestamp*. User-defined datatypes have the same datatype hierarchy as the system datatypes on which they are based. Unlike SQL Server-supplied datatypes, user-defined datatype names are case sensitive.

Here is the syntax for **sp_addtype**:

```
sp_addtype datatypename,
   phystype [(length) | (precision [, scale])]
   [, "identity" |nulltype]
```

Here's how *tid* was defined:

```
sp_addtype tid, "char(6)", "not null"
```

You must enclose a parameter within single or double quotes if it includes a blank or some form of punctuation, or if it is a keyword other than **null** (for example, **identity** or **sp_helpgroup**). In this example, quotes are required around *char(6)* because of the parentheses, and around NOT NULL because of the blank. They are not required around *tid*.

Specifying Length, Precision, and Scale

When you build a user-defined datatype based upon certain SQL Server datatypes, you must specify additional parameters:

- The *char*, *nchar*, *varchar*, *nvarchar*, *binary*, and *varbinary* datatypes expect a length in parentheses. If you do not supply one, SQL Server assumes the default length of one character.
- The *float* datatype expects a precision in parentheses. If you do not supply one, SQL Server uses the default precision for your platform.
- The *numeric* and *decimal* datatypes expect a precision and scale, in parentheses and separated by a comma. If you do not supply them, SQL Server uses a default precision of 18 and scale of 0.

You cannot change the length, precision, or scale specification when you include the user-defined datatype in a **create table** statement.

Specifying Null Type

The null type determines how the user-defined datatype treats nulls. You can create a user-defined datatype with a null type of "null", "NULL", "nonull", "NONULL", "not null", or "NOT NULL". By definition, *bit* and IDENTITY types do not allow null values.

If you omit the null type, SQL Server uses the null mode defined for the database (by default, NOT NULL). For compatibility with the SQL standards, use the **sp_dboption** system procedure to set the **allow nulls by default** option to **true.**

You can override the null type when you include the user-defined datatype in a **create table** statement.

Associating Rules and Defaults with User-Defined Datatypes

Once you have created a user-defined datatype, you can use the system procedures **sp_bindrule** and **sp_bindefault** to associate rules and defaults with the datatype. Using the **sp_help** system procedure, you can print a report that lists the rules, defaults, and other information associated with the datatype.

Rules and defaults are discussed in Chapter 12. For complete information on system procedures, see the *SQL Server Reference Manual*.

Dropping a User-Defined Datatype

To drop a user-defined datatype, execute **sp_droptype**:

```
sp_droptype typename
```

➤ ***Note***

You cannot drop a datatype that is already in use in any table.

Getting Information About Datatypes

Use the **sp_help** system procedure to display information about the properties of a system datatype or a user-defined datatype. The report indicates the base type from which the datatype was created, whether or not it allows nulls, the names of any rules and defaults bound to the datatype, and whether it has the IDENTITY property.

The following examples display the information about the *money* system datatype and the *tid* user-defined datatype:

```
sp_help money
Type_name  Storage_type Length Prec  Scale
---------- ------------ ------ ----- -----
money      money             8 NULL  NULL
Nulls   Default_name  Rule_name   Identity
-----   ------------  ---------   --------
    1   NULL          NULL                0

(return status = 0)
sp_help tid
Type_name  Storage_type Length Prec  Scale
---------- ------------ ------ ----- -----
tid        varchar           6 NULL  NULL
Nulls   Default_name  Rule_name   Identity
-----   ------------  ---------   --------
    0   NULL          NULL                0

(return status = 0)
```

7 Creating Databases and Tables

This chapter describes how to set up databases and tables, a process called **data definition**. It discusses:

- A general overview of databases and their tables
- How to use and create a database
- How to create tables and define their columns
- How to create user-defined datatypes
- How to change existing tables
- How to get information about databases and tables

If you do not plan to create your own databases and tables, read about the basic database and table concepts (described in the following section) and about the **use** command (described in a later section). You can then skip the rest of this chapter.

What Are Databases and Tables?

A database stores information (data) in a set of database objects, such as tables, that relate to each other. A table is a collection of rows that have associated columns containing individual data items. You define how your data is organized when you create your databases and tables. This process is called data definition.

SQL Server database objects include:

- Tables
- Rules
- Defaults
- Stored procedures
- Triggers
- Views
- Referential integrity constraints
- Check integrity constraints

This chapter covers only the creation, modification, and deletion of databases and tables, including integrity constraints.

Rules and defaults are covered in Chapter 12; views are discussed in Chapter 9; stored procedures are covered in Chapter 14; and triggers are discussed in Chapter 15.

Columns and **datatypes** define the type of data included in tables, while indexes describe how that data is organized in tables. They are not considered database objects by SQL Server and are not listed in *sysobjects*. Columns and datatypes are covered in this chapter; indexes are discussed in Chapter 11.

Enforcing Data Integrity in Databases

Data integrity refers to the correctness and completeness of data within a database. To enforce data integrity, you can constrain or restrict the data values that users can insert, delete, or update in the database. For example, the integrity of data in the *pubs2* database requires that a book title in the *titles* table must have a publisher in the *publishers* table. You must not insert books into *titles* that do not have a valid publisher, since it violates the data integrity of *pubs2*.

Transact-SQL provides several mechanisms for integrity enforcement in a database such as rules, defaults, indexes, and triggers. They allow you to maintain the following types of data integrity:

- Requirement – This integrity requires that a table column must contain a valid value in every row; it cannot allow null values. The **create table** statement allows you to restrict null values for a column.
- Check or Validity – This integrity limits or restricts the data values inserted into a table column. You can use triggers or rules to enforce this data integrity.
- Uniqueness – This integrity requires that no two table rows have the same non-null values for one or more table columns. You can use indexes to enforce this integrity.
- Referential – This integrity requires that data inserted into a table column must already have matching data in another table column or another column in the same table. You can use triggers to enforce this integrity.

Consistency of data values in the database is another data integrity, which is described in Chapter 17.

As an alternative to using rules, defaults, indexes, and triggers, Transact-SQL provides a series of **integrity constraints** as part of the

create table statement to enforce data integrity as defined by the SQL standards. These integrity constraints are described later in this chapter.

Permissions Within Databases

Whether or not you can create and drop databases and database objects depends on your permissions or privileges. Ordinarily, a System Administrator or Database Owner sets up the permissions for you, based on the kind of work you do and the functions you need. These permissions might be different for each user in a given installation or database.

You can determine what your permissions are by executing:

```
sp_helprotect user_name
```

where *user_name* is your SQL Server login name.

To make your experiments with database objects as convenient as possible, the *pubs2* database has a "**guest**" user in its *sysusers* system table. The script that creates *pubs2* grants a variety of permissions to "guest".

The "guest" mechanism means that anyone who has a **login** on SQL Server, that is, is listed in *master..syslogins*, has access to *pubs2* and permission to create and drop objects such as tables, indexes, defaults, rules, procedures, and so on. The "guest" user name also allows you to use certain stored procedures, create user-defined datatypes, query the database, and modify the data in it.

To use the *pubs2* database, issue the **use** command. SQL Server checks whether you are listed under your own name in *pubs2..sysusers*. If not, you are admitted as a guest without any action on your part. If you are listed in *pubs2..sysusers*, you'll be admitted as yourself, but your permissions may be different than those of "guest". All the examples in this chapter assume you are being treated as "guest".

Using and Creating Databases

A database is a collection of related tables and other database objects—views, indexes, and so on.

When SQL Server is first installed, it contains these **system databases**:

- The *master* database controls the user databases and the operation of SQL Server as a whole.

- The *sybsystemprocs* database contains the system stored procedures.
- The temporary database, *tempdb*, stores temporary objects, including temporary tables created with the name prefix "*tempdb..*".
- The *model* database is used by SQL Server as a template for creating new user databases.

In addition, System Administrators can install the sample database, *pubs2*, and the syntax database, *sybsyntax*, by using **isql** and the SQL scripts that are included in the *scripts* directory. The *pubs2* database serves as the basis of most of the examples in the SQL Server documentation set. The *sybsyntax* database stores all the syntax information for commands and procedures accessed by **sp_syntax**.

The *pubs2* and *sybsyntax* databases are user databases. All of your data—your reason for using a database management system—is stored in user databases. SQL Server manages each database by means of system tables. The **data dictionary** tables in the *master* database and in other databases are considered system tables.

Choosing a Database: *use*

Much of the time, you will be using a database that already exists. The syntax of the command that accesses an already existing database is:

```
use database_name
```

For example, to access the database called *pubs2*, type:

```
use pubs2
```

This command allows you to access the *pubs2* database only if you are a known user in *pubs2*. Otherwise, SQL Server displays an error message. It is up to the owner of the database to give you access to it by executing the system procedure **sp_adduser**.

Most users will be able to look at the system tables in the *master* database by means of the guest mechanism previously described. Users not recognized by name in the *master* database are allowed in and treated as a user named "guest". The "guest" user is added to the *master* database in the script that creates the *master* database when it is installed.

A Database Owner, "dbo", can add a "guest" user to any user database with the **sp_adduser** system procedure. System Administrators automatically become the Database Owner in any

database they use. For more information, see the *System Administration Guide* and the *SQL Server Reference Manual*.

It is likely that you will be automatically connected to the *master* database when you log onto SQL Server, so that you must issue the **use** command in order to access some other database. You or a System Administrator can change the database to which you initially connect by using the system procedure **sp_modifylogin**. Only a System Administrator can change the default database for another user.

Creating a User Database: *create database*

You can create a new database if a System Administrator has granted you permission to use the **create database** command. You must be using the *master* database when you create a new database. In many enterprises, a System Administrator creates all databases. The creator of a database is its owner. Another user who creates a database for you can transfer ownership of it to you with the system procedure **sp_changedbowner**.

The Database Owner is responsible for giving users access to the database and for granting and revoking certain other permissions to users. In some organizations, the Database Owner is also responsible for maintaining regular backups of the database and for reloading it in case of system failure. The Database Owner can always impersonate any other user of the database, temporarily attaining that user's permissions, with the **setuser** command.

Because each database is allocated a significant amount of space, even if it contains only small amounts of data, you may not be given permission to use the **create database** command. If this is the case, you may want to skip this section and go on to the discussion of creating database tables, "Creating Tables" on page 7-10.

The simplest form of the **create database** command is:

```
create database database_name
```

To create the *newpubs* database, be sure you are using the *master* database rather than *pubs2*, and then type this command:

```
create database newpubs
```

A database name must be unique on SQL Server, and must follow the rules for identifiers given in Chapter 1. SQL Server can manage up to 32,767 databases. You can create only one database at a time. The maximum number of segments for any database is 32.

SQL Server creates a new database as a copy of the *model* database, which contains the system tables that belong in every user database.

The creation of a new database is recorded in the *master* database tables *sysdatabases* and *sysusages*.

The full syntax of the **create database** command is:

```
create database database_name
   [on {default | database_device} [= size]
       [, database_device [= size]]...]
   [log on database_device [= size]
       [, database_device [= size]]...]
   [with override]
   [for load]
```

This chapter describes all the **create database** options except for **with override**. For information about **with override**, see the *System Administration Guide*.

➤ ***Note***

In the earlier examples and the examples in the next section, the **log on** clause is not shown for the sake of simplicity. When you create production databases, however, you should always create them with the **log on** clause. See the following section.

The *on* Clause

The optional **on** clause allows you to specify where to store the database and how much space to allocate for it in megabytes. If you use the keyword **default**, the database is assigned to an available database device in the pool of default database devices indicated in the *master* database table *sysdevices*. Use the system procedure **sp_helpdevice** to see which devices are in the default list.

➤ ***Note***

A System Administrator may have made certain storage allocations based on performance statistics and other considerations. Before creating databases, you should check with a System Administrator.

To specify a size of 5MB for a database to be stored in this default location, use **on default** = *size* like this:

```
create database newpubs
on default = 5
```

If you wish to specify a different location for the database, give the logical name of the database device on which you want it stored. A database can be stored on more than one database device, with different amounts of space on each.

This statement creates the *newpubs* database and allocates to it 3MB on *pubsdata* and 2MB on *newdata*:

```
create database newpubs
on pubsdata = 3, newdata = 2
```

If the **on** clause and the size are omitted, the database is created with 2MB of space from the pool of default database devices indicated in *sysdevices*.

A database allocation can range in size from 2MB to 2^{23}MB.

The *log on* Clause

Unless you are creating very small, noncritical databases, you should always use the **log on *database_device*** extension to **create database**. It places the transaction logs on a separate database device. There are several reasons for placing the logs on a separate device:

- It allows you to use the **dump transaction** command, rather than **dump database**, thus saving time and tapes.
- It allows you to establish a fixed size for the log, keeping it from competing with other database activity for space.

There are additional reasons for placing the log on a separate **physical** device from the data tables:

- It improves performance.
- It ensures full recovery in the event of hard disk crashes.

The following command places the log for *newpubs* on the logical device *"pubslog"*, with a size of 1 megabyte:

```
create database newpubs
on pubsdata = 3, newdata = 2
log on pubslog = 1
```

➤ ***Note***

When you use the **log on** extension, you are placing the database transaction log on a segment named "logsegment". If you ever need to add more space for the log, you need to use **alter database** and, in some cases, the system procedure **sp_extendsegment**. See the *SQL Server Reference Manual* or the *SQL Server System Administration Guide* for further details.

The size of the device required for the transaction log varies according to the amount of update activity and the frequency of transaction log dumps. As a rule of thumb, allocate to the log 10 percent to 25 percent of the space you allocate to the database itself.

The *for load* Option

The optional **for load** clause invokes a streamlined version of **create database** that can only be used for loading a database dump. Use the option for recovery from media failure or for moving a database from one machine to another. See the *System Administration Guide* for further details.

Dropping Databases

Removing a database is accomplished with the **drop database** command. **drop database** deletes the database and all of its contents from SQL Server, frees the storage space that had been allocated for it, and deletes references to it from the *master* database.

The syntax of this command is:

```
drop database database_name [, database_name]...
```

You cannot drop a database that is in use, that is, open for reading or writing by any user.

As indicated, you can drop more than one database in a single command. For example:

```
drop database newpubs, newdb
```

A damaged database cannot be removed with **drop database**. Use the **dbcc dbrepair** command.

Altering the Sizes of Databases

If a database has filled all of its allocated storage space, you cannot add new data or updates to it. Existing data, of course, is always preserved. If the space allocated for a database proves to be too small, the Database Owner can increase it with the **alter database** command. **alter database** permission defaults to the Database Owner, and cannot be transferred. You must be using the *master* database in order to use the **alter database** command.

The default increase is 2MB from the default pool of space. This statement adds 2MB to *newpubs* on the default database device:

```
alter database newpubs
```

The full **alter database** syntax allows you to extend a database by a specified number of megabytes (minimum 1MB) and to specify where the storage space is to be added:

```
alter database database_name
   [on {default | database_device } [= size]
       [, database_device [= size]]...]
   [log on { default | database_device } [ = size ]
       [ , database_device [= size]]...]
   [with override]
   [for load]
```

The **on** clause in the **alter database** command is just like the **on** clause in the **create database** command. The **for load** clause is just like the **for load** clause in the **create database** command and can be used only on a database created with the **for load** clause.

To increase the space allocated for *newpubs* by 2MB on the database device *pubsdata*, and by 3MB on the database device *newdata*, type:

```
alter database newpubs
on pubsdata = 2, newdata = 3
```

When you use **alter database** to allocate more space on a device already in use by the database, all of the segments already on that device use the added space fragment. All of the objects that are already mapped to the existing segments can now grow into the added space. The maximum number of segments for any database is 32.

When you use **alter database** to allocate space on a device that is not yet in use by a database, the *system* and *default* segments are mapped to the new device. If you wish to change this segment mapping, you must use **sp_dropsegment** to drop the unwanted segments from the device.

➤ ***Note***

Using **sp_extendsegment**, **logsegment**, or **device_name** automatically unmaps the system and default segments.

For information about **with override**, see the *System Administration Guide*.

Creating Tables

When you create a table, you name its columns and supply a datatype for each column. You can also specify whether or not a particular column can hold null values or specify any integrity constraints for columns in the table.

There can be as many as 2 billion tables per database.

Example of Creating a Table

Be sure you are using the *newpubs* database you created in the previous section if you want to try these examples. Otherwise, all these changes will affect another database, like *pubs2*.

To create a table, use the **create table** command. Its simplest form is:

```
create table table_name
(column_name datatype)
```

For example, to create a table named *names* with one column named *some_name*, and a fixed length of 11 bytes, enter:

```
create table names
(some_name char(11))
```

You can define up to 250 columns. If you have **set quoted_identifier on**, both the table name and the column names can be delimited identifiers. Otherwise, they must follow the rules for identifiers given in Chapter 1, "Introduction." Column names must be unique within a given table, but you can use the same column name in different tables in the same database.

There must be a datatype for each column. The word "char" after the column name in the example above refers to the datatype of the column—the type of value that column will contain. Datatypes are discussed in Chapter 6, "Using and Creating Datatypes."

The number in parentheses after the datatype gives the maximum number of bytes that can be stored in the column. You give a

maximum length for some datatypes. Others have a system-defined length.

Be sure to put parentheses around the list of column names, and commas after each column definition.

Choosing Table Names

The **create table** command builds the new table in the currently open database. Table names must be unique for each user.

You can create temporary tables either by preceding the table name in a **create table** statement with a pound sign (#), or by specifying the name prefix "*tempdb..*".

Temporary tables created with a pound sign are accessible only during the current SQL Server session and are deleted at the end of the session. The first 13 bytes of the table's name, including the pound sign (#), must be unique. SQL Server assigns the names of such tables a 17-byte numeric suffix.

Temporary tables created with the "*tempdb..*" prefix are stored in *tempdb* and can be shared among SQL Server sessions. SQL Server does not change the names of temporary tables created this way. The table exists either until you reboot SQL Server, or until its owner drops it using **drop table**. Temporary tables are not recoverable.

```
create table #authors
(au_id char (11))
```

creates a non-shareable temporary table.

```
create table tempdb..authors
(au_id char(11))
```

creates a temporary table which can be shared among SQL Server sessions.

You can use any tables or other objects that you have created without qualifying their names. You can also use objects created by the Database Owner without qualifying their names, as long as you have the appropriate permissions on them. These rules hold for all users, including the System Administrator and the Database Owner.

While table names must be unique for each user, different users can create tables of the same name. For example, a user named "jonah" and a user named "sally" can both create tables named *info*. Users who have permission on both of those tables will have to qualify them as *jonah.info* and *sally.info*. Sally will have to qualify all

references to Jonah's *info* table, although she can refer to her own simply as *info*.

create table Syntax

The syntax of the **create table** command is:

```
create table [database.[owner].]table_name
  (column_name datatype
      [default {constant_expression | user | null}]
      {[{identity | null | not null}]
      | [[constraint constraint_name]
          {{unique | primary key}
             [clustered | nonclustered]
          [with {fillfactor | max_rows_per_page} = x]
          [on segment_name]
           | references [[database.]owner.]ref_table
               [(ref_column)]
           | check (search_condition)}]}...

  | [constraint constraint_name]
      {{unique | primary key}
          [clustered | nonclustered]
           (column_name [{, column_name}...])
          [with {fillfactor | max_rows_per_page} = x]
          [on segment_name]
      | foreign key (column_name [{,
column_name}...])
          references [[database.]owner.]ref_table
             [(ref_column [{, ref_column}...])]
      | check (search_condition)}

  [{, {next_column | next_constraint}}...])
+
  [with max_rows_per_page = x][on segment_name]
```

The **create table** statement defines each column in the table. It provides the column name and datatype and specifies how each column handles null values. It specifies which column, if any, has the IDENTITY property. **create table** can also define column-level integrity constraints and table-level integrity constraints. Each table definition can have multiple constraints per column and per table.

For example, the **create table** statement for the *titles* table in the *pubs2* database is:

```
create table titles
(title_id tid,
title varchar(80) not null,
type char(12),
pub_id char(4) null,
price money null,
advance money null,
royalty int null,
total_sales int null,
notes varchar(200) null,
pubdate datetime
contract bit not null)
```

The following sections describe several different components of table definition: system-supplied datatypes, user-defined datatypes, null types, and IDENTITY columns. Defining integrity constraints for a table is described after those sections.

➤ ***Note***

The **on *segment_name*** extension to **create table** allows you to place your table on a segment, a name that points to a specific database device or a collection of database devices. Before creating a table on a segment, see a System Administrator or the Database Owner for a list of segments that you can use. Certain segments may be allocated to specific tables or indexes for performance reasons, or for other considerations.

Allowing Null Values

For each column, you can specify whether or not to allow null values. A null value is **not** the same as "zero" or "blank." NULL means no entry has been made, and usually implies "value unknown" or "value inapplicable." It indicates that the user did not make any entry, for whatever reason. For example, a null entry in the *price* column of the *titles* table does not mean that the book is being given away free, but that the price is not known or has not yet been set.

If the user fails to make an entry in a column defined with the keyword **null**, SQL Server supplies the value NULL. A column defined with the keyword **null** will also accept an explicit entry of NULL from the user, no matter what datatype it is. However, be careful when you enter null values in character columns. If you put the word "null" inside single or double quotes, SQL Server interprets the entry as a character string rather than as the value NULL.

If you omit **null** or **not null** in the **create table** statement, SQL Server uses the null mode defined for the database (by default, NOT NULL). For compatibility with the SQL standards, use the **sp_dboption** system procedure to set the **allow nulls by default** option to **true**.

For a column defined as NOT NULL, SQL Server insists on an entry. If there is no entry for a NOT NULL column, you'll get an error message.

Defaults, that is, values supplied automatically if no entry is made, can be used with both NULL and NOT NULL columns. A default counts as an entry. However, you cannot designate a NULL default for a NOT NULL column. You can specify null values as defaults using the **default** constraint of the **create table** statement or using the **create default** statement. The **default** constraint is described later in this chapter; **create default** is described in Chapter 12.

Defining columns as NULL provides a placeholder for data you may not yet know. For example, in the *titles* table, *price*, *advance*, *royalty*, and *total_sales* are set up to allow NULL.

However, *title_id* and *title* are not, because the lack of an entry in these columns would be meaningless and confusing. A price without a title would make no sense, whereas a title without a price would simply mean the price had not been decided upon yet or was not available.

In the **create table** statement, use the keywords **not null** when the information in the column is critical to the meaning of the other columns.

Using IDENTITY Columns

Each table can include a single IDENTITY column. IDENTITY columns store sequential numbers such as invoice numbers, employee numbers, or record numbers that are generated automatically by SQL Server. The value of the IDENTITY column uniquely identifies each row in a table.

You define an IDENTITY column by specifying the keyword **identity**, instead of **null** or **not null**, in the **create table** statement. (By definition, IDENTITY columns do not allow nulls.) IDENTITY columns must have a datatype of *numeric* and scale of 0.

The precision determines how large a value can be inserted into the column. The maximum possible column value is $10^{\text{PRECISION}} - 1$. Following is an example of a table whose IDENTITY column allows a maximum value of $10^5 - 1$, or 9,999:

```
create table sales_daily
    (row_id numeric(5,0) identity,
    stor_id char(4) not null)
```

You can create automatic IDENTITY columns by using the **auto identity** database option and the **size of auto identity** configuration parameter. To include IDENTITY columns in nonunique indexes, use the **identity in nonunique index** database option.

Creating IDENTITY Columns with User-Defined Datatypes

You can also use user-defined datatypes to create IDENTITY columns. The user-defined datatype must have an underlying type of *numeric* and a scale of 0.

If the user-defined datatype was created with the IDENTITY property, you do not have to repeat the **identity** keyword when creating the column. Here's an example of a user-defined datatype with the IDENTITY property:

```
sp_addtype ident, "numeric(5)", "identity"
```

Here's an IDENTITY column based on this type:

```
create table sales_monthly
    (row_id ident, stor_id char(4) not null)
```

If the user-defined type was created as **not null**, you must specify the **identity** keyword in the **create table** statement. You cannot create an IDENTITY column from a user-defined datatype that allows null values.

Referring to IDENTITY Columns with *syb_identity*

Once you have defined an IDENTITY column, you do not have to remember the actual column name. You can use the **syb_identity** keyword, qualified by the table name where necessary, in select, update, and delete operations on the table. For example, to select the row for which *row_id* equals 30, use the following query:

```
select * from sales_daily
    where syb_identity = 30
```

Generating Column Values

The first time you insert a row into a table, SQL Server assigns the IDENTITY column a value of 1. Each new row gets a column value one higher than the last. Transaction rollbacks, deletion of rows, the

identity grab size configuration parameter, and the manual insertion of data into the IDENTITY column can cause gaps in column values.

Server failures can also create gaps in IDENTITY column values.The size of these gaps, as a percentage of maximum table size, depends on the setting of the **identity burning set factor** configuration parameter. This parameter is set during installation and can be reset by the System Administrator.

Using Temporary Tables

If you use the pound sign (#) or "*tempdb..*" before the name of the table in the **create table** command, the new table is temporary.

There are two kinds of temporary tables:

- Tables which can be shared among SQL Server sessions.

  You create these shareable temporary tables by specifying *tempdb* as part of the table's name in the **create table** statement. For example:

  ```
  create table tempdb..my_temptbl
  ```

  SQL Server does not change the names of temporary tables created this way. The table exists either until you reboot SQL Server, or until its owner drops it using **drop table**.

- Tables that are accessible only by the current SQL Server session.

  Non-shareable temporary tables must begin with a pound (#) sign. You create a temporary table which cannot be shared by specifying only the table's name in the **create table** statement. For example:

  ```
  create table #my_temptbl
  ```

  SQL Server ensures that the temporary table name is unique on the current session. It truncates long temporary table names to 13 characters (including the pound sign), and pads short names to 13 characters, using underscores (_). SQL Server then appends a 17-digit numeric suffix that is unique for a SQL Server session. The table exists until the current session ends, or until its owner drops it using **drop table**.

If you do not use the pound sign or "*tempdb..*" before the table name, and you are not currently using *tempdb*, the table is created as a permanent table. A permanent table stays in the database until it is explicitly dropped by its owner.

Here is a statement that creates a non-shareable temporary table:

```
create table #myjobs
(task char(30),
start datetime,
stop datetime,
notes varchar(200))
```

You can use this table to keep a list of today's chores and errands, along with a record of when you start and finish and any comments you may have. This table and its data will vanish at the end of the current work session.

Temporary tables are not recoverable.

You can associate rules, defaults and indexes with temporary tables, but you cannot create views on temporary tables or associate triggers with them. You can use a user-defined datatype when creating a temporary table only if that datatype is in *tempdb..systypes*.

There are two ways to add a user-defined datatype, or any other object, to *tempdb*. To add an object for the current session only, execute **sp_addtype** while using *tempdb*. To add a user-defined datatype permanently, execute **sp_addtype** in *model* and then restart SQL Server so that *model* is copied to *tempdb*.

Creating Tables in Different Databases

As the **create table** syntax shows, you can create a table in a database other than the current one by qualifying the table name with the name of the other database. However, you must be an authorized user of the database in which you are creating the table, and you must have **create table** permission in it.

If you are using *pubs2* and there is another database called *newpubs*, you can create a table called *newtab* in *newpubs* like this:

```
create table newpubs..newtab (col1 int)
```

For **create table** to succeed, your *curread* session label must dominate the hurdle of the database where you want to create the table.

You cannot create other database objects—views, rules, defaults, stored procedures, or triggers—in a database other than the current one.

Defining Integrity Constraints for Tables

Transact-SQL provides two methods of maintaining data integrity in a database:

- Defining rules, defaults, indexes, and triggers
- Defining **create table** integrity constraints

Choosing one method over the other depends on your requirements. Integrity constraints offer the advantages of defining integrity controls in one step during the table creation process (as defined by the SQL standards) and of simplifying the process to create those integrity controls. However, integrity constraints are more limited in scope and less comprehensive than defaults, rules, indexes, and triggers.

For example, triggers provide more complex handling of referential integrity than those declared in **create table**. Also, the integrity constraints defined by a **create table** are specific for that table. Unlike rules or defaults, you cannot bind them to other tables, and you can only drop or change them using **alter table**. Constraints cannot contain subqueries or aggregate functions, even on the same table.

The two methods are not mutually exclusive. You can use integrity constraints along with defaults, rules, indexes, and triggers. This gives you the flexibility to choose the best method for your application. This section describes the **create table** integrity constraints. Defaults, rules, indexes, and triggers are described in later chapters.

You can create the following types of constraints:

- **unique** and **primary key** constraints require that no two rows in a table have the same values in the specified columns. In addition, a **primary key** constraint requires that there cannot be a null value in any row of the column.
- Referential integrity (**references**) constraints require that data being inserted in specific columns already have matching data in the specified table and columns.
- **check** constraints limit the values of data inserted into columns.

You can also enforce data integrity by restricting the use of null values in a column (the **null** or **not null** keywords) and by providing default values for columns (the **default** clause). See the previous section ‘‘Allowing Null Values,’’ for information about the **null** and **not null** keywords.

You can create error messages and bind them to constraints. Create messages with **sp_addmessage**, and bind them to constraints with **sp_bindmsg.** For more information, see **sp_addmessage** and **sp_bindmsg** in the *SQL Server Reference Manual*.

For information about any constraints defined for a table, use the **sp_helpconstraint** system procedure. **sp_helpconstraint** is described at the end of this chapter.

Specifying Table-Level or Column-Level Constraints

You can declare integrity constraints at the table or column level. The difference is syntactic. You should place column-level constraints after the column name and datatype, before the delimiting comma. You enter table-level constraints as separate comma-delimited clauses. SQL Server treats table-level and column-level constraints the same way; neither way is more efficient than the other.

However, you must declare constraints that operate on more than one column as table-level constraints. For example, the following **create table** statement has a **check** constraint that operates on two columns, *pub_id* and *pub_name*:

```
create table my_publishers
(pub_id      char(4),
pub_name     varchar(40),
constraint my_chk_constraint
    check(pub_id in ("1389", "0736", "0877")
        or pub_name not like "Bad News Books"))
```

You can declare constraints that operate on just one column as column-level constraints, but it is not required. For example, if the above **check** constraint just uses one column (*pub_id)*, you can place the constraint on that column:

```
create table my_publishers
(pub_id     char(4) constraint my_chk_constraint
        check(pub_id in ("1389", "0736", "0877")),
pub_name     varchar(40))
```

In either case, the **constraint** keyword and accompanying *constraint_name* are optional. The **check** constraint is described in a later section.

Specifying Default Column Values

Before defining any column-level integrity constraints, you can specify a default value for the column with the **default** clause. The **default clause** assigns a default value to a column in one step, as part of the **create table** statement. When a user does not enter a value for the column, SQL Server inserts the default value automatically.

You can use the following values with the **default** clause:

- *constant_expression* – specifies a constant expression to use as a default value for the column. It cannot include the name of any columns or other database objects, but you can include built-in functions that do not reference database objects. This default value must be compatible with the datatype of the column.
- **user** – specifies that SQL Server insert the user name as the default. The datatype of the column must be either *char(30)* or *varchar(30)* to use this default.
- **null** – specifies that SQL Server insert the null value as the default. You cannot define this default for columns that do not allow null values (using the **not null** keyword).

For example, this **create table** statement defines two column defaults:

```
create table my_titles
(title_id        char(6),
title            varchar(80),
price            money       default null,
total_sales      int         default 0)
```

You can include only one **default** clause per column of a table.

Using the **default** clause to assign defaults is simpler than the two-step Transact-SQL method. In Transact-SQL, you can use **create default** to declare the default value and then bind it to the column with **sp_bindefault**.

Specifying Unique and Primary Key Constraints

You can declare **unique** or **primary key** constraints to ensure that no two rows in a table have the same values in the specified columns. Both constraints create unique indexes to enforce this data integrity. However, **primary key** constraints are more restrictive than **unique** constraints. Columns with **primary key** constraints cannot contain a null value. You normally use a table's **primary key** constraint in conjunction with referential integrity constraints defined on other tables.

The definition of **unique** constraints in the SQL standards specifies that the column definition shall not allow null values. By default, SQL Server defines the column as not allowing null values (if you have not changed this using **sp_dboption**) when you omit **null** or **not null** keywords in the column definition. In Transact-SQL, you can define the column to allow null values along with the **unique** constraint, since the unique index used to enforce the constraint allows you to insert a null value.

➤ ***Note***

Do not confuse the unique and primary key integrity constraints with the information defined by the **sp_primarykey**, **sp_foreignkey**, and **sp_commonkey** system procedures. The unique and primary key constraints actually create indexes to define unique or primary key attributes of table columns. **sp_primarykey**, **sp_foreignkey**, and **sp_commonkey** define the logical relationship of keys (in the *syskeys* table) for table columns, which you enforce by creating indexes and triggers.

unique constraints create unique nonclustered indexes by default; **primary key** constraints create unique clustered indexes by default. You can declare either clustered or nonclustered indexes with either type of constraint.

For example, this **create table** statement uses a table-level **unique** constraint to ensure that no two rows have the same values in the *stor_id* and *ord_num* columns:

```
create table my_sales
(stor_id     char(4),
ord_num      varchar(20),
date         datetime,
unique clustered (stor_id, ord_num))
```

There can be only one clustered index on a table, so you can specify only one **unique clustered** or **primary key clustered** constraint.

You can use the **unique** and **primary key** constraints to create unique indexes (including the **with fillfactor, with max_rows_per_page,** and **on *segment_name*** options) when enforcing data integrity. However, indexes provide additional capabilities. For information about indexes and their options, including the differences between clustered and nonclustered indexes, see Chapter 11, “Creating Indexes on Tables.”

Specifying Referential Integrity Constraints

You can declare **referential integrity** constraints to require that data inserted into a "referencing" table which defines the constraint must have matching values in a "referenced" table. A referential integrity constraint is satisfied for either of the following conditions:

- If any column in the referencing table included with the constraint contains a null value
- If the columns in the referencing table included with the constraint match the corresponding columns in the referenced table

For example, this **create table** statement uses two referential integrity constraints:

```
create table my_salesdetail
(stor_id     char(4),
ord_num      varchar(20),
title_id     char(6)
        references my_titles(title_id),
qty          smallint,
constraint salesdet_constr
        foreign key (stor_id, ord_num)
        references my_sales (stor_id, ord_num))
```

The first constraint ensures that any row inserted into *my_salesdetail* must contain a value for *title_id* that matches a value for the *title_id* column of the *my_titles* table. *my_salesdetail* is the referencing table and *my_titles* is the referenced table. The second constraint (named *salesdet_constr*) ensures that values inserted for the columns *stor_id* and *ord_num* of one row match similarly named columns in a row of the *my_sales* table.

A table can include a referential integrity constraint on itself. You cannot delete rows or update column values from a referenced table that match values in a referencing table. Also, you cannot drop the referenced table until the referencing table is dropped or the referential integrity constraint is removed.

Table-level **referential integrity constraints** must include the **foreign key** clause and a list of one or more column names. Column names in the **references** clause are optional only if the columns in the referenced table are designated as a primary key through a **primary key** constraint.

Any referenced columns you specify must be constrained by a unique index in that table. You can create that unique index using either the **unique** or **primary key** constraint or the **create index** statement.

Also, the datatypes of the referencing table columns must exactly match the datatype of the referenced table columns. For example:

```
create table test_type
(col1 char(4) not null
    references publishers(pub_id),
col2 varchar(20) not null)
```

The datatype of *col1* in the referencing table (*test_type*) matches the datatype of *pub_id* in the referenced table (*publishers*).

You must have **references** permission on the referenced table to use referential integrity constraints. For information about permissions, see the *Security Features User's Guide.*

Referential integrity constraints provide a simpler way to enforce data integrity when compared to creating triggers. However, triggers provide additional capabilities to enforce referential integrity between tables. For information about triggers, see Chapter 15.

Specifying Check Constraints

You can declare a **check** constraint to limit what values users can insert into a column of a table. A **check** constraint specifies a *search_condition* which any value must pass before it is inserted into the table. A *search_condition* can include:

- A list of constant expressions introduced with **in**
- A range of constant expressions introduced with **between**
- A set of conditions introduced with **like**, which may contain wildcard characters

An expression can include arithmetic operations and Transact-SQL built-in functions. The *search_condition* cannot contain subqueries, a set function specification, or a target specification.

For example, this **create table** statement ensures that only certain values can be entered for the *pub_id* column:

```
create table my_new_publishers
(pub_id      char(4)
       check (pub_id in ("1389", "0736", "0877",
            "1622", "1756")
       or pub_id like "99[0-9][0-9]"),
pub_name     varchar(40),
city         varchar(20),
state        char(2))
```

If the check constraint is a column-level check constraint, it can only reference the column on which it is defined, it cannot reference other columns in the table. Table-level check constraints can reference any columns in the table. **create table** allows multiple **check** constraints in a column definition.

How to Design and Create a Table

This section gives an example of a **create table** statement you can use to create a practice table of your own. If you do not have **create table** permission, see a System Administrator or the owner of the database in which you are working.

Creating a table usually implies creating indexes, defaults, and rules to go with it. Custom datatypes, triggers, and views are frequently involved, too.

Of course, you can create a table, input some data, and work with it for a while before you create indexes, defaults, rules, triggers, or views. This gives you a chance to see what kind of transactions are most common and what kind of data is frequently entered.

On the other hand, it is often most efficient to design a table and all the components that go with it at once. Here is an outline of the steps you go through. You might find it easiest to sketch your plans on paper before you actually create a table and its accompanying objects.

1. Decide what columns you need in the table, and the datatype, length, precision, and scale, for each.
2. Create any new user-defined datatypes **before** you define the table they are to be used in.
3. Decide which column, if any, should be the IDENTITY column.
4. Decide which columns should accept null values, and which should not.
5. Decide what integrity constraints or column defaults, if any, you need to add to the columns of the table. This also means deciding when to use column constraints and defaults instead of defaults, rules, indexes, and triggers to enforce data integrity.
6. Decide whether or not you need defaults and rules, and if so, where and what kind. Consider the relationship between the NULL and NOT NULL status of a column and defaults and rules.

7. Decide what kind of indexes you need and where. Indexes are discussed in Chapter 11.
8. Create the table and its indexes with the **create table** and **create index** commands.
9. Create new defaults and rules you need with the **create default** and **create rule** commands. These commands are discussed in Chapter 12.
10. Bind any defaults and rules you need with the system procedures **sp_bindefault** and **sp_bindrule**. If there were any defaults or rules on a user-defined datatype that you used in a **create table** statement, they are automatically in force. These system procedures are discussed in Chapter 14.
11. Create triggers with the **create trigger** command. Triggers are discussed in Chapter 15.
12. Create views with the **create view** command. Views are discussed in Chapter 9.

Make a Design Sketch

The table called *friends_etc* is used in subsequent chapters to show how to create indexes, defaults, rules, triggers, and so forth. It can hold names, addresses, telephone numbers, and personal information about your friends. It does not define any column defaults or integrity constraints, so as to not conflict with those objects.

If you are planning to follow the examples and create all the objects that go with *friends_etc* yourself, check with a System Administrator or the Database Owner. That person should make sure that if the table has already been created by another user, it and the indexes, defaults, rules, and triggers that go with it have been dropped, so that there will be no conflict when you create the objects.

Here is a chart showing the proposed structure of the table and the indexes, defaults, and rules that will go with each column.

Table 7-1: Sample table design

Column	Datatype	Null?	Index	Default	Rule
pname	*nm*	Not null	*nmind*(composite)		
sname	*nm*	Not null	*nmind*(composite)		

Table 7-1: Sample table design (continued)

Column	Datatype	Null?	Index	Default	Rule
address	*varchar(30)*	Null			
city	*varchar(30)*	Not null		*citydflt*	
state	*char(2)*	Not null		*statedflt*	
zip	*char(5)*	Null	*zipind*	*zipdflt*	*ziprule*
phone	*p#*	Null			*phonerule*
age	*tinyint*	Null			*agerule*
bday	*datetime*	Not null		*bdflt*	
sex	*bit*	Not null		*sexdflt*	
debt	*money*	Not null		*sexdflt*	
notes	*varchar(255)*	Null			

Create the User-Defined Datatypes

The first two columns are for personal name and surname. They are defined as *nm* datatype. Before you create the table, you need to create the datatype. The same is true of the *p#* datatype for the *phone* column.

The *nm* datatype allows for a variable-length character entry with a maximum of 30 bytes. The *p#* datatype allows for a *char* datatype with a fixed-length size of 10 bytes.

Enter the user datatype definitions for *nm* and *p#* like this:

```
execute sp_addtype nm, "varchar(30)"
execute sp_addtype p#, "char(10)"
```

Choose the Columns That Accept Null Values

Except for columns that are assigned user-defined datatypes, each column has an explicit NULL or NOT NULL entry. Remember that you do not need to specify NOT NULL in the table definition, because it is the default. This table design specifies NOT NULL explicitly, for readability.

The NOT NULL default means that some entry is required, for example for the two name columns in this table. The other data is meaningless without the names. In addition, the *sex* column must be NOT NULL because you cannot use NULL with *bit* columns.

If a column is designated NULL and a default is bound to it, the default value, rather than NULL, is entered when no other value is given on input. If a column is designated NULL and a rule is bound to it that does not specify NULL, the column definition overrides the rule when no value is entered for the column. Columns can have both defaults and rules. The relationship between these two is discussed in a later chapter.

Define the Table

Now you are ready to write the **create table** statement:

```
create table friends_etc
(pname       nm             not null,
sname        no             not null,
address      varchar(30)    null,
city         varchar(30)    not null,
state        char(2)        not null,
postalcode   char(5)        null,
phone        p#             null,
age          tinyint        null,
bday         datetime       not null,
sex          bit            not null,
debt         money          not null,
notes        varchar(255)   null
```

Now you have columns defined for personal name and surname, address, city, state, postal code, telephone number, age, birthday, sex, debt information, and notes. Other chapters describe how to create the rules, defaults, indexes, triggers, and views that involve this table.

Creating New Tables from Query Results: *select into*

You can use the **select into** clause to **select into** a permanent table only if the **select into/bulkcopy** database option is set to **on.** A System Administrator can turn on this option using the **sp_dboption** system procedure. To see whether this option is on, execute the system procedure **sp_helpdb**.

Here is what the command and its results look like when the option is set to **on**:

```
sp_helpdb pubs2
```

```
name       db_size owner   dbid    created      status
---------  ------- ------  -----   -----------  ------------
pubs       2 MB    sa      5       Jun  3 1988  select into
                                                /bulkcopy
(1 row affected)

device              size         usage
-----------------   ---------    --------------
master              2 MB         data and log

(1 row affected)
```

If the option is not on, the report from **sp_helpdb** indicates this. Only the System Administrator or the Database Owner can set the database options.

If the **select into/bulkcopy** database option is on, you can use the **select into** clause to build a new permanent table without using a **create table** statement. You can **select into** a temporary table, even if the option is not on.

➤ ***Note***

Since **select into** is a nonlogged operation, use **dump database** to back up your database following a **select into**. Do not use **dump transaction**, since a dump of the log following a nonlogged operation is not usable with **load transaction**.

Unlike a view that displays a portion of a table, a table created with **select into** is a separate, independent entity. See Chapter 9 for details on views.

The new table is based on the columns you specify in the select list, the tables you name in the **from** clause, and the rows you choose in the **where** clause. The name of the new table must be unique in the database, and must conform to the rules for identifiers.

A **select** statement with an **into** clause allows you to define a table and put data into it, based on existing definitions and data, without going through the usual data definition process.

The following example shows a **select into** statement and its results. A table called *newtable* is created, using two of the columns in the four-column table *publishers*. Because this particular statement includes no **where** clause, data from all of the rows (but only two of the columns) of *publishers* is copied into *newtable*.

```
select pub_id, pub_name
into newtable
from publishers

(3 rows affected)
```

SQL Server's message "3 rows affected" refers to the three rows inserted into *newtable*. Here's what *newtable* looks like:

```
select *
from newtable

pub_id  pub_name
------  ------------------------------------
0736    New Age Books
0877    Binnet & Hardley
1389    Algodata Infosystems

(3 rows affected)
```

The new table contains the results of the **select** statement. It becomes part of the database, just like its parent table. The **into** clause is useful for creating test tables, new tables as copies of existing tables, and for making several smaller tables out of one large table.You can also use **select into** to create a skeleton table with no data by putting a false condition in the **where** clause. For example:

```
select *
into newtable2
from publishers
where 1=2

(0 rows affected)

select *
from newtable2

pub_id    pub_name          city      state
------    --------------    --------  -----

(0 rows affected)
```

No rows are inserted into the new table, because 1 never equals 2.

You can also use **select into** with aggregate functions to create tables with summary data:

```
select type, "Total_amount" = sum(advance)
into #whatspent
from titles
group by type

(6 rows affected)
```

```
select * from #whatspent
 type         Total_amount
 ------------ ------------------------
 UNDECIDED                       NULL
 business                   25,125.00
 mod_cook                   15,000.00
 popular_comp               15,000.00
 psychology                 21,275.00
 trad_cook                  19,000.00
(6 rows affected)
```

You must always supply a column name for any column in the **select into** result table that results from an aggregate function or any other expression, such as performing arithmetic (***amount* * 2**), **concatenation** (**lname + fname**), or use of SQL Server built-in functions (**lower(lname)**). Here's an example using concatenation:

```
select au_id,
    "Full_Name" = au_fname + ' ' + au_lname
into #g_authortemp
from authors
where au_lname like "G%"

(3 rows affected)

select * from #g_authortemp

 au_id       Full_Name
 ----------- -------------------------
 213-46-8915 Marjorie Green
 472-27-2349 Burt Gringlesby
 527-72-3246 Morningstar Greene
(3 rows affected)
```

Selecting an IDENTITY Column

To select an IDENTITY column into a new table, include the column name (or the **syb_identity** keyword) in the select statement's column list. The new column inherits the IDENTITY property, unless any of the following conditions is true:

- The IDENTITY column is selected more than once.
- The IDENTITY column is selected as part of an expression.
- The **select** statement contains a **group by** clause, aggregate function, **union** operator, or join.

Adding a New IDENTITY Column with *select into*

To define a new IDENTITY column in a **select into** statement, add the column definition before the **into** clause. Note that the definition includes the column's precision but not its scale:

```
select column_list
identity_column_name = identity(precision)
into table_name
from table_name
```

You cannot use **select into** to create a new table with multiple IDENTITY columns. If the **select** statement includes both an existing IDENTITY column and a new IDENTITY specification, the statement fails.

For more information about IDENTITY columns, see **select** and "IDENTITY Columns" in the *SQL Server Reference Manual*.

Dropping Tables

The command to remove a table from a database is **drop table**. Its syntax is:

```
drop table [[database.]owner.] table_name
     [, [[database.]owner.] table_name]...
```

When you issue this command, SQL Server removes the specified tables from the database, together with their contents and all the indexes and privileges associated with them. Rules or defaults bound to the table are no longer bound, but are otherwise not affected.

You must be the owner of a table in order to drop it. However, no one can drop a table while it is in use, that is, being read or written by a user or a front-end program. The **drop table** command cannot be used on any of the system tables, either in the *master* database or in a user database.

As the syntax indicates, you can drop a table in another database as long as you are the table owner.

If you **delete** all the rows in a table or use the **truncate table** command on it, the table still exists until you **drop** it.

drop table and **truncate table** permission cannot be transferred to other users.

Altering Existing Tables

If you change your mind about a table's structure after you have used it for a while and decide you need to modify the way the table is put together, you have these options:

- You can add columns and constraints, drop constraints, or change column default values using the **alter table** command.
- You can change the name of a table, a column, or any other database object with the system procedure **sp_rename**.

Changing Table Structures: *alter table*

The **alter table** command allows you to make these changes to existing tables:

- Add columns (except *bit* datatype columns)
- Add constraints
- Drop constraints
- Replace the defaults defined for its columns

Here is the **alter table** syntax:

```
alter table [database.[owner].]table_name
   {add column_name datatype
       [default {constant_expression | user | null}]
       {[{identity | null}]
       | [[constraint constraint_name]
           {{unique | primary key}
               [clustered | nonclustered]
           [with {fillfactor | max_rows_per_page} = x]
           [on segment_name]
           | references [[database.]owner.]ref_table
               [(ref_column)]
           | check (search_condition)}]}...
       {[, next_column]}...
```

```
| add {[constraint constraint_name]
    {unique | primary key}
        [clustered | nonclustered]
        (column_name [{, column_name}...])
        [with {fillfactor | max_rows_per_page} = x]
        [on segment_name]
| foreign key (column_name [{, column_name}...])
        references [[database.]owner.]ref_table
           [(ref_column [{, ref_column}...])]
    | check (search_condition)}

| drop constraint constraint_name

| replace column_name
    default {constant_expression | user | null}}
```

The number of columns in a table cannot exceed 250, whether they are added with an **alter table** statement or defined with the original **create table** statement.

A table can have only one IDENTITY column with a datatype of *numeric* and a scale of zero. When you add an IDENTITY column with the **alter table** statement, SQL Server assigns a unique, sequential value to each existing row.

All other columns that you add must allow null values. This is because when the new column is added to the existing rows, there must be some value for it. Specify **null** when adding any column other than the IDENTITY column.

➤ ***Note***

If stored procedures using **select *** reference a table that has been altered, the procedure, even if you use the **with recompile** option, will not pick up any new columns you may have added to the table. You must drop the procedure and re-create it.

For example, you can add a column to the *friends_etc* table as follows:

```
alter table friends_etc
    add country varchar(20) null
```

You can later add one or more integrity constraints on the new column (or any other column) of *friends_etc*:

```
alter table friends_etc
    add constraint no_old_country
        check (country not in ("GDR", "E. Germany",
                    "East Germany"))
```

When you no longer need a constraint, you can drop it:

```
alter table friends_etc
    drop constraint no_old_country
```

To drop constraints, you must specify the *constraint_name*. To determine the names of constraints defined for a table, use the **sp_helpconstraint** system stored procedure. **sp_helpconstraint** is described in "Using sp_helpconstraint on Tables" on page 7-39.

alter table also allows you to change the default value defined for a column (or add a column default if one does not exist). For example:

```
alter table friends_etc
    replace country default "USA"
```

For information about column defaults and integrity constraints, see the section "Defining Integrity Constraints for Tables" on page 7-18.

Renaming Tables and Other Objects

To rename tables and other database objects—views, indexes, rules, defaults, procedures, and triggers—use the system procedure **sp_rename**. You must be the owner of an object in order to rename it.

To rename the database, use the system procedure **sp_renamedb**. See the *SQL Server Reference Manual* for information about **sp_renamedb**.

The syntax of **sp_rename** is:

```
sp_rename objname, newname
```

For example, to change the name of *friends_etc* to *infotable*, type this:

```
sp_rename friends_etc, infotable
```

You can use **sp_rename** to rename other objects as well: columns, defaults, rules, procedures, views, triggers, check constraints, referential integrity constraints, and user datatypes. If you are renaming a column, use this syntax:

```
sp_rename "table.column", newcolumnname
```

Note that you must leave off the table name prefix from the new column name, or the new name won't be accepted. To change the name of an index, use this syntax:

```
sp_rename "table.index", newindexname
```

Again, do not include the table name in the new name.

Here is how to change the name of the user datatype *tid* to *t_id*:

```
exec sp_rename tid, "t_id"
```

You cannot change the name of system objects or system datatypes. The object whose name you are changing must be in the current database. You may change the names only of those objects you own. However, the Database Owner may change the name of any user's objects.

A user may change the names only of those objects he or she owns. The Database Owner may change the name of any user's objects.

Effect of Renaming on Dependent Objects

Procedures, triggers, and views that depend on an object whose name has been changed work fine until they are recompiled. However, recompilation takes place for many reasons and without notification to the user, for example, if a database is loaded, or if a user drops and re-creates a table or drops an index.

When SQL Server recompiles the procedure, trigger, or view, it will no longer work. The user must change its text to reflect the new object name. Also, the old object name will appear in query results until the procedure, trigger, or view has been changed and recompiled. The safest course is to change the definitions of any **dependent** objects when you execute **sp_rename**. You can get a list of dependent objects using the **sp_depends** system procedure.

Assigning Permissions to Users

The SQL commands **grant** and **revoke** control the SQL Server command and object protection system. You can give various kinds of permissions to users, groups, and roles with the **grant** command and rescind them with the **revoke** command. **grant** and **revoke** are used to give users permission to:

- Create databases
- Create objects in a database
- Access tables, views, and columns
- Execute stored procedures

Some commands can be used at any time by any user, with no permissions required. Others can be used only by users of certain

status (for example, only by a System Administrator) and are not transferable.

The ability to assign permissions for the commands that can be granted and revoked is determined by each user's status (as System Administrator, Database Owner, or database object owner) and by whether or not a particular user has been granted a permission with the option to grant that permission to other users.

The owner of a database does not automatically receive permissions on objects that are owned by other users. But a Database Owner or System Administrator can always acquire any permission by assuming the identity of the object owner with the **setuser** command and then writing the appropriate **grant** or **revoke** statement.

Two kinds of permissions can be assigned with **grant** and **revoke**: **object access permissions** and **object creation permissions**.

Object access permissions regulate the use of certain commands that access certain database objects. For example, you must explicitly be granted permission to use the **select** command on the *authors* table. Object access permissions are granted and revoked by the owner of the object.

The following statement grants Mary and Joe the object access permission to **insert** into and **delete** from the *titles* table:

```
grant insert, delete
on titles
to mary, joe
```

Object creation permissions regulate the use of commands that create objects. These permissions can be granted only by a System Administrator or Database Owner.

The following statement revokes object creation permission to create tables and rules in the current database from Mary:

```
revoke create table, create rule
from mary
```

For complete information about using **grant** and **revoke** for object access permissions and object creation permissions, see the *Security Features User's Guide*.

Getting Information About Databases and Tables

SQL Server provides several system stored procedures to get information about databases, tables, and other database objects. This

section describes four of them: **sp_help**, **sp_helpdb**, **sp_helpconstraint**, and **sp_spaceused**.

For complete information on the system procedures, see the *SQL Server Reference Manual*.

Using *sp_help* on Database Objects

The system procedure **sp_help** reports information about a specified database object (that is, any object listed in *sysobjects*), about a specified datatype (listed in *systypes*), or about all objects and datatypes in the current database.

The syntax of **sp_help** is:

sp_help [*objname*]

Here is the output for the *publishers* table:

```
Name                           Owner       Type
-------------------------    ----------- --------
publisher                      dbo         user table

Data_located_on_segment            When_created
------------------------------     --------------------
default                            Jan  1 1900 12:00AM

Column_name Type     Length   Prec  Scale
----------- -------  ------   ----- -----
pub_id      char          4    NULL  NULL
pub_name    varchar      40    NULL  NULL
city        varchar      20    NULL  NULL
state       char          2    NULL  NULL

Nulls       Default_name   Rule_name     Identity
-----       -------------  ---------     --------
    0       NULL           NULL                0
    1       NULL           NULL                0
    1       NULL           NULL                0
    1       NULL           NULL                0

index_name     index_description                    index_keys
-------------- ------------------------------------ ----------
pubind         clustered, unique located on default pub_id

(1 row affected)
```

```
keytype  object     related_object  object_keys
related_keys
-------  ---------- --------------  ----------------------------
-------------------------
primary  publishers -- none --      pub_id, *, *, *, *, *, *, *
*, *, *, *, *, *, *, *
foreign  titles     publishers      pub_id, *, *, *, *, *, *, *
pub_id, *, *, *, *, *, *, *

(return status = 0)
```

If you execute **sp_help** without supplying an object name, the resulting report shows a brief listing of each object in *sysobjects*, giving its name, owner, and object type. Also shown is each user-defined datatype in *systypes* and its name, storage type, length, whether null values are allowed, and the names of any defaults or rules bound to it. The report also notes whether any primary or **foreign key** columns have been defined for a table or view with the system procedures **sp_primarykey** or **sp_foreignkey**.

sp_help lists any indexes on a table, including indexes created by defining unique or primary key constraints of the **create table** or **alter table** statements. However, it does not describe any information about the integrity constraints defined for a table. You must use **sp_helpconstraint** for information about any integrity constraints.

Using *sp_helpdb* on Databases

The system procedure **sp_helpdb** reports information about a specified database, or about all the databases on SQL Server. It reports on the name, size, and usage of each fragment you have assigned to the database with **create** or **alter database**. Its syntax is:

sp_helpdb [*dbname*]

Here is how you would get a report on *pubs2*:

```
sp_helpdb pubs2
name    db_size   owner  dbid created           status
-----   -------   ------ ---- ---------------   -------------
pubs2   2 MB      sa     4    Jan 10 1988       no options set

(1 row affected)
```

```
device             size         usage
-----------------  -----------  --------------
pubsdev            2 MB         data + log

(1 row affected)
```

Using *sp_helpconstraint* on Tables

The system procedure **sp_helpconstraint** reports information about any integrity constraints specified for a table. This information includes the constraint name and the definition of the default, unique or **primary key constraint**, referential constraint, or check constraint. Its syntax is:

sp_helpconstraint *objname* [, detail]

By default, **sp_helpconstraint** prints just the name and definition of the integrity constraint. If you specify the **detail** option with this system procedure, it also returns information about the constraint's user or error messages.

For example, assume table *states* is defined as follows:

```
create table states
(rank smallint,
abbrev char(2),
name varchar(20) null,
population int check (population > 1000000),
constraint stateconstr primary key (rank, abbrev))
```

You can execute **sp_helpconstraint** to report its constraints:

```
sp_helpconstraint states

name                      defn
------------------------  ---------------------------------------
states_popula_1088006907  CHECK (population > 1000000)
stateconstr               PRIMARY KEY INDEX (rank, abbrev):
                          CLUSTERED,FOREIGN REFERENCE

(3 rows affected, return status = 0)
```

Using *sp_spaceused* on Tables

You can find out how much space a table uses with the system procedure **sp_spaceused**. Its syntax is:

sp_spaceused [*objname*]

This system procedure also works for indexes, which are described in Chapter 11, “Creating Indexes on Tables”. It computes and displays the number of rows and data pages used by a table or a clustered or nonclustered index. Here is how to get a report on the space used by the *titles* table:

```
sp_spaceused titles
name   rows   reserved  data  index_size  unused
------- ----- ---------- ----- ---------  ------
titles  18    48 KB      6 KB  4 KB       38 KB

(0 rows affected)
```

If no object name is given as a parameter, **sp_spaceused** displays a summary of space used by all database objects.

8 Adding, Changing, and Deleting Data

Once you have created a database, tables, and indexes, you'll want to put data into the tables and work with it—adding, changing, and deleting data as necessary.

This chapter discusses:

- A general overview of the ways to modify data
- The rules associated with entering data for certain datatypes
- How to add new data to tables
- How to change data that already exists in tables
- How to change *text* data
- How to delete data from tables
- How to delete (or truncate) all rows from a table

What Choices Are Available to Modify Data?

The **insert** command lets you add new rows to the database. The **update** command lets you change existing rows in the database. The **delete** command lets you remove rows from the database. The **writetext** command lets you add or change *text* and *image* data without writing lengthy changes in the system's transaction log.

Such operations are collectively called **data modification statements**. The **truncate table** command, which deletes all the rows in a table, is also discussed in this chapter. Another method of adding data to a table is to transfer it from a file using the bulk copy utility program **bcp**. For information about these facilities, see the *SQL Server Reference Manual* and the SQL Server utilities manual for your operating system.

You can modify data, using the **insert**, **update**, or **delete** statements, in only one table per statement. However, the modifications you make can be based on data in other tables, even those in other databases. This is a Transact-SQL enhancement to standard versions of SQL.

The data modification commands work on views as well as on tables, with some restrictions. See Chapter 9, "Views: Limiting Access to Data," for details.

Permissions

Data modification commands are not necessarily available to everyone. The Database Owner and the owners of database objects use the **grant** and **revoke** commands to decide who has access to which data modification functions.

Permissions or privileges can be granted to individual users, groups, or the public for any combination of the data modification commands. Permissions are discussed in the *Security Features User's Guide.*

Referential Integrity

insert, **update**, **delete**, **writetext**, and **truncate table** all allow you to change the data in the database. However, if you change data in one table without changing related data in other tables, disparities may develop.

For example, if you discover that the *au_id* entry for Sylvia Panteley is incorrect and change it in the *authors* table, you must also change it in the *titleauthor* table, and in any other table in the database with a column containing that value. If you do not, you will never be able to find information such as the names of Ms. Panteley's books, because it will be impossible to make joins on her *au_id* column.

The general problem of keeping data modifications consistent throughout all tables in a database is called referential integrity. One way to deal with it is to create special procedures called triggers that automatically go into effect when you give **insert**, **update**, and **delete** commands for particular tables or columns (the **truncate table** command is not caught by a trigger). Another way is to define referential integrity constraints for the table. Triggers are discussed in Chapter 15, "Triggers: Enforcing Referential Integrity"; integrity constraints are discussed in Chapter 7, "Creating Databases and Tables."

Transactions

A copy of the old and new state of each row affected by each data modification statement, except for **writetext**, is written to the transaction log. This means that if you begin a transaction by issuing the **begin transaction** command, realize you have made a mistake, and roll the transaction back, the database can be restored to its previous condition.

➤ ***Note***

Changes made on a remote SQL Server by means of a remote procedure call (RPC) cannot be rolled back.

The default mode of operation for **writetext** does **not** log the transactions. This avoids filling up the transaction log with the extremely long blocks of data that *text* and *image* fields may contain. The **with log** option to the **writetext** command must be used to log changes made with this command.

A more complete discussion of transactions appears in Chapter 17, "Transactions: Maintaining Data Consistency and Recovery."

Using the Sample Database

If you follow the examples in this chapter on your own screen, you will probably want to start with a clean copy of the *pubs2* database and return it to that state when you are finished. See a System Administrator for help in getting a clean copy of the *pubs2* database.

If you are starting with a clean *pubs2* database, you can prevent any changes you make from becoming permanent by enclosing all the statements you enter inside a transaction, and then aborting the transaction when you are finished with this chapter. Start the transaction by typing:

```
begin tran modify_pubs2
```

This transaction is named *modify_pubs2*. You can cancel the transaction at any time and return the database to the condition it was in before you began the transaction by typing:

```
rollback tran modify_pubs2
```

Datatype Entry Rules

Several of the SQL Server-supplied datatypes have special rules for entering and searching for data. These rules are reviewed in the following subsections. For more information on datatypes, see Chapter 7, "Creating Databases and Tables."

char, *nchar*, *varchar*, *nvarchar*, and *text*

Do not forget that all *character*, *text*, and *datetime* data must be enclosed in single or double quotes when it is input and when you are searching for it. Use single quotes if the **quoted_identifier** option is set **on**. If you use double quotes, SQL Server treats the text as an **identifier**. See the *SQL Server Reference Manual* for details on inserting *text* data.

If you enter strings that are longer than the specified length of a *char*, *nchar*, *varchar*, or *nvarchar* column, the entry is truncated. Set the **string_rtruncation** option **on** to receive a warning message when this occurs.

There are two ways to specify literal quotes within a character entry. The first method is to use two quotes. For example, if you have begun a character entry with a single quote and wish to include a single quote as part of the entry, use two single quotes: 'I don' 't understand.' With double quotes: "He said, ""It's not really confusing."""

The second method is to enclose a quote in the opposite kind of quotation mark. In other words, surround an entry containing a double quote with single quotes, or vice versa. For example: 'George said, "There must be a better way." '

To enter a character string longer than the width of your screen, enter a backslash (\) before going to the following line.

The **like** keyword and wildcard characters described in Chapter 2, "Queries: Selecting Data from a Table," can be used when searching for *character*, *text*, and *datetime* data.

See "Datatypes" in the *SQL Server Reference Manual* for information about trailing blanks in *character* data.

datetime and *smalldatetime*

Display and entry formats for *datetime* data provide a wide range of date output formats, and recognize a wide variety of input formats as well. The display and entry formats are controlled separately. The default display format provides output that looks like "Apr 15 1987 10:23PM". The **convert** command provides options to display seconds and milliseconds and to display the date with other date-part orderings. See Chapter 10, "Using the Built-In Functions in Queries," for more information on displaying date values.

SQL Server recognizes a wide variety of data entry formats for dates. Case is always ignored, and spaces can occur anywhere between date parts. When you enter *datetime* and *smalldatetime* values, always enclose them in single or double quotes. (Use single quotes if the **quoted_identifier option** is set **on**; if you use double quotes SQL Server treats the entry as an identifier.)

SQL Server recognizes the two date and time portions of the data separately, so the time can precede or follow the date. Either portion can be omitted; SQL Server provides defaults, also described below. If both portions are omitted, the default date is January 1, 1900, 12:00:00:000AM.

For *datetime*, the earliest date you can use is January 1, 1753; the latest is December 31, 9999. For *smalldatetime*, the earliest date you can use is January 1, 1900; the latest is June 6, 2079. Dates earlier or later than these dates must be entered, stored, and manipulated as *char* or *varchar* values. SQL Server rejects all values it cannot recognize as dates between those ranges.

Entering Times

The order of time components is significant for the time portion of the data. Enter hours; then minutes; then seconds; then milliseconds; then AM, am, PM, or pm. 12AM is midnight, 12PM is noon. To be recognized as time, a value must contain either a colon or an AM / PM signifier. Note that *smalldatetime* is accurate only to the minute.

Milliseconds can be preceded either with a colon or a period. If preceded by a colon, the number means thousandths of a second. If preceded by a period, a single digit means tenths of a second, two digits mean hundredths of a second, and three digits mean thousandths of a second. For example, '12:30:20:1' means 20 and one-thousandth of a second past 12:30; '12:30:20.1' means 20 and one-tenth of a second past 12:30.

Among the acceptable formats for time data are:

```
14:30
14:30[:20:999]
14:30[:20.9]
4am
4 PM
[0]4[:30:20:500]AM
```

Entering Dates

The **set dateformat** command allows a user to specify the order of the date parts (month, day, and year) when dates are entered as strings of numbers with separators. Changing the language with **set language** can also affect the format for dates, depending on the default date format for the language. The default language is *us_english*, and the default date format is *mdy*. See the **set** command in the *SQL Server Reference Manual* for more information.

➤ ***Note***

dateformat only affects those dates entered as numbers-with-separators, such as “4/15/90” or “20.05.88”. It does not affect dates where the month is provided in alphabetic format, such as “April 15, 1990”, or where there are no separators, such as “19890415”.

SQL Server recognizes three basic styles of date input. Each of the date formats shown below must be enclosed in quotes when it is used, and may be preceded or followed by a time specification, as described above.

- The month is entered in alphabetic format.
  - Month can be a 3-character abbreviation, or the full month name, as given in the specification for the current language.
  - Commas are optional.
  - Case is ignored.
  - If you specify only the last two digits of the year, values less than 50 are interpreted as “20yy”, and values of 50 or greater are interpreted as “19yy”.
  - You must type the century only when the day is omitted, or when you need a century other than the default, as described above.
  - If the day is missing, it defaults to the first day of the month.
  - When you specify the month in alphabetic form, the **dateformat** (see the **set** command) setting is always ignored.
  - Valid formats for specifying the date alphabetically are:

```
Apr[il] [15][,] 1988
Apr[il] 15[,] [19]88
Apr[il] 1988 [15]
```

```
[15] Apr[il][,] 1988
15 Apr[il][,] [19]88
15 [19]88 apr[il]
[15] 1988 apr[il]

1988 APR[IL] [15]
[19]88 APR[IL] 15
1988 [15] APR[IL]
```

- The month is entered in numeric format, in a string with slash (/), hyphen (-) or period (.) separators.
  - The month, day, and year must be specified.
  - The strings must be in the form:

    <num> <sep> <num> <sep> <num> [<time spec>]

    or:

    [<time spec>] <num> <sep> <num> <sep> <num>
  - The interpretation of the values of the date parts depends on the **dateformat** setting. If the ordering does not match the setting, either the values will not be interpreted as dates, because values are out of range, or the values will be misinterpreted. For example, "12/10/08" could be interpreted as one of six different dates, depending on the **dateformat** setting. See the **set** command for more information.
  - To enter "April 15, 1988" in *mdy* **dateformat**, you can use these formats:

    ```
    [0]4/15/[19]88
    [0]4-15-[19]88
    [0]4.15.[19]88
    ```
- The other entry orders are shown below with "/" as separators; hyphens or periods can also be used:

  ```
  15/[0]4/[19]88 (dmy)
  [19]88/[0]4/15 (ymd)
  [19]88/15/[0]4 (ydm)
  [0]4/[19]88/15 (myd)
  15/[19]88/[0]4 (dym)
  ```
- The date is given as an unseparate 4-, 6-, or 8-digit string, or as an empty string, or only the time value, but no date value, is given.
  - The **dateformat** is always ignored with this entry format.
  - If 4 digits are given, the string is interpreted as the year, and the month is set to January, the day to the first of the month. The century cannot be omitted.

- 6- or 8-digit strings are always interpreted as *ymd*; the month and day must always be 2 digits. This format is recognized: [19]880415.
- An empty string ("") or missing date is interpreted as the base date, January 1, 1900. For example, a time value like "4:33" without a date is interpreted as "January 1, 1900, 4:33AM".

Searching for Dates and Times

You can use the **like** keyword and wildcard characters with *datetime* and *smalldatetime* data as well as with *char, nchar, varchar, nvarchar,* and *text*. When you use **like** with *datetime* or *smalldatetime* values, SQL Server converts the dates to the standard *datetime* format, and then to *varchar*. Since the standard display format doesn't include seconds or milliseconds, you cannot search for seconds or milliseconds with **like** and a match pattern. Use the **type conversion function**, **convert**, to search for seconds and milliseconds.

It is a good idea to use **like** when you search for *datetime* or *smalldatetime* values, since *datetime* or *smalldatetime* entries may contain a variety of date parts. For example, if you insert the value "9:20" into a column named *arrival_time*, the clause:

```
where arrival_time = '9:20'
```

would not find it because SQL Server converts the entry into "Jan 1, 1900 9:20AM." However, the following clause would find it:

```
where arrival_time like '%9:20%'
```

If you are using **like**, and the day of the month is less than 10, you must insert two spaces between the month and day to match the *varchar* conversion of the datetime value. Similarly, if the hour is less than 10, the conversion places two spaces between the year and the hour. The clause **like May 2%** with one space between "May" and "2", will find all dates from May 20 through May 29, but not May 2. You do not need to insert the extra space with other date comparisons, only with **like**, since the datetime values are converted to *varchar* only for the **like** comparison.

binary, *varbinary*, and *image*

When *binary, varbinary,* or *image* data is entered or searched for, it must be preceded with "0x". For example, to enter "FF", type "0xFF".

If you enter strings that are longer than the specified length of a *binary* or *varbinary* column, the entry is truncated without warning.

A length of 10 for a *binary* or *varbinary* column means 10 bytes, each storing 2 hexadecimal digits.

When you create a default on a *binary* or *varbinary* column, precede it with *"0x"*.

See "Datatypes" in the *SQL Server Reference Manual* for information on trailing zeroes in hexadecimal values.

money and *smallmoney*

Monetary values entered with E notation are interpreted as *float*. This may cause an entry to be rejected or to lose some of its precision when it is stored as a *money* or *smallmoney* value.

money and *smallmoney* values can be entered with or without a preceding currency symbol, such as the dollar sign ($), yen sign (¥) or pound sterling sign (£). To enter a negative value, place the minus sign after the currency symbol. Do not include commas in your entry.

You cannot enter *money* or *smallmoney* values with commas, although the default print format for *money* or *smallmoney* data places a comma after every three digits. When *money* or *smallmoney* values are displayed, they are rounded up to the nearest cent. All the arithmetic operations except modulo are available with *money*.

float, real, and *double precision*

You enter the approximate numeric types—*float*, *real*, and *double precision*—as a mantissa followed by an optional exponent. The mantissa can include a positive or negative sign and a decimal point. The exponent, which begins after the character "e" or "E", can include a sign but not a decimal point.

To evaluate approximate numeric data, SQL Server multiplies the mantissa by 10 raised to the given exponent. Here are some examples of *float*, *real*, and *double precision* data:

Table 8-1: Evaluating numeric data

Data Entered	Mantissa	Exponent	Value
10E2	10	2	$10 * 10^2$
15.3e1	15.3	1	$15.3 * 10^1$
-2.e5	-2	5	$-2 * 10^5$

Table 8-1: Evaluating numeric data (continued)

Data Entered	Mantissa	Exponent	Value
2.2e-1	2.2	-1	$2.2 * 10^{-1}$
+56E+2	56	2	$56 * 10^{2}$

The column's binary precision determines the maximum number of binary digits allowed in the mantissa. For float columns, you can specify a precision of up to 48 digits; for real and double precision columns, the precision is machine dependent. If a value exceeds the column's binary precision, SQL Server flags the entry as an error.

decimal and *numeric*

The exact numeric types—*dec*, *decimal*, and *numeric*—begin with an optional positive or negative sign and can include a decimal point. The value of exact numeric data depends on the column's decimal *precision* and *scale*, which you specify using the following syntax:

datatype **[(*precision* [, *scale*])]**

SQL Server treats each combination of precision and scale as a distinct datatype. For example, *numeric (10,0)* and *numeric (5,0)* are two separate datatypes. The precision and scale determine the range of values that can be stored in a *decimal* or *numeric* column:

- The precision specifies the maximum number of decimal digits that can be stored in the column. It includes all digits to the right and left of the decimal point. You can specify a precision ranging from 1-38 digits or use the default precision of 18 digits.
- The scale specifies the maximum number of digits that can be stored to the right of the decimal point. The scale must be less than or equal to the precision. You can specify a scale ranging from 0-38 digits or use the default scale of 0 digits.

If a value exceeds the column's precision or scale, SQL Server flags the entry as an error. Here are some examples of valid *dec* and *numeric* data:

Table 8-2: Valid precision and scale for numeric data

Data Entered	Datatype	Precision	Scale	Value
12.345	*numeric*(5,3)	5	3	12.345
-1234.567	*dec*(8,4)	8	4	-1234.567

The following entries result in errors because they exceed the column's precision or scale:

Table 8-3: Invalid precision and scale for numeric data

Data Entered	Datatype	Precision	Scale
1234.567	*numeric*(3,3)	3	3
1234.567	*decimal*(6)	6	1

int, *smallint*, and *tinyint*

You can insert numeric values into *int*, *smallint*, and *tinyint* columns with the E-notation described in the preceding section.

timestamp

You cannot insert data into a *timestamp* column. You must either insert an explicit null, by typing "NULL" in the column, or use an implicit null, by providing a column list that skips the *timestamp* column. SQL Server automatically updates the timestamp value after each insert or update. See the discussion in "Inserting Data into Specific Columns" on page 8-12 for further information.

Adding New Data

You can use the **insert** command to add rows to the database in two ways: with the **values** keyword or with a **select** statement.

The **values** keyword is used to specify values for some or all of the columns in a new row. A simplified version of the syntax for the **insert** command using the **values** keyword is:

```
insert table_name
   values (constant1, constant2, ...)
```

You can use a **select** statement in an **insert** statement to pull values from one or more tables. A simplified version of the syntax for the **insert** command using a **select** statement is:

```
insert table_name
   select column_list
     from table_list
     where search_conditions
```

insert Syntax

Here is the full syntax for the **insert** command:

```
insert [into] [database.[owner.]]{table_name |
   view_name} [(column_list)]
   {values (constant_expression
   [, constant_expression]...) | select_statement}
```

➤ *Note*

When *text* and *image* values are added with **insert**, all of the data is written to the transaction log. The **writetext** command allows you to add these values without logging the long chunks of data that may comprise *text* or *image* values. See “Inserting Data into Specific Columns” on page 8-12, and “Changing text and image Data” on page 8-23.

Adding New Rows with *values*

This **insert** statement adds a new row to the *publishers* table, giving a value for every column in the row:

```
insert into publishers
values ('1622', 'Jardin, Inc.', 'Camden', 'NJ')
```

Notice that the data values are typed in the same order as the column names in the original **create table** statement, that is, first the ID number, then the name, the city, and finally the state. The **values** data is surrounded by parentheses and all character data is enclosed in single or double quotes.

Use a separate **insert** statement for each row you add.

Inserting Data into Specific Columns

You can add data to some, but not all, of the columns in a row, by specifying those columns and the data for just those columns. All other columns that aren’t included in the column list must be defined to allow null values. The skipped columns can accept defaults. If you skip a column that has a default bound to it, the default will be used.

You may especially wish to use this form of the command to insert all of the values in a row except the *text* or *image* values, and then use **writetext** to insert the long data values, so that these values won’t be

stored in the transaction log. Also, this form of the command can be used to skip over *timestamp* data.

Adding data in only two columns, say *pub_id* and *pub_name*, requires a command like this:

```
insert into publishers (pub_id, pub_name)
values ('1756', 'The Health Center')
```

The order in which you list the column names must match the order in which you list the values. The following example produces the same results as the previous one:

```
insert publishers (pub_name, pub_id)
values('The Health Center', '1756')
```

Either of the **insert** statements would put "1756" in the identification number column and "The Health Center" in the publisher name column. Since the *pub_id* column in *publishers* has a unique index, you cannot execute both of these **insert** statements; the second attempt to insert a *pub_id* value of "1756" produces an error message.

The following **select** statement shows the row that was added to *publishers*:

```
select *
from publishers
where pub_name = 'The Health Center'
pub_id  pub_name             city    state
------- -----------------    ------  -------
1756    The Health Center    NULL    NULL
```

SQL Server enters null values in the *city* and *state* columns because no value was given for these columns in the **insert** statement, and the *publisher* table allows null values in these columns.

SQL Server-Generated Values for IDENTITY Columns

When you insert a row into a table with an IDENTITY column, SQL Server automatically generates the column value. Do not include the name of the IDENTITY column in the column list, or its value in the values list.

This **insert** statement adds a new row to the *sales_daily* table. Notice that the column list does not include the IDENTITY column, *row_id*:

```
insert sales_daily (stor_id)
values ("7896")
```

The following statement shows the row that was added to *sales_daily*. SQL Server automatically generated the next sequential value, 2, for *row_id*:

```
select * from sales_daily
where stor_id = "7896"

row_id  stor_id
------  -------
    2   7896
```

Null Values, Defaults, IDENTITY Columns, and Errors

When you specify values for only some of the columns in a row, one of four things can happen to the columns with no values:

- A default value is entered if one exists for the column or user-defined datatype. See Chapter 12, "Defining Defaults and Rules for Data," or the **create default** entry in the *SQL Server Reference Manual* for details.
- NULL is entered if NULL was specified for the column when the table was created and no default value exists for the column or datatype. See also the **create table** entry in the *SQL Server Reference Manual*.
- A unique, sequential value is entered if the column has the IDENTITY property.
- An error message is displayed and the row is not added if NULL was not specified and no default exists.

The following table shows what you would see under these circumstances:

Table 8-4: Columns with no values

Default?	Not Null	Null	Identity
Yes	The default	The default	Next sequential value
No	Error message	NULL	Next sequential value

You can use the system procedure **sp_help** to get a report on a specified table or default or on any other object listed in the system table *sysobjects*. To see the definition of a default, use the system procedure **sp_helptext**.

Explicitly Inserting Data into an IDENTITY Column

At times, you may want to insert a specific value into an IDENTITY column, rather than accepting a server-generated value. For example, you may want the first row inserted into the table to have an IDENTITY value of 101, rather than 1. Or you may need to reinsert a row that was deleted by mistake.

Only the table owner, the Database Owner, or the System Administrator can explicitly insert a value into an IDENTITY column. Before inserting the data, the user must set the **identity_insert** option **on** for the table. A user can **set identity_insert on** for only one table at a time in a database.

This example specifies a "seed" value of 101 for the IDENTITY column:

```
set identity_insert sales_daily on
insert into sales_daily (syb_identity, stor_id)
values (101, '13-J-9')
```

Notice that the **insert** statement lists each column, including the IDENTITY column, for which a value is specified. When the **identity_insert** option is turned on, each **insert** statement for the table must specify an explicit column list. The values list must specify an IDENTITY column value, since IDENTITY columns do not allow nulls.

➤ ***Note***

SQL Server does not enforce the uniqueness of the inserted value. You can specify any positive integer within the range allowed by the column's declared precision. To ensure that only unique column values are accepted, you must create a unique index on the IDENTITY column before inserting any rows.

Restricting Column Data: Rules

You can create a rule and bind it to a column or user-defined datatype. Rules govern the kind of data that can or cannot be added.

The *pub_id* column of the *publishers* table is an example. A rule called *pub_idrule*, which specifies acceptable publisher identification numbers, is bound to the column. The acceptable IDs are "1389", "0736", "0877", "1622", and "1756", or any four-digit number the

first two digits of which are “99”. If you try to enter any other number, you get an error message.

When you get this kind of error message, you may want to look at the definition of the rule. Use the system procedure **sp_helptext**:

```
sp_helptext pub_idrule

---------
        1

(1 row affected)

text
--------------------------------------------------
create rule pub_idrule
as @pub_id in ("1389", "0736", "0877", "1622", "1756")
or @pub_id like "99[0-9][0-9]"

(1 row affected)
```

For more general information on a specific rule, use **sp_help**. Or use **sp_help** with a table as a parameters in order to find out whether any of its defined columns has a rule. Chapter 12, “Defining Defaults and Rules for Data,” describes rules in more detail.

Adding New Rows with *select*

To pull values into a table from one or more other tables, use a **select** clause in the **insert** statement. The **select** clause can insert values into some or all of the columns in a row.

Inserting values for only some columns can come in handy when you want to take some values from an existing table. Then you can use **update** to add the values for the other columns.

Before inserting values for some but not all of the columns in a table, make sure that a default exists or NULL has been specified for the columns for which you are not inserting values. Otherwise, you’ll get an error message.

When you insert rows from one table into another, the two tables must have compatible structures—that is, the matching columns must be either the same datatypes or datatypes between which SQL Server automatically converts.

➤ ***Note***

You cannot insert data from a table that allows null values into a table that does not, if any of the data being inserted is null.

If the columns are in the same order in their **create table** statements, you don't need to specify column names in either table. Suppose you had a table *newauthors* which contained some rows of author information in the same format as *authors*. To add to *authors* all the rows in *newauthors*:

```
insert authors
select *
from newauthors
```

To insert rows into a table based on data in another table, the columns in the two tables do not have to be listed in the same sequence in their respective **create table** statements. You can use either the **insert** or the **select** statement to order the columns so that they match.

For example, say the **create table** statement for the *authors* table contained the columns *au_id*, *au_fname*, *au_lname*, and *address* in that order, while *newauthors* contained *au_id*, *address*, *au_lname*, and *au_fname*. You would have to make the column sequence match in the **insert** statement. You could do this in either of these two ways:

```
insert authors (au_id, address, au_lname, au_fname)
select * from newauthors
```

```
insert authors
select au_id, au_fname, au_lname, address
    from newauthors
```

If the column sequence in the two tables fails to match, SQL Server cannot complete the **insert** operation, or completes it "incorrectly," putting data in the wrong column. For example, you might get address data in the *au_lname* column.

Computed Columns

You can use computed columns in a **select** statement inside an **insert** statement. For example, imagine that a table named *tmp* contains some new rows for the *titles* table with some out-of-date data; the *price* figures need to be doubled. A statement to increase the prices and insert the *tmp* rows into *titles* looks like the following:

```
insert titles
select title_id, title, type, pub_id, price*2,
    advance, total_sales, notes, pubdate, contract
from tmp
```

When you perform computations on a column, you cannot use the **select *** syntax. Each column must be named individually in the select list.

Inserting Data into Some Columns

You can use the **select** statement to add data to some, but not all, of the columns in a row just as you do with the **values** clause. Simply specify the columns to which you want to add data in the **insert** clause.

For example, there are some authors in the *authors* table that do not have titles and hence do not have entries in the *titleauthor* table. To pull their *au_id* numbers out of the *authors* table and insert them into the *titleauthor* table as placeholders, you might try to use this statement:

```
insert titleauthor (au_id)
select au_id
    from authors
    where au_id not in
    (select au_id from titleauthor)
```

However, this statement is not legal, because a value is required for the *title_id* column. Null values are not permitted and no default is specified. You can put in the dummy value "*xx1111*" for *titles_id* by using a constant, as follows:

```
insert titleauthor (au_id, title_id)
select au_id, "xx1111"
    from authors
    where au_id not in
    (select au_id from titleauthor)
```

The *titleauthor* table now contains four new rows with entries for the *au_id* column, dummy entries for the *title_id* column, and null values for the other two columns.

Inserting Data from the Same Table

You can insert data into a table based on other data in the same table. Essentially, this means copying all or part of a row.

For example, you can insert a new row in the *publishers* table that is based on the values in an already existing row in the same table. Make sure you follow the rule on the *pub_id* column. Here's how:

```
insert publishers
select "9999", "test", city, state
    from publishers
    where pub_name = "New Age Books"

(1 row affected)
select * from publishers

pub_id  pub_name              city        state
------- --------------------- -------     ------
0736    New Age Books         Boston      MA
0877    Binnet & Hardley      Washington  DC
1389    Algodata Infosystems  Berkeley    CA
9999    test                  Boston      MA

(4 rows affected)
```

The example inserts the two constants ("9999" and "test") and the values from the *city* and *state* columns in the row that satisfied the query.

Changing Existing Data

You can use the **update** command to change single rows, groups of rows, or all the rows in a table. The **update** command is followed by the name of the table or view. As in all the data modification statements, you can change the data in only one table at a time.

The **update** command specifies the row or rows you want changed, and the new data. The new data can be a constant or expression that you specify, or data pulled from other tables.

If an **update** statement violates an integrity constraint, the update does not take place and an error message is generated. The update is cancelled, for example, if it affects the table's IDENTITY column, or one of the values being added is the wrong datatype, or if it violates a rule that has been defined for one of the columns or datatypes involved.

SQL Server does not prevent you from issuing an **update** command that updates a single row more than once. However, because of the way that **update** is processed, updates from a single statement do not

accumulate. That is, if an **update** statement modifies the same row twice, the second update is not based on the new values from the first update but on the original values. The results are unpredictable, since they depend on the order of processing.

See Chapter 9, "Views: Limiting Access to Data," for restrictions on updating views.

➤ ***Note***

The **update** command is logged. If you are changing large blocks of text or image data, you may wish to use the **writetext** command, which is not logged. Also, you are limited to approximately 125K per **update** statement. See the discussion of **writetext** in "Changing text and image Data" on page 8-23.

update Syntax

Here's a simplified version of the **update** syntax for updating specified rows with an expression:

```
update table_name
   set column_name = expression
   where search_conditions
```

For example, if Reginald Blotchet-Halls decides to change his name to Goodbody Health in order to boost his visualization processes, here's how to change his row in the *authors* table:

```
update authors
set au_lname = "Health", au_fname = "Goodbody"
where au_lname = "Blotchet-Halls"
```

This simplified syntax statement updates a table based on data from another table:

```
update table_name
   set column_name = expression
     from table_name
     where search_conditions
```

Here's an example that updates the *total_sales* column of the *titles* table to reflect the most recent sales recorded in the *salesdetail* table:

```
update titles
set total_sales = total_sales + qty
from titles, sales, salesdetail
where titles.title_id = salesdetail.title_id
and salesdetail.stor_id = sales.stor_id
and sales.date in (select max(sales.date) from sales)
```

The preceding example assumes that only one set of sales is recorded for a given title on a given date, and that updates are up to date. The full syntax for **update** is:

```
update [[database.]owner.]{table_name | view_name}
    set [[[database.]owner.]{table_name. |
           view_name.}] column_name1 =
           {expression1 | null | (select_statement)}
           [, column_name2 = {expression2 | null |
           (select_statement)}]...
   [from [[database.]owner.]{table_name | view_name}
           [, [[database.]owner.]{table_name |
           view_name}]]...
   [where search_conditions]
```

Using the *set* Clause with *update*

The **set** clause specifies the columns and the changed values. The **where** clause determines which row or rows will be updated. Note that if you don't have a **where** clause, the specified columns of **all** the rows will be updated with the values given in the **set** clause.

➤ ***Note***

Before trying the examples in this section, make sure you know how to reinstall the *pubs2* database.

For example, if all the publishing houses in the *publishers* table move their head offices to Atlanta, Georgia, this is how you update the table:

```
update publishers
set city = "Atlanta", state = "GA"
```

In the same way, you can change the names of all the publishers to NULL with this statement:

```
update publishers
set pub_name = null
```

You can also use computed column values in an update. To double all the prices in the *titles* table, use this statement:

```
update titles
set price = price * 2
```

Since there is no **where** clause, the change in prices is applied to every row in the table.

Using the *where* Clause with *update*

The **where** clause specifies which rows are to be updated. For example, in the unlikely event that northern California is renamed Pacifica (abbreviated PC) and the people of Oakland vote to change the name of their city to something exciting, like Big Bad Bay City, here is how you can update the *authors* table for all former Oakland residents whose addresses are now out of date:

```
update authors
set state = "PC", city = "Big Bad Bay City"
where state = "CA" and city = "Oakland"
```

You need to write another statement to change the name of the state for residents of other northern California cities.

Using the *from* Clause with *update*

Use the **from** clause to pull data from one or more tables into the table you're updating.

For example, earlier in this chapter, an example was given for inserting some new rows into the *titleauthor* table for authors without titles, filling in the *au_id* column and giving dummy or null values for the other columns. When one of these authors, Dirk Stringer, writes a book, *The Psychology of Computer Cooking*, a title identification number is assigned to his book in the *titles* table. You can modify his row in the *titleauthor* table by adding a title identification number for him:

```
update titleauthor
set title_id = titles.title_id
from titleauthor, titles, authors
    where titles.title =
    "The Psychology of Computer Cooking"
    and authors.au_id = titleauthor.au_id
    and au_lname = "Stringer"
```

Note that an **update** without the *au_id* join changes all the *title_ids* in the *titleauthor* table so that they are the same as *The Psychology of Computer Cooking*'s identification number. If two tables are identical in structure except that one has NULL fields and some null values and the other has NOT NULL fields, it is impossible to insert the data from the NULL table into the NOT NULL table with a **select.** In other words, a field that does not allow nulls cannot be updated by selecting from a field that does, if any of the data is NULL.

Changing *text* and *image* Data

The **writetext** command is used to change *text* or *image* values when you don't want to store the long text values in the database transaction log. **update** commands, which can also be used for *text* or *image* columns, are always logged. In its default mode, **writetext** commands are not logged.

➤ ***Note***

To use **writetext** in its default, non-logged state, a System Administrator must use **sp_dboption** to set **select into/bulkcopy** on. This permits the insertion of non-logged data. After using **writetext**, it is necessary to **dump database**. A transaction dump which is done after unlogged changes to the database is not usable in a **load transaction** operation.

The **writetext** command completely overwrites any existing data in the column it affects. For **writetext** to work, the column must already contain a valid text pointer. There are 2 ways to create a text pointer:

- **insert** actual data into the *text* or *image* column
- **update** the column with data or a NULL

Since an "initialized" *text* column uses 2K of storage, even to store a couple of words, SQL Server saves space by not initializing *text* columns when explicit or implicit null values are placed in *text* columns with **insert.** This command will **not** initialize a *text* column:

```
insert blurbs
values ("172-32-1176", NULL)
```

After an **insert** like the one above, you can use this **update** statement to initialize the *text* column:

```
update blurbs
set copy=NULL
where au_id="172-32-1176"
```

Once you have initialized the pointer, you can use **writetext**. The following **writetext** example adds a text to an existing row in the *blurbs* table:

```
declare @val varbinary(16)
select @val = textptr(copy) from blurbs
where  au_id="172-32-1176"

writetext blurbs.copy @val
"This book is a must for true data junkies."
```

This example puts the text pointer into the local variable *@val*, then **writetext** places the new text string into the row pointed to by *@val*.

Deleting Data

Like **insert** and **update**, **delete** works for single-row as well as multiple-row operations, but is more suitable for the latter. As for the other data modification statements, you can delete rows based on data in other tables.

For example, if you decide to remove one row from *publishers*—the row added for Jardin, Inc.—type:

```
delete publishers
where pub_name = "Jardin, Inc."
```

delete Syntax

A simplified version of **delete** syntax is:

```
delete table_name
   where column_name = expression
```

Here is the complete syntax statement, which shows that you can remove rows either on the basis of specified expressions or based on data from other tables:

```
delete [from] [[database.]owner.]{table_name |
         view_name}
     [from [[database.]owner.]{table_name | view_name}
         [, [[database.]owner.]{table_name |
         view_name}]...]
     [where search_conditions]
```

The optional **from** immediately after the **delete** keyword is included for compatibility with other versions of SQL. The **from** on the second line

is a SQL Server enhancement that allows you to make deletions based on data in other tables.

Using the *where* Clause with *delete*

The **where** clause specifies which rows are to be removed. When no **where** clause is given in the **delete** statement, **all** rows in the table are removed.

Using the *from* Clause with *delete*

The **from** clause in the second position of a **delete** statement is a special Transact-SQL feature that allows you to select data from a table or tables and delete corresponding data from the first-named table. The rows you select in the **from** clause specify the conditions for the **delete**.

Suppose that a complex corporate deal results in the acquisition of all the Big Bad Bay City, formerly Oakland, authors and their books by another publisher. You need to remove all these books from the *titles* table right away, but you don't know their titles or identification numbers. The only information you have is the author's names and addresses.

You can delete the rows in *titles* by finding the author identification numbers for the rows that have Big Bad Bay City as the town in the *authors* table and using these numbers to find the title identification numbers of the books in the *titleauthor* table. In other words, a three-way join is required to find the rows you want to delete in the *titles* table.

The three tables are all included in the **from** clause of the **delete** statement. However, only the rows in the *titles* table that fulfill the conditions of the **where** clause are deleted. You would have to do separate deletes to remove relevant rows in tables other than *titles*.

Here is the statement you need:

```
delete titles
from authors, titles, titleauthor
where titles.title_id = titleauthor.title_id
and authors.au_id = titleauthor.au_id
and city = "Big Bad Bay City"
```

The *deltitle* trigger in the *pubs2* database prevents you from actually performing this deletion, since it won't allow you to delete any titles that have sales recorded in the *sales* table.

Deleting All Rows from a Table

Use **truncate table** as a fast method of deleting all the rows in a table. It's almost always faster than a **delete** statement with no conditions, because the **delete** logs each change, while **truncate table** just logs the deallocation of whole data pages. **truncate table** immediately frees all the space that the table's data and indexes had occupied. The freed space can then be used by any object. The distribution pages for all indexes are also deallocated. Remember to run **update statistics** after adding new rows to the table.

As with **delete**, a table emptied with the **truncate table** command remains in the database, along with its indexes and other associated objects, unless you enter a **drop table** command.

truncate table Syntax

The syntax of **truncate table** is:

```
truncate table [[database.]owner.]table_name
```

For example, to remove all the data in *sales*, type:

```
truncate table sales
```

Permission to use the **truncate table** command, like **drop table**, defaults to the table owner and cannot be transferred.

A **truncate table** command is not caught by a **delete** trigger. See Chapter 15, "Triggers: Enforcing Referential Integrity," for details on triggers.

9 Views: Limiting Access to Data

You can use views to focus, simplify, and customize each user's perception of the database. Views also provide a security mechanism by allowing users to access only the data they require. These and other advantages are described in this chapter.

This chapter discusses:

- A general overview of using views
- How to create views
- How to retrieve data through views
- How to update data through views
- How to generate information about views

What Are Views?

A **view** is an alternative way of looking at the data in one or more tables. You can think of a view as a frame through which you can see the particular data in which you're interested. That is why one speaks of looking at data or changing data "through" a view.

A view is derived from one or more real tables whose data is physically stored in the database. The tables from which a view is derived are called its base tables or underlying tables. A view can also be derived from another view.

The definition of a view, in terms of the base tables from which it is derived, is stored in the database. No separate copies of data are associated with this stored definition. The data that you view is stored in the underlying tables.

A view looks exactly like any other database table. You can display it and operate on it almost exactly as you can any other table. Transact-SQL has been enhanced so that there are no restrictions at all on querying through views, and fewer than usual on modifying them. The exceptions are explained later in this chapter.

When you modify the data you see through a view, you are actually changing the data in the underlying base tables. Conversely, changes to data in the underlying base tables are automatically reflected in the views derived from them.

Advantages of Views

The examples in this chapter demonstrate that views can be used to focus, simplify, and customize each user's perception of the database. Views also provide an easy-to-use security measure. In addition, they can be helpful when changes are made to the structure of the database and users prefer to work with the database in the style to which they have become accustomed.

Focus

Views allow users to focus in on the particular data that interests them and on the particular tasks for which they're responsible. Data that is not of interest to a particular user or for a particular task can be left out of the view.

Simpler Data Manipulation

Not only the users' perception of the data, but also their manipulation of it, can be simplified with views. Frequently used joins, projections, and selections can be defined as views so that users don't have to specify all the conditions and qualifications each time a further operation on that data is performed.

Customization

Views allow different users to see the same data in different ways, even when they're using the same data at the same time. This advantage is particularly important when users of many different interests and skill levels share the same database.

Security

Through a view, users can query and modify only the data they can see. The rest of the database is neither visible nor accessible.

With the **grant** and **revoke** commands, each user's access to the database can be restricted to specified database objects—including views. If the view and all the tables and views from which it was derived are owned by the same user, that owner can grant permission to others to use the view while denying permission to use its underlying tables and views. This is a simple but effective security mechanism. See the *Security Administration Guide* for details on the **grant** and **revoke** commands.

By defining different views and selectively granting permissions on them, a user or any combination of users can be restricted to different subsets of data. The following illustrates the use of views for security purposes:

- Access can be restricted to a subset of the rows of a base table, that is, a value-dependent subset. For example, you might define a view that contains only the rows for business and psychology books, in order to keep information about other types of books hidden from some users.
- Access can be restricted to a subset of the columns of a base table, that is, a value-independent subset. For example, you might define a view that contains all the rows of the *titles* table, but omits the *royalty* and *advance* columns, since this information is sensitive.
- Access can be restricted to a row-and-column subset of a base table.
- Access can be restricted to the rows that qualify for a join of more than one base table. For example, you might define a view that joins the *titles*, *authors*, and *titleauthor* table in order to display the names of the authors and the books they have written. This view would hide personal data about authors and financial information about the books.
- Access can be restricted to a statistical summary of data in a base table. For example, through the view *category_price* a user can access only the average price of each type of book.
- Access can be restricted to a subset of another view or a combination of views and base tables. For example, through the view *hiprice_computer* a user can access the title and price of computer books that meet the qualifications in the view definition of *hiprice*.

In order to create a view, a user must be granted **create view** permission by the Database Owner, and must have appropriate permissions on any tables or views referenced in the view definition.

If a view references objects in different databases, users of the view must be valid users or guests in each of the databases.

As the owner of an object on which other users have created views, you must be aware of who can see what data through what views. Consider this situation: The Database Owner has granted “harold” **create view** permission, and a user named “maude” has granted “harold” permission to **select** from a table she owns. Given these

permissions, "harold" can create a view that selects all columns and rows from the table owned by "maude." If "maude" subsequently revokes permission for "harold" to **select** from her table, he can still look at her data through the view he has created.

Logical Data Independence

Views help to shield users from changes in the structure of the real tables if such changes become necessary.

For example, say the database is restructured by using **select into** to split the *titles* table into these two new base tables and dropping *titles*:

```
titletext (title_id, title, type, notes)

titlenumbers (title_id, pub_id, price, advance,
royalty, total_sales, pub_date)
```

Notice that the old *titles* table can be "regenerated" by joining on the *title_id* columns of the two new tables. To shield the changed structure of the database from users, you can create a view that is the join of the two new tables. You can even name it *titles*.

Any query or stored procedure that previously referred to the base table *titles* now refers to the view *titles*. As far as the users are concerned, **select** operations continue to work exactly as before. Users who only retrieve from the new view need not even know that the restructuring has occurred.

Unfortunately, views provide only partial logical independence. Some data modification statements on the new *titles* are not allowed because of certain restrictions.

View Examples

The first example is a view derived from the *titles* table. Suppose you are interested only in books priced higher than $15 and for which an advance of more than $5,000 was paid. This straightforward **select** statement would find the rows that qualify:

```
select *
from titles
where price > $15
  and advance > $5000
```

Now suppose you have a lot of retrieval and update operations to do on this collection of data. You could, of course, combine the conditions shown in the previous query with any command that you

issue. However, for convenience, you can create a view in which just the records of interest are visible:

```
create view hiprice
as select *
from titles
where price > $15
  and advance > $5000
```

When SQL Server receives this command, it does not actually execute the **select** statement that follows the keyword **as**. Instead, it stores the **select** statement, which is in fact the definition of the view *hiprice*, in the system table *syscomments*. Entries are also made in *sysobjects* and in *syscolumns* for each column included in the view.

Now, when you display or operate on *hiprice*, SQL Server combines your statement with the stored definition of *hiprice*. For example, you can change all the prices in *hiprice* just as you can change any other table:

```
update hiprice
set price = price * 2
```

SQL Server actually finds the view definition in the system tables and converts this update command into the statement:

```
update titles
set price = price * 2
where price > $15
  and advance > $5000
```

In other words, SQL Server knows from the view definition that the data to be updated is in *titles*. It also knows that it should increase the prices only in those rows that meet the conditions on the *price* and *advance* columns given in the view definition and those in the update statement.

Having issued the first update statement—the update to *hiprice*—you can see its effect either in the view or in the *titles* table. Conversely, if you had created the view and then issued the second update statement, which operates directly on the base table, the changed prices would also be visible through the view.

Updating a view's underlying table in such a way that different rows qualify for the view affects the view. For example, say you increase the price of the book *You Can Combat Computer Stress* to $25.95. Since this book now meets the qualifying conditions in the view definition statement, it is considered part of the view.

However, if you alter the structure of a view's underlying table by adding columns, the new columns will **not** appear in a view defined

with a **select** * clause unless the view is dropped and redefined. This is because the asterisk shorthand is interpreted and expanded when the view is first created.

Creating Views

View names must be unique for each user among the already existing tables and views. If you have **set quoted_identifier on**, you can use a delimited identifier for the view. Otherwise, the view name must follow the rules for identifiers given in Chapter 1.

You can build views on other views and procedures that reference views. You can define primary, foreign, and common keys on views. You cannot associate rules, defaults, or triggers with views or build indexes on them. Temporary views cannot be created, nor can views be created on temporary tables.

create view Syntax

Here is the full syntax for creating a view:

```
create view [[database.]owner.]view_name
        [(column_name [, column_name]...)]
   as select [distinct] select_statement
   [with check option]
```

As illustrated in the example given in the previous section, you need not specify any column names in the **create** clause of a view definition statement. SQL Server gives the columns of the view the same names and the same datatypes as the columns referred to in the select list of the **select** statement. The select list can be an "*****", as in the example, or a full or partial list of the column names in the base tables.

You can build views which do not contain duplicate rows. Use the **distinct** keyword of the **select** statement to ensure that each row in the view is unique.

distinct views cannot be updated.

It is always legal to specify column names. However, column names **must** be specified in the **create** clause for **every** column in the view if any of the following are true:

- Any of the view's columns are derived from an arithmetic expression, an aggregate, a built-in function, or a constant.
- Two or more of the view's columns would otherwise have the same name. This usually happens because the view definition

includes a join, and the columns being joined have the same name.

- You wish to give any column in the view a different name than the column from which it is derived. You can also rename columns in the **select** statement. Whether or not you rename a view column, it inherits the datatype of the column from which it is derived.

Here is a view definition statement that makes the name of a column in the view different from its name in the underlying table:

```
create view pub_view (Publisher, city, state)
as select pub_name, city, state
from publishers
```

Here is an alternate method of creating the same view but renaming the columns in the **select** statement:

```
create view pub_view2
as select Publisher = pub_name, city, state
from publishers
```

The examples of view definition statements given in a later section illustrate the rest of the rules for including column names in the **create** clause.

The next section discusses the **select** statement, use of the **distinct** keyword, and the **with check option** clause of view definitions. The **drop view** command is discussed after that.

Using the *select* Statement with *create view*

The **select** statement in the **create view** statement defines the view. You must have permission to **select** from any objects referenced in the **select** statement of a view you are creating.

A view need not be a simple subset of the rows and columns of one particular table, as in our example. You can create a view using more than one table and other views, with a **select** statement of any complexity.

There are a few restrictions on the **select** statements in a view definition:

- You cannot include **order by** or **compute** clauses.
- You cannot include the **into** keyword.
- You cannot reference a temporary table.

View Definition with Projection

To create a view with all the rows of the *titles* table, but with only a subset of its columns, type the statement:

```
create view titles_view
as select title, type, price, pubdate
from titles
```

Note that no column names are included in the **create view** clause. The view *titles_view* will inherit the column names given in the select list.

View Definition with a Computed Column

Here is a view definition statement that creates a view with a computed column generated from the columns *price*, *royalty*, and *total_sales*:

```
create view accounts (title, advance, amt_due)
as select titles.title_id, advance, (price *
royalty /100 ) * total_sales
from titles, roysched
where price > $15
and advance > $5000
and titles.title_id = roysched.title_id
and total_sales between lorange and hirange
```

In this example, a list of columns must be included in the **create** clause, since there is no name that can be inherited by the column computed by multiplying together *price*, *royalty*, and *total_sales*. The computed column is given the name *amt_due*. It must be listed in the same position in the **create** clause as the expression from which it is computed is listed in the **select** clause.

View Definition with an Aggregate or Built-In Function

A view definition that includes an aggregate or built-in function must include column names in the **create** clause. For example:

```
create view categories (category, average_price)
as select type, avg(price)
from titles
group by type
```

If you create a view for security reasons, you must be careful when using aggregate functions and the **group by** clause. The Transact-SQL extension that does not restrict the columns you can include in the

select with **group by** may also cause the view to return more information than required. For example:

```
create view categories (category, average_price)
as select type, avg(price)
from titles
where type = "business"
```

In the above case, you may have wanted the view to restrict its results to "business" categories, but the results have information about other categories. For more information about **group by** and this **group by** Transact-SQL extension, see Chapter 3.

View Definition with a Join

You can create a view derived from more than one base table. Here's an example of a view derived from both the *authors* and the *publishers* tables. The view contains the names and cities of the authors that live in the same city as a publisher, along with each publisher's name and city.

```
create view cities (authorname, acity,
publishername, pcity)
as select au_lname, authors.city, pub_name,
publishers.city
from authors, publishers
where authors.city = publishers.city
```

Views Derived from Other Views

You can define a view in terms of another view, as in this example:

```
create view hiprice_computer
as select title, price
from hiprice
where type = 'popular_comp'
```

distinct Views

You can ensure that the rows contained in a view are unique, as in this example:

```
create view author_codes
as select distinct au_id
from titleauthor
```

A row is a duplicate of another row if all of its columns' values exactly match the same columns' values contained in another row. Two null values are considered to be identical.

SQL Server applies the **distinct** requirement to the view's definition when it accesses the view for the first time, and before it does any projecting or selecting. Views look and act like any database table. If you select a projection of the distinct view (that is, you select some of the view's columns, but all of its rows), you can get results which appear to be duplicates. However, each row in the view itself is still unique. For example, suppose that you create a **distinct** view, *myview*, with three columns, *a*, *b*, and *c*, which contains these values:

Table 9-1: distinct view myview sample values

a	b	c
1	1	2
1	2	3
1	1	0

When you enter this query:

```
select a, b from myview
```

the results look like this:

```
a    b
---  ---
1    1
1    2
1    1
```

The first and third rows appear to be duplicates. However, the underlying view's rows are still unique.

Views That Include IDENTITY Columns

You can define a view that includes an IDENTITY column, as in this example:

```
create view sales_view
as select syb_identity, stor_id
from sales_daily
```

You can select the IDENTITY column from the view using the **syb_identity** keyword unless the view:

- Selects the IDENTITY column more than once, or

- Includes columns from more than one table, or
- Computes a new column from the IDENTITY column, or
- Includes an aggregate function.

If one or more of these conditions is true, SQL Server does not recognize the column as an IDENTITY column with respect to the view. When you execute the **sp_help** system procedure on the view, the column displays an IDENTITY value of 0.

Using the *with check option* Keyword with *create view*

Normally, SQL Server does not check **insert** and **update** statements on views to determine whether the affected rows are within the scope of the view. A statement can insert a row into the underlying base table but not into the view, or change an existing row so that it no longer meets the view's selection criteria.

When you create a view **with check option**, each **insert** and **update** through the view is validated against the view's selection criteria. All rows inserted or updated through the view must remain visible through the view, or the statement fails.

Here's an example of a view, *stores_cal*, created **with check option**. This view includes information about stores located in California, but excludes information about stores located in any other state. The view is created by selecting all rows from the *stores* table for which *state* has a value of "CA":

```
create view stores_cal
as select * from stores
where state = "CA"
with check option
```

When you try to insert a row through *stores_cal*, SQL Server verifies that the new row falls within the scope of the view. The following **insert** statement fails because the new row would have a *state* value of "NY", rather than "CA":

```
insert stores_cal
values ("7100", "Castle Books", "351 West 24 St.",
"New York", "NY", "USA", "10011", "Net 30")
```

When you try to update a row through *stores_cal*, SQL Server verifies that the update will not cause the row to disappear from the view. The following **update** statement fails because it would change the value of *state* from "CA" to "MA". After the update, the row would no longer be visible through the view.

```
update stores_cal
set state = "MA"
where stor_id = "7066"
```

Views Derived from Other Views

If a view is created **with check option**, all views that are derived from the "base" view must satisfy its check option. Each row inserted through the derived view must be visible through the base view. Each row updated through the derived view must remain visible through the base view.

Consider the view *stores_cal30*, which is derived from *stores_cal*. The new view includes information about stores in California with payment terms of "Net 30":

```
create view stores_cal30
as select * from stores_cal
where payterms = "Net 30"
```

Because *stores_cal* was created **with check option**, all rows inserted or updated through *stores_cal30* must be visible through *stores_cal*. Any row with a *state* value other than "CA" is rejected.

Notice that *stores_cal30* does not have a **with check option** clause of its own. This means that it is possible to insert or update a row with a *payterms* value other than "Net 30" through *stores_cal30*. The following **update** statement would be successful, even though the row would no longer be visible through *stores_cal30*:

```
update stores_cal30
set payterms = "Net 60"
where stor_id = "7067"
```

Limitations on Views Defined with Outer Joins

Views defined with outer joins have some limitations that may lead to unexpected results when you retrieve data from them. Care should be taken when using such views.

If you define a view with an outer join, and then query the view with a qualification on a column from the inner table of the outer join, the results may be other than what you expect. The qualification in the query does not restrict the number of rows returned, but rather affects which rows contain the NULL value. For rows that do not meet the qualification, a NULL value appears in the inner table's columns of those rows. This is a result of the fact that in Transact-

SQL, the internal representation of a query on a view is a combination of the view definition and the qualification on the view.

For example, suppose we have the following tables:

Table A:

a
1
2
3

Table B:

b	*c*
1	10
2	11
6	12

Then create a view on these two tables. The view definition contains an outer join:

```
create view A_B as
select a,b,c from A,B
where A.a*=B.b
```

Then run the following query, which produces the results shown:

```
select a,c from A_B where c = 10
```

```
a           c
-----       -----

    1          10          Joins and qualifies on c
    2       NULL           Joins, but does not meet qualifications
    3       NULL           Does not join
(3 rows affected)
```

The qualification (c = 10) does not affect the number of rows returned. Rather, NULL appears in the inner table's column for each row that does not meet the qualification or does not join with rows in the outer table.

Retrieving Data Through Views

When you retrieve data through a view, SQL Server checks to make sure that all the database objects referenced anywhere in the statement exist, and that they are valid in the context of the statement. If the checks are successful, SQL Server combines the statement with the stored definition of the view and translates it into a query on the view's underlying tables, as explained in an earlier section. This process is called **view resolution**.

Consider the view definition statement given earlier in this chapter and a query against it:

```
create view hiprice
as select *
from titles
where price > $15
and advance > $5000

select title, type
from hiprice
where type = 'popular_comp'
```

Internally, SQL Server combines the query of *hiprice* with its definition, converting the query to:

```
select title, type
from titles
where price > $15
and advance > $5000
and type = 'popular_comp'
```

In general, you can query any view in any way just as if it were a real table. You can use joins, **group by** clauses, subqueries, and other query techniques on views, in any combination. Note, however, that if the view is defined with an outer join or aggregate function, you may get unexpected results when you query the view. See the "Limitations on View Definitions" section above.

➤ ***Note***

You can use **select** on *text* and *image* columns in views, but the **readtext** and **writetext** commands are not allowed. In addition, you cannot select the *sensitivity* column from a view.

View Resolution

When you define a view, SQL Server checks to make sure that all the tables or views listed in the **from** clause exist, and gives you an error message if there's a problem. Similar checks are performed when you query through the view.

Between the time a view is defined and the time it is used in a statement, things can change. For example, one or more of the tables or views listed in the **from** clause of the view definition may have been dropped. Or one or more of the columns listed in the **select** clause of the view definition may have been renamed.

In order to fully resolve a view, SQL Server checks to make sure that:

- All the tables, views, and columns from which the view was derived still exist.
- The datatype of each column on which a view column depends has not been changed to an incompatible type.
- If the statement is an **update**, **insert**, or **delete**, it does not violate the restrictions on modifying views. These are discussed in a later section of this chapter.

If any of these checks fail, SQL Server issues an error message.

Redefining Views

Unlike many other database management systems, SQL Server allows you to redefine a view without forcing you to redefine other views that depend on it, unless the redefinition makes it impossible for SQL Server to translate the dependent view.

As an example, the *authors* table and three possible views are shown here. Each succeeding view is defined using the view that preceded it: *view2* is created from *view1*, and *view3* is created from *view2*. In this way, *view2* depends on *view1* and *view3* depends on both of the preceding views.

Each view name is followed by the **select** statement used to create it.

view1:

```
create view view1 as
select au_lname, phone from authors
where postalcode like "94%"
```

view2:

```
create view view2 as
select au_lname, phone from view1
where au_lname like "[M-Z]%"
```

view3:

```
create view view3 as
select au_lname, phone from view2
where au_lname = "MacFeather"
```

The *authors* table on which these views are based consists of the columns: *au_id*, *au_lname*, *au_fname*, *phone*, *address*, *city*, *state*, *postalcode*.

You can drop *view2* and replace it with another view, also named *view2*, that contains slightly different selection criteria, such as *au_lname*, *phone* from *view_1* where *au_lname* like "[M-P]". *view3*, which depends on *view2*, is still valid and need not be redefined. When you use a query that references either *view2* or *view3*, view resolution takes place as usual.

If you redefine *view2* so that *view3* cannot be derived from it, *view3* becomes invalid. For example, if another new version of *view2* contains a single column, *au_lname*, rather than the two columns that *view3* expects, *view3* can no longer be used since it cannot derive the *phone* column from the object on which it depends.

However, *view3* still exists, and can be used again by dropping the offending *view2* and re-creating *view2* with both the *au_lname* and the *phone* columns.

In short, you can change the definition of an intermediate view without affecting dependent views so long as the **select** list of the dependent views remains valid. If this rule is violated, a query that references the invalid view will produce an error message.

Renaming Views

You can rename a view with the system procedure **sp_rename**. Here is its syntax:

```
sp_rename objname, newname
```

For example, to rename *titleview* to *bookview*:

```
sp_rename titles_view, bookview
```

Of course, the new name must follow the rules for identifiers. (You cannot use **sp_rename** to specify a new, delimited identifier for a view.) You can change the name only of views that you own. The Database Owner can change the name of any user's view. The view must be in the current database.

Altering or Dropping Underlying Objects

Problems can arise if you change the name of a view's underlying object. Views that depend on a table or view whose name has been changed may work fine for a while. In fact, they work until SQL Server recompiles them. Recompilation takes place for many reasons and without notification to the user—for example, if a database is loaded, or if a user drops and re-creates a table or drops an index. Because of this, attempts to query or modify the view may suddenly cause SQL Server to return error messages.

At that point, you must drop the view and re-create it, so that its text reflects the new name of the object on which it depends. To avoid such problems, the safest course is not to rename any tables or views that are referenced by a view, or to change the definitions of their dependent views when you rename them.

A similar situation arises if a view depends on a table or view that has been dropped. When someone tries to use the view, SQL Server produces an error message. However, if a new table or view is created to replace the one that was dropped, the view will again become usable.

If you define a view with a **select *** clause, and then alter the structure of its underlying tables by adding columns, the new columns will not appear. This is because the asterisk shorthand is interpreted and expanded when the view is first created. To see the new columns through the view, drop the view and re-create it.

Modifying Data Through Views

Although SQL Server places no restrictions at all on retrieving data through views, and although Transact-SQL places fewer restrictions on modifying data through views than other versions of SQL, there are several kinds of data modification operations not allowed through views:

- **update**, **insert**, or **delete** operations that refer to any column in the view that is a computation, that is, a computed column or a built-in function, are not allowed.
- **update**, **insert**, or **delete** operations that refer to a view that includes aggregates or row aggregates, that is, built-in functions and a **group by** clause or a **compute** clause, are not allowed.
- **insert, delete,** and **update** operations that refer to a **distinct** view are not allowed.
- **insert** statements are not allowed unless all NOT NULL columns in the underlying tables or views are included in the view through which you are inserting new rows. SQL Server has no way to supply values for NOT NULL columns in the underlying objects.
- If a view has a **with check option** clause, all rows inserted or updated through the view (or through any derived views) must satisfy the view's selection criteria.
- **delete** statements are not allowed on multitable views.
- **insert** statements are not allowed on multitable views created **with check option**.
- **update** statements are allowed on multitable views **with check option**. The update fails if any of the affected columns appears in the **where** clause, in an expression that includes columns from more than one table.
- **insert** and **update** statements are not allowed on multitable **distinct** views.
- **update** statements cannot specify a value for an IDENTITY column. The table owner, Database Owner, or a System Administrator can **insert** an explicit value into an IDENTITY column after setting **identity_insert on** for the column's base table.
- If you insert or update a row through a multitable view, all affected columns must belong to the same base table.
- **writetext** is not allowed on the *text* and *image* columns in a view.

When you attempt an **update**, **insert**, or **delete** for a view, SQL Server checks to make sure that none of the above restrictions is violated, and that no data integrity rules are violated.

Why can some views be updated and some not? You can best understand the restrictions by examining an example of each kind of view that cannot be updated.

Restrictions on Updating Views

Computed Columns in View Definition

The first restriction applies to columns of views that are derived from computed columns or built-in functions. For example, the *amt_due* column in the view *accounts*, created earlier, is a computed column.

```
create view accounts (title_id, advance, amt_due)
as select titles.title_id, advance, (price *
royalty/100) * total_sales
from titles, roysched
where price > $15
  and advance > $5000
and titles.title_id = roysched.title_id
and total_sales between lorange and hirange
```

The rows visible through *accounts* are:

```
select * from accounts

title_id     advance     amt_due
--------     --------    ---------
PC1035       7,000.00    32,240.16
PC8888       8,000.00     8,190.00
PS1372       7,000.00       809.63
TC3218       7,000.00       785.63

(4 rows affected)
```

updates and **inserts** to the *amt_due* column are not allowed because there is no way to deduce the underlying values for price, royalty, or year-to-date sales from any value you might enter in the *amt_due* column. **delete** operations don't make any sense because there is no underlying value to delete.

group by or *compute* in View Definition

The second restriction applies to all columns in views that contain aggregate values—that is, views whose definition includes a **group by** or **compute** clause. Here is a view defined with a **group by** clause, and the rows seen through it:

```
create view categories (category, average_price)
as select type, avg(price)
from titles
group by type

select * from categories
```

```
category         average_price
-------------    -------------
UNDECIDED                 NULL
business                 13.73
mod_cook                 11.49
popular_comp             21.48
psychology               13.50
trad_cook                15.96

(6 rows affected)
```

It would not make sense to **insert** rows into the view *categories*. To what group of underlying rows would an inserted row belong? Updates on the *average_price* column cannot be allowed because there is no way to know from any value you might enter there how the underlying prices should be changed.

Theoretically, updates to the *category* column and deletes could be allowed, but SQL Server does not support them.

Null Values in Underlying Objects

The third restriction applies to **insert** statements if there are some NOT NULL columns in the tables or views from which the view is derived.

For example, suppose null values are not allowed in a column of a table that underlies a view. Normally, when you **insert** new rows through a view, any columns in underlying tables that are not included in the view are given null values. If null values are not allowed in one or more of these columns, no inserts can be allowed through the view.

Consider the view:

```
create view titleview
as select title_id, price, total_sales
from titles
where type = 'business'
```

Null values are not allowed in the *title* column of the underlying table *titles*, so no **insert** statements can be allowed through *titleview*. Although the *title* column doesn't even exist in the view, its prohibition of null values makes any inserts into the view illegal.

Similarly, if the *title_id* column has a unique index, updates or inserts that would duplicate any values in the underlying table are rejected, even if the entry doesn't duplicate any value in the view.

Views Created *with check option*

The fourth restriction determines what types of modifications you can make through views with check options. If a view has a **with check option** clause, each row inserted or updated through the view must be visible within the view. This is true whether you insert or update the view directly, or indirectly through another derived view.

Multitable Views

The fifth restriction determines what types of modifications you can make through views that join columns from multiple tables. SQL Server prohibits **delete** statements on multitable views, but does allow **update** and **insert** statements that would not be allowed in other systems.

You can insert or update a multitable view if:

- The view has no **with check option** clause.
- All columns being inserted or updated belong to the same base table.

For example, consider the following view, which includes columns from both *titles* and *publishers* and has no **with check option** clause:

```
create view multitable_view
as select title, type, titles.pub_id, state
from titles, publishers
where titles.pub_id = publishers.pub_id
```

A single **insert** or update statement can specify values **either** for the columns from *titles* **or** for the column from *publishers*. The following **update** statement succeeds:

```
update multitable_view
set type = “user_friendly”
where type = “popular_comp”
```

But the statement below fails because it affects columns from both *titles* and *publishers*:

```
update multitable_view
set type = “cooking_trad”,
state = “WA”
where type = “trad_cook”
```

Views That Include IDENTITY Columns

The last restriction determines what types of modifications you can make to views that include IDENTITY columns. By definition, IDENTITY columns are not updatable. Updates through a view cannot specify an IDENTITY column value.

Inserts to IDENTITY columns are restricted to the table owner, the Database Owner, and the System Administrator. To enable such inserts through a view, **set identity_insert on** for the column's base table. It is not sufficient to **set identity_insert on** for the view through which you are inserting.

Dropping Views

To delete a view from the database, use the **drop view** command. The syntax is:

```
drop view [[database.]owner.]view_name
     [, [[database.]owner.]view_name]...
```

As indicated, you can drop more than one view at a time. Only its owner (or the database owner) can drop a view.

Here is how to drop the view *hiprice*:

```
drop view hiprice
```

When you issue the **drop view** command, information about the named view is deleted from the system tables *sysprocedures*, *sysobjects*, *syscolumns*, *syscomments*, *sysprotects*, and *sysdepends*. All privileges on that view are also deleted.

If a view depends on a table or on another view that has been dropped, SQL Server produces an error message if anyone tries to use the view. If a new table or view is created to replace the one that has been dropped, with the same name, the view again becomes usable, as long as the columns referenced in the view definition exist.

Using Views As Security Mechanisms

Permission to access the subset of data in a view must be explicitly granted or revoked, regardless of the permissions in force on the view's underlying tables. Data in an underlying table that is not included in the view is hidden from users who are authorized to access the view but not to access the underlying table.

For example, you may not want some users to access the columns that have to do with money and sales in the *titles* table. You can create a view of the *titles* table that omits those columns, and then give all users permission on the view, and give only the Sales Department permission on the table. For example:

```
revoke all on titles to public
grant all on bookview to public
grant all on titles to sales
```

For information about how to **grant** or **revoke** permissions, see the *Security Features User's Guide*.

Getting Information About Views

Several system procedures provide information from the system tables about views.

You can get a report on a view with the system procedure **sp_help**. For example:

```
sp_help hiprice
Name           Owner   type    Created_on
---------      ------  -----   -------------------
hiprice        dbo     view    Feb 12 1987 11:57AM

Data_located_on_segment        When_created
------------------------------ --------------------

Column_name      Type      Length   Precision    Scale
-----------     -------  ------   ---------   -----
title_id        tid         6         NULL    NULL
title           varchar    80         NULL    NULL
type            char       12         NULL    NULL
pub_id          char        4         NULL    NULL
price           money       8         NULL    NULL
advance         money       8         NULL    NULL
royalty         int         4         NULL    NULL
total_sales     int         4         NULL    NULL
notes           varchar   200         NULL    NULL
pubdate         datetime    8         NULL    NULL
```

```
Null       Default_name Rule_name    Identity
------     ------------ ---------    --------
0          NULL              NULL            0
0          NULL              NULL            0
0          NULL              NULL            0
1          NULL              NULL            0
1          NULL              NULL            0
1          NULL              NULL            0
1          NULL              NULL            0
1          NULL              NULL            0
1          NULL              NULL            0
0          NULL              NULL            0

No defined keys for this object.

(return status = 0)
```

To display the text of the **create view** statement, execute the system procedure **sp_helptext**:

```
sp_helptext hiprice

----------
        1

(1 row affected)

text
--------------------------------------------
create view hiprice

as select *
from titles
where price > $15
and advance > $5000

(1 row affected, return status = 0)
```

The system procedure **sp_depends** lists all the objects that the view or table references in the current database, and it lists all the objects that reference that view or table. Here's an example:

```
sp_depends titles

Things inside the current database that reference
the object.
```

```
object           type
-------------    ---------------------------
dbo.hiprice      view
dbo.titleview    view
dbo.reptq1       stored procedure
dbo.reptq2       stored procedure
dbo.reptq3       stored procedure

(return status = 0)
```

For complete information about system procedures, see the *SQL Server Reference Manual.*

Part 2:Advanced Topics

10 Using the Built-In Functions in Queries

Transact-SQL provides several different types of built-in functions that return different kinds of information from the database. These functions are Transact-SQL extensions to SQL.

You can use built-in functions in the **select** list, in the **where** clause, and anywhere else an expression is allowed. You can also use them as part of a stored procedure or program. SQL Server provides a variety of built-in functions.

This chapter discusses:

- System functions, most of which return information from the system tables.
- String functions, which manipulate *char*, *nchar*, *varchar*, *nvarchar*, *binary*, and *varbinary* values.
- Text functions, which manipulate *text* and *image* values.
- Mathematical functions, which perform trigonometric, geometric, and other kinds of number handling.
- Date functions, which manipulate *datetime* and *smalldatetime* values.
- Datatype conversion functions, which convert expressions from one datatype to another, and which format dates in a wide variety of styles.

System Functions

The system functions return special information from the database. Many of them provide a shorthand way of querying the system tables.

The general syntax of the system functions is:

```
select function_name(argument[s])
```

The system functions can be used in the **select** list, in the **where** clause, and anywhere an expression is allowed.

For example, to find the user identification number of your coworker who logs in as "harold," type:

```
select user_id("harold")
```

Assuming that "harold"'s user ID is 13, the result is:

```
------------
          13

(1 row affected)
```

Generally, the name of the function tells you what kind of information is returned.

The **system function** **user_name** takes an ID number as its argument and returns the user's name:

```
select user_name(13)

---------
harold

(1 row affected)
```

To find the name of the current user, that is, your name, the argument is omitted:

```
select user_name()

---------
dbo

(1 row affected)
```

Note that the System Administrator becomes the Database Owner in any database they are using by assuming the **server user ID** 1. A "guest" user is always given the server user ID -1. Inside a database, the **user_name** of the Database Owner is always "dbo"; his or her user ID is 1. Inside a database, the "guest" user ID is always 2.

This list gives the name of each system function, the argument it takes, and the result it returns:

Table 10-1: System functions, arguments, and results

Function	Argument	Result
col_name	(*object_id*, *column_id* [, *database_id*])	Returns the column name.
col_length	(*object_name*, *column_name*)	Returns the defined length of column. Use **datalength** to see the actual data size.

Table 10-1: System functions, arguments, and results (continued)

Function	Argument	Result
curunreservedpgs	(*dbid, lstart, unreservedpgs*)	Returns the number of free pages in a disk piece. If the database is open, the value is taken from memory; if the database is not in use, the value is taken from the *unreservedpgs* column in *sysusages*.
data_pgs	(*object_id*, {*doampg* | *ioampg*})	Returns the number of pages used by the table (*doampg*) or index (*ioampg*). The result does not include pages used for internal structures.
datalength	(*expression*)	Returns the length of expression in bytes. *expression* is usually a column name. If *expression* is a character constant, it must be enclosed in quotes.
db_id	([*database_name*])	Returns the database ID number. *database_name* must be a character expression; if it is a constant expression, it must be enclosed in quotes. If no *database_name* is supplied, **db_id** returns the ID number of the current database.
db_name	([*database_id*])	Returns the database name. *database_id* must be a numeric expression. If no *database_id* is supplied, **db_name** returns the name of the current database.
host_id	( )	Returns the host process ID of the client process (not the SQL Server process).
host_name	( )	Returns the current host computer name of the client process (not the SQL Server process).
index_col	(*object_name*, *index_id*, *key_#* [, *user_id*])	Returns the name of the indexed column; returns NULL if *object_name* is not a table or view name.

Table 10-1: System functions, arguments, and results (continued)

Function	Argument	Result
isnull	(*expression1*, *expression2*)	Substitutes the value specified in *expression2* when *expression1* evaluates to NULL.The datatypes of the expressions must convert implicitly, or you must use the **convert** function.
lct_admin	({{ **"lastchance"** | **"logfull"** | **"unsuspend"**} , *database_id*} | **"reserve"**, *log_pages*})	Manages the log segment's last-chance threshold. **lastchance** creates a last-chance threshold in the specified database. **logfull** returns 1 if the last-chance threshold has been crossed in the specified database and 0 if it has not. **unsuspend** awakens suspended tasks in the database and disables the last-chance threshold if that threshold has been crossed. **reserve** returns the number of free log pages required to successfully dump a transaction log of the specified size.
object_id	(*object_name*)	Returns the object ID.
object_name	(*object_id*[, *database_id*])	Returns the object name.
proc_role	(**"sa_role"** | **"sso_role"** | **"oper_role"**)	Checks to see if the invoking user possesses the correct role to execute the procedure. Returns 1 if the invoker has the required role. Otherwise, returns 0.
reserved_pgs	(*object_id*, {*doampg* | *ioampg*})	Returns the number of pages allocated to table or index. This function **does** report pages used for internal structures.
rowcnt	(*doampg*)	Returns the number of rows in a table (estimate).

Table 10-1: System functions, arguments, and results (continued)

Function	Argument	Result
show_role	( )	Returns the user's current active roles, if any (sa_role, sso_role, or oper_role). If the user has no roles, returns NULL.
suser_id	([*server_user_name*])	Returns the server user's ID number from *syslogins*. If no *server_user_name* is supplied, it returns the server ID of the current user.
suser_name	([*server_user_id*])	Returns the server user's name. Server user's IDs are stored in *syslogins*. If no *server_user_id* is supplied, it returns the name of the current user.
used_pgs	(*object_id, doampg, ioampg*)	Returns the total number of pages used by a table and its clustered index.
tsequal	(*timestamp, timestamp2*)	Compares *timestamp* values to prevent update on a row that has been modified since it was selected for browsing. *timestamp* is the timestamp of the browsed row; *timestamp2* is the timestamp of the stored row. Allows you to use browse mode without calling DB-Library. (See "Browse Mode.")
user		Returns the user's name.
user_id	([*user_name*])	Returns the user's ID number. Reports the number from *sysusers* in the current database. If no *user_name* is supplied, it returns the ID of the current user.
user_name	([*user_id*])	Returns the user's name, based on the user's ID in the current database. If no *user_id* is supplied, it returns the name of the current user.

Table 10-1: System functions, arguments, and results (continued)

Function	Argument	Result
valid_name	(*character_expression*)	Returns 0 if the *character expression* is not a valid identifier (illegal characters or more than 30 bytes long), a nonzero number if it is a valid identifier.
valid_user	(*server_user_id*)	Returns 1 if the specified ID is a valid user or alias in at least one database on this SQL Server. You must have the **sa_role** or **sso_role** role to use this function on a *server_user_id* other than your own.

When the argument to a system function is optional, the current database, host computer, server user, or database user is assumed. With the exception of **user**, built-in functions are always used with parentheses even if the argument is NULL.

Examples of Using System Functions

col_length

This query finds the length of the *title* column in the *titles* table (the "x=" is included so that the result has a column heading):

```
select x = col_length("titles", "title")

 x
--------
      80

 (1 row affected)
```

datalength

In contrast to **col_length**, which finds the defined length of a column, **datalength** reports the actual length, in bytes, of the data stored in each row. Use this function on *varchar*, *nvarchar*, *varbinary*, *text*, and *image* datatypes, since they can store variable lengths. **datalength** of any NULL data returns NULL. All other datatypes report their defined

length. Here's an example that finds the length of the *pub_name* column in the *publishers* table:

```
select Length=datalength(pub_name), pub_name
from publishers
Length  pub_name
------  ------------------------
13      New Age Books
16      Binnet & Hardley
20      Algodata Infosystems

(3 rows affected)
```

isnull

This query finds the average of the prices of all titles, substituting the value ''$10.00'' for all NULL entries in *price*:

```
select avg(isnull(price,$10.00))
from titles
------------
       14.24

(1 row affected)
```

user_name

This query finds the row in *sysusers* where the name is equal to the result of applying the system function **user_name** to user ID 1:

```
select name
from sysusers
where name = user_name(1)
name
------------------------
dbo

(1 row affected)
```

String Functions

String functions are used for various operations on character strings or expressions. A few string functions can be used on binary data as well as on character data. You can also concatenate binary data or character strings or expressions.

String built-in functions return values commonly needed for operations on character data. String function names are not keywords.

The syntax for string functions takes the general form:

```
select function_name(arguments)
```

You can concatenate binary or character expressions like this:

```
select (expression + expression [+ expression]...)
```

When concatenating noncharacter, nonbinary expressions, you must use the **convert** function:

```
select "The price is " + convert(varchar(12),price)
from titles
```

Most string functions can be used only on *char*, *nchar*, *varchar*, and *nvarchar* datatypes and on datatypes which implicitly convert to *char* or *varchar*. A few string functions can also be used on *binary* and *varbinary* data. **patindex** can be used on *text*, *char*, *nchar*, *varchar*, and *nvarchar* columns.

Concatenation can be used on *binary* and *varbinary* columns and *char*, *nchar*, *varchar*, and *nvarchar* columns. However, you **cannot** concatenate *text* columns.

String functions can be nested, and they can be used anywhere an expression is allowed. When you use constants with a string function, enclose them in single or double quotes.

Table 10-2 lists the arguments used in string functions. If a function takes more than one expression of the same type, the arguments are numbered, as *char_expr1*, *char_expr2*.

Table 10-2: Arguments used in string functions

Argument Type	Can Be Replaced By
char_expr	A character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type. Functions which accept *text* column names are noted in the explanation. Constant expressions must be enclosed in quotation marks.
expression	A binary or character column name, variable or constant expression. Can be *char*, *varchar*, *nchar* or *nvarchar* data, as for *char_expr*, plus *binary* or *varbinary*.
pattern	A character expression of *char*, *nchar*, *varchar*, or *nvarchar* datatype that may include any of the pattern-match wildcards supported by SQL Server.

Table 10-2: Arguments used in string functions (continued)

Argument Type	Can Be Replaced By
approx_numeric	Any approximate numeric (*float*, *real*, or *double precision*) column name, variable, or constant expression.
integer_expr	Any integer (such as *tinyint*, *smallint* or *int*), column name, variable, or constant expression. Maximum size ranges are noted, as they apply.
start	An *integer_expr.*
length	An *integer_expr.*

Each function also accepts arguments that can be implicitly converted to the specified type. For example, functions that accept approximate numeric expressions also accept integer expressions. SQL Server automatically converts the argument to the desired type.

Table 10-3 lists function names, arguments, and results.

Table 10-3: String functions, arguments and results

Function	Argument	Result
ascii	(*char_expr*)	Returns the ASCII code for the first character in the expression.
char	(*integer_expr*)	Converts a single-byte *integer* value to a *character* value. **char** is usually used as the inverse of **ascii**. *integer_expr* must be between 0 and 255. Returns a *char* datatype. If the resulting value is the first byte of a multibyte character, the character may be undefined.
charindex	(*expression1*, *expression2*)	Searches *expression2* for the first occurrence of *expression1* and returns an integer representing its starting position. If *expression1* is not found, returns 0. If *expression1* contains wildcard characters, **charindex** treats them as literals.
char_length	(*char_expr*)	Returns an integer representing the number of characters in a character expression or *text* value. For variable-length data, **char_length** strips the expression of trailing blanks before counting the number of characters. For multibyte character sets, the number of characters in the expression is usually less than the number of bytes; use the system function datalength to determine the number of bytes.
difference	(*char_expr1*, *char_expr2*)	Returns an integer representing the difference between two **soundex** values. See **soundex**, below.
lower	(*char_expr*)	Converts uppercase to lowercase. Returns a character value.

Table 10-3: String functions, arguments and results (continued)

Function	Argument	Result
ltrim	(*char_expr*)	Removes leading blanks from the character expression. Only values equivalent to the space character in the SQL special character specification are removed.
patindex	("%*pattern*%", *char_expr* [**using** {**bytes** | **chars** | **characters**}])	Returns an integer representing the starting position of the first occurrence of *pattern* in the specified character expression, zero if *pattern* is not found. By default, **patindex** returns the offset in characters. To return the offset in bytes, that is, multibyte character strings, specify **using bytes**. The '%' wildcard character must precede and follow *pattern*, except when searching for first or last characters. See "Wildcard Characters" in the *SQL Server Reference Manual* for a description of the wildcard characters that can be used in *pattern*. Can be used on *text* data.
replicate	(*char_expr*, *integer_expr*)	Returns a string with the same datatype as *char_expr*, containing the same expression repeated the specified number of times or as many times as will fit into a 255 byte space, whichever is less.
reverse	(*char_expr*)	Returns the reverse of *char_expr*; if *char_expr* is "abcd", it returns "dcba".
right	(*char_expr*, *integer_expr*)	Returns the part of the character expression starting at the specified number of characters from the right. Return value has the same datatype as the character expression.
rtrim	(*char_expr*)	Removes trailing blanks. Only values equivalent to the space character in the SQL special character definition are removed.
soundex	(*char_expr*)	Returns a four-character **soundex** code for character strings that are composed of a contiguous sequence of valid single- or double-byte roman letters.
space	(*integer_expr*)	Returns a string with the indicated number of single-byte spaces.
str	(*approx_numeric* [, *length* [, *decimal*]])	Returns a character representation of the floating point number. *length* sets the number of characters to be returned (including the decimal point, all digits to the right and left of the decimal point, and blanks); *decimal* sets the number of decimal digits to be returned. *length* and *decimal* are optional. If given, they must be non-negative. Default *length* is 10; default *decimal* is 0. **str** rounds the decimal portion of the number so that the results fit within the specified *length*.

Table 10-3: String functions, arguments and results (continued)

Function	Argument	Result
stuff	(*char_expr1*, *start*, *length*, *char_expr2*)	Delete *length* characters from *char_expr1* at *start*, then insert *char_expr2* into *char_expr1* at *start*. To delete characters without inserting other characters, *char_expr2* should be NULL, not " ", which indicates a single space.
substring	(*expression*, *start*, *length*)	Returns part of a character or binary string. *start* specifies the character position at which the substring begins. *length* specifies the number of characters in the substring.
upper	(*char_expr*)	Converts lowercase to uppercase. Returns a character value.

Examples of Using String Functions

charindex and *patindex*

The **charindex** and **patindex** functions return the starting position of a pattern you specify. Both take two arguments, but work slightly differently, since **patindex** can use wildcard characters, but **charindex** does not. **charindex** can be used only on *char*, *nchar*, *varchar*, and *nvarchar* columns; **patindex** works on these columns plus *text* columns.

Both functions take two arguments. The first is the pattern whose position you want. With **patindex**, you must include percent signs before and after the pattern, unless you're looking for the pattern as the first (omit the preceding %) or last (omit the trailing %) characters in a column. For **charindex**, the pattern cannot include wildcard characters. The second argument is a character expression, usually a column name, in which SQL Server searches for the specified pattern.

To find the position at which the pattern "wonderful" begins in a certain row of the *notes* column of the *titles* table using both functions, type this query:

```
select charindex("wonderful", notes),
    patindex("%wonderful%", notes)
from titles
where title_id = "TC3218"

------------ ------------
          46           46

(1 row affected)
```

If you do not restrict the rows to be searched, the query returns all rows in the table and reports zero values for those rows which don't contain the pattern. In the following example, **patindex** finds all the rows in *sysobjects* which start with "sys" and whose fourth character is a, b, c, or d:

```
select name
from sysobjects
where patindex("sys[a-d]%", name) > 0
name
------------------------------
sysalternates
sysattributes
syscharsets
syscolumns
syscomments
sysconfigures
sysconstraints
syscurconfigs
sysdatabases
sysdepends
sysdevices

(11 rows affected)
```

str

The **str** function converts numbers to characters, with optional arguments for specifying the length of the number (including sign, decimal point, and digits to the right and left of the decimal point), and the number of places after the decimal point.

Length and decimal arguments to **str** (if supplied) must be positive. The default length is 10. The default decimal is 0. The length should be long enough to accommodate the decimal point and the number's sign. The decimal portion of the result is rounded to fit within the specified length. If the integer portion of the number does not fit within the length, however, **str** returns a row of asterisks of the specified length.

For example:

```
select str(123.456, 2, 4)
```

```
--
**

(1 row affected)
```

A short *approx_numeric* is right-justified in the specified length, and a long *approx_numer* is truncated to the specified number of decimal places.

stuff

The **stuff** function inserts a string into another string. It deletes a specified length of characters in *expr1* at the start position. It then inserts *expr2* string into *expr1* string at the start position. If the start position or the length is negative, a NULL string is returned.

If the start position is longer than *expr1*, a NULL string is returned. If the length to delete is longer than *expr1*, it is deleted through the last character in *expr1*. For example:

```
select stuff("abc", 2, 3, "xyz")

----
axyz

(1 row affected)
```

To use **stuff** to delete a character, replace *expr2* with NULL, not with empty quotation marks. Using " " to specify a null character replaces it with a space.

```
select stuff("abcdef", 2, 3, null)

---
aef

(1 row affected)
select stuff("abcdef", 2, 3, "")

----
a ef

(1 row affected)
```

soundex and *difference*

The **soundex** function converts a character string to a four-digit code for use in a comparison. Vowels are ignored in the comparison.

Nonalphabetic characters terminate the **soundex** evaluation. This function always returns some value. These two names have identical **soundex** codes:

```
select soundex ("smith"), soundex ("smythe")

----- -----
S530  S530
```

The **difference** function compares the **soundex** values of two strings and evaluates the similarity between them, returning a value from 0 to 4. A value of 4 is the best match. For example:

```
select difference("smithers", "smothers")

---------
        4

(1 row affected)
select difference("smothers", "brothers")

 ---------
         2

(1 row affected)
```

Most of the remaining string functions are easy to use and to understand. For example:

Table 10-4: String function examples

Statement	Result
select right("abcde", 3)	cde
select right("abcde", 6)	abcde
select upper("torso")	TORSO
select ascii("ABC")	65

substring

The following example uses the **substring** function. It displays the last name and first initial of each author, for example, "Bennet A".

```
select au_lname, substring(au_fname, 1, 1)
from authors
```

The **substring** function does what its name implies—it returns a portion of a character or binary string.

The **substring** function always takes three arguments. The first can be a character or binary string, a column name, or a string-valued expression that includes a column name. The second argument

specifies the position at which the substring should begin. The third specifies the length, in number of characters, of the string to be returned.

The syntax of the **substring** function looks like this:

substring(*expression*, *start*, *length*)

For example, here is how to specify the second, third, and fourth characters of the string constant "abcdef":

```
select x = substring("abcdef", 2, 3)

x
---------
bcd
```

Concatenation

You can concatenate binary or character expressions—combine two or more character or binary strings, character or binary data, or a combination of them—with the + string concatenation operator.

If you're concatenating character strings, enclose each character expression in single or double quotes.

The concatenation syntax is:

select (*expression* + *expression* [+ *expression*]...)

Here's how to combine two character strings:

```
select ("abc" + "def")

-------
abcdef

(1 row affected)
```

This query displays California author names under the column heading *Moniker* in last name-first name order, with a comma and space after the last name:

```
select Moniker = (au_lname + ", " + au_fname)
from authors
where state = "CA"
```

```
Moniker
------------------------------------------------
White, Johnson
Green, Marjorie
Carson, Cheryl
O'Leary, Michael
Straight, Dick
Bennet, Abraham
Dull, Ann
Gringlesby, Burt
Locksley, Chastity
Yokomoto, Akiko
Stringer, Dirk
MacFeather, Stearns
Karsen, Livia
Hunter, Sheryl
McBadden, Heather

(15 rows affected)
```

To concatenate numeric or *datetime* datatypes, you must use the **convert** function:

```
select "The due date is " + convert(varchar(30),
    pubdate)
from titles
where title_id = "BU1032"

--------------------------------------
The due date is Jun 12 1985 12:00AM

(1 row affected)
```

Concatenation and the Empty String

The empty string (“” or ‘’) is evaluated as a single space. This statement:

```
select "abc" + "" + "def"
```

produces:

```
abc def
```

Nested String Functions

The string functions can be nested. For example, to display the last name and the first initial of each author, with a comma after the last name and a period after the first name, you can type:

```
select (au_lname + "," + " " + substring(au_fname,
1, 1) + ".")
from authors
where city = "Oakland"

------------------------------------------
Green, M.
Straight, D.
Stringer, D.
MacFeather, S.
Karsen, L.

(5 rows affected)
```

To display the *pub_id* and the first two characters of each *title_id* for books over $20, type:

```
select substring(pub_id + title_id, 1, 6)
from titles
where price > $20

--------------
1389PC
0877PS
0877TC

(3 rows affected)
```

Text Functions

Text built-in functions are used for operations on *text* and *image* data. Text function names, arguments, and results are listed in *Table 10-5*.

Table 10-5: Built-in text functions for text and image data

Function	Argument	Result
patindex	("%*pattern*%", *char_expr* [**using** {**bytes** | **chars** | **characters**}])	Returns an integer value representing the starting position of the first occurrence of *pattern* in the specified character expression, zero if *pattern* is not found. By default, **patindex** returns the offset in characters; to return the offset in bytes for multibyte character strings, specify **using bytes**. The % wildcard character must precede and follow *pattern*, except when searching for first or last characters. See "Wildcard Characters" in the *SQL Server Reference Manual* for a description of the wildcard characters that can be used in *pattern*.

Table 10-5: Built-in text functions for text and image data (continued)

Function	Argument	Result
textptr	(*text_columname*)	Returns the text pointer value, a 16-byte binary value. The text pointer is checked to ensure that it points to the first text page.
textvalid	("*table_name..col_name*", *textpointer*)	Checks if a given text pointer is valid. Note that the identifier for a *text* or *image* column must include the table name. Returns 1 if the pointer is valid, 0 if the pointer is invalid.
set textsize	{ *n* | 0 }	Specifies the limit, in bytes, of the *text* or *image* data to be returned with a **select** statement. The current setting is stored in the *@@textsize* global variable. *n* is an integer that specifies the limit on the number of bytes to return; 0 restores the default limit of 32K.

In addition to these functions, **datalength** (described in "System Functions" on page 10-1) works on *text* columns. You can also use the *@@textcolid*, *@@textdbid*, *@@textobjid*, *@@textptr*, and *@@textsize* global variables to manipulate text and image data.

Examples of Using Text Functions

This example uses the **textptr** function to locate the *text* column, *blurb*, associated with *title_id* BU7832 in table *texttest*. The text pointer, a 16-byte binary string, is put into a local variable, *@val*, and supplied as a parameter to the **readtext** command. **readtext** returns 5 bytes starting at the second byte, with an offset of 1.

```
create table texttest
(title_id varchar(6),blurb text null, pub_id
    char(4))

insert texttest values ("BU7832", "Straight Talk
    About Computers is an annotated analysis of
    what computers can do for you: a no-hype guide
    for the critical user", "1389")

declare @val varbinary(16)
select @val = textptr(blurb) from texttest
where title_id = "BU7832"
readtext texttest.blurb @val 1 5
```

The **textptr** function returns a 16-byte binary string. It is a good idea to put this string into a local variable, as in the preceding example, and use it by reference.

An alternative to using the **textptr** function in the preceding example is the *@@textptr* global variable:

```
create table texttest
(title_id varchar(6),blurb text null, pub_id
    char(4))

insert texttest values ("BU7832", "Straight Talk
    About Computers is an annotated analysis of
    what computers can do for you: a no-hype guide
    for the critical user", "1389")

readtext texttest.blurb @@textptr 1 5
```

The value of *@@textptr* is set from the last **insert** or **update** to any *text* or *image* field by the current SQL Server process. Inserts and updates by other processes do not affect the current process.

Explicit conversion using the **convert** function is supported from *text* to *char*, *nchar*, *varchar* or *nvarchar*, and from *image* to *varbinary* or *binary*, but the *text* or *image* data is truncated to 255 bytes. Conversion of *text* or *image* to datatypes other than these is not supported, implicitly or explicitly.

Mathematical Functions

Mathematical built-in functions return values commonly needed for operations on mathematical data.

The mathematical functions take the general form:

***function_name*(*arguments*)**

The chart below lists the types of arguments that are used in the built-in mathematical functions:

Table 10-6: Arguments used in mathematical functions

Argument Type	Can Be Replaced By
approx_numeric	Any approximate numeric (*float*, *real*, or *double precision*) column name, variable, constant expression, or a combination of these.

Table 10-6: Arguments used in mathematical functions (continued)

Argument Type	Can Be Replaced By
integer	Any integer (*tinyint*, *smallint* or *int)* column name, variable, constant expression, or a combination of these.
numeric	Any exact numeric (*numeric*, *dec*, *decimal*, *tinyint*, *smallint*, or *int*), approximate numeric (*float*, *real*, or *double precision*), or *money* column, variable, constant expression, or a combination of these.
power	Any exact numeric, approximate numeric, or *money* column, variable, or constant expression, or a combination of these.

Each function also accepts arguments that can be implicitly converted to the specified type. For example, functions that accept approximate numeric types also accept integer types. SQL Server automatically converts the argument to the desired type.

If a function takes more than one expression of the same type, the expressions are numbered (for example, *approx_numeric1*, *approx_numeric2*).

Following are the mathematical functions, their arguments, and the results they return:

Table 10-7: Mathematical functions

Function	Argument	Result
abs	(*numeric*)	Returns the absolute value of a given expression. Results are of the same type, and have the same precision and scale, as the numeric expression.
acos	(*approx_numeric*)	Returns the angle (in radians) whose cosine is the specified value.
asin	(*approx_numeric*)	Returns the angle (in radians) whose sine is the specified value.
atan	(*approx_numeric*)	Returns the angle (in radians) whose tangent is the specified value.
atn2	(*approx_numeric1*, *approx_numeric2*)	Returns the angle (in radians) whose tangent is (*approx_numeric1* / *approx_numeric2*).

Table 10-7: Mathematical functions (continued)

Function	Argument	Result
ceiling	(*numeric*)	Returns the smallest integer greater than or equal to the specified value. Results are of the same type as the numeric expression. For *numeric* and *decimal* expressions, the results have a precision equal to that of the expression and a scale of 0.
cos	(*approx_numeric*)	Returns the trigonometric cosine of the specified angle (in radians).
cot	(*approx_numeric*)	Returns the trigonometric cotangent of the specified angle (in radians).
degrees	(*numeric*)	Converts radians to degrees. Results are of the same type as the numeric expression. For *numeric* and *decimal* expressions, the results have an internal precision of 77 and a scale equal to that of the expression. When *money* datatype is used, internal conversion to *float* may cause loss of precision.
exp	(*approx_numeric*)	Returns the exponential value of the specified value.
floor	(*numeric*)	Returns the largest integer less than or equal to the specified value. Results are of the same type as the numeric expression. For expressions of type *numeric* or *decimal*, the results have a precision equal to that of the expression and a scale of 0.
log	(*approx_numeric*)	Returns the natural logarithm of the specified value.
log10	(*approx_numeric*)	Returns the base 10 logarithm of the specified value.
pi	()	Returns the constant value of 3.1415926535897931.
power	(*numeric*, *power*)	Returns the value of *numeric* to the power of *power*. Results are of the same type as *numeric*. For expressions of type *numeric* or *decimal*, the results have a precision of 77 and a scale equal to that of the expression.

Table 10-7: Mathematical functions (continued)

Function	Argument	Result
radians	(*numeric_expr*)	Converts degrees to radians. Results are of the same type as *numeric*. For expressions of type *numeric* or *decimal*, the results have an internal precision of 77 and a scale equal to that of the *numeric* expression. When the *money* datatype is used, internal conversion to *float* may cause loss of precision.
rand	([*integer*])	Returns a random float value between 0 and 1, using the optional integer as a seed value.
round	(*numeric*, *integer*)	Rounds the *numeric* so that it has *integer* significant digits. A positive *integer* determines the number of significant digits to the right of the decimal point; a negative *integer*, the number of significant digits to the left of the decimal point. Results are of the same type as the numeric expression and, for *numeric* and *decimal* expressions, have an internal precision of 77 and scale equal to that of the numeric expression.
sign	(*numeric*)	Returns the sign of *numeric*: positive (+1), zero (0), or negative (-1). Results are of the same type, and have the same precision and scale, as the numeric expression.
sin	(*approx_numeric*)	Returns the trigonometric sine of the specified angle (measured in radians).
sqrt	(*approx_numeric*)	Returns the square root of the specified value.
tan	(*approx_numeric*)	Returns the trigonometric tangent of the specified angle (measured in radians).

Examples of Using Mathematical Functions

The mathematical built-in functions operate on numeric data. Certain functions require integer data and others approximate numeric data. A number of functions operate on exact numeric, approximate numeric, *money*, and *float* types. The precision of built-in operations on *float* type data is 6 decimal places by default.

Error traps are provided to handle domain or range errors of the mathematical functions. Users can **set** the **arithabort** and **arithignore** options to determine how domain errors are handled. For more

information about these options, see the section "Conversion Errors" on page 10-34.

Some simple examples of mathematical functions follow:

Table 10-8: Examples of mathematical functions

Statement	Result
select floor(123)	123
select floor(123.45)	123.000000
select floor(1.2345E2)	123.000000
select floor(-123.45)	-124.000000
select floor(-1.2345E2)	-124.000000
select floor($123.45)	123.00
select ceiling(123.45)	124.000000
select ceiling(-123.45)	-123.000000
select ceiling(1.2345E2)	124.000000
select ceiling(-1.2345E2)	-123.000000
select ceiling($123.45)	124.00
select round(123.4545, 2)	123.4500
select round(123.45, -2)	100.00
select round(1.2345E2, 2)	123.450000
select round(1.2345E2, -2)	100.000000

The **round(numeric, integer)** function always returns a value. If *integer* is negative and exceeds the number of significant digits in *numeric*, SQL Server rounds only the most significant digit. For example:

```
select round(55.55, -3)
```

returns a value of 100.000000. (The number of zeros to the right of the decimal point is equal to the scale of *numeric*.)

Date Functions

The date built-in functions are used to display information about dates and times. They manipulate *datetime* and *smalldatetime* values, performing arithmetic operations on them.

The date functions can be used in the **select** list, in the **where** clause, or wherever an expression can be used.

Values with the *datetime* datatype are stored internally by SQL Server as two 4-byte integers. The first 4 bytes store the number of days before or after the base date, January 1, 1900. The base date is the system's reference date. *datetime* values earlier than January 1, 1753

are not permitted. The other 4 bytes of the internal datetime representation store the time of day to an accuracy of 1/300 second.

The *smalldatetime* datatype stores dates and times of day with less precision than *datetime*. *smalldatetime* values are stored as two 2-byte integers. The first 2 bytes store the number of days after January 1, 1900. The other 2 bytes store the number of minutes since midnight. Dates range from January 1, 1900 to June 6, 2079, with accuracy to the minute.

The default display format for dates looks like this:

```
Apr 15 1987 10:23PM
```

See the section on **convert**, later in this chapter, for information on changing the display format for *datetime* or *smalldatetime*. When you enter *datetime* or *smalldatetime* values, enclose them in single or double quotes. SQL Server recognizes a wide variety of datetime data entry formats. For more information about *datetime* and *smalldatetime* values, see Chapter 7, "Creating Databases and Tables," and Chapter 8, "Adding, Changing, and Deleting Data."

The following table lists the date functions and the results they produce:

Table 10-9: Date functions

Function	Argument	Result
getdate	( )	Current system date and time
datename	(*datepart*, *date*)	Part of a *datetime* or *smalldatetime* value as an ASCII string
datepart	(*datepart*, *date*)	Part of a *datetime* or *smalldatetime* value for example, the month, as an integer
datediff	*(datepart, date, date)*	The amount of time between the second and first of two dates, converted to the specified date part, for example, months, days, hours
dateadd	*(datepart, number, date)*	A date produced by adding date parts to another date

The **datename**, **datepart**, **datediff**, and **dateadd** functions take as arguments a **date part**—the year, month, hour, and so on. The following table lists each date part, its abbreviation (if there is one), and the possible

integer values for that date part. The **datename** function produces ASCII values where appropriate, such as for day of week.

Table 10-10:Date parts

Date Part	Abbreviation	Values
year	**yy**	1753-9999
quarter	**qq**	1 - 4
month	**mm**	1 - 12
week	**wk**	1 - 366
day	**dd**	1 - 31
dayofyear	**dy**	1 - 54
weekday	**dw**	1 - 7 (1 is Sunday in us_english)
hour	**hh**	0 - 23
minute	**mi**	0 - 59
second	**ss**	0 - 59
millisecond	**ms**	0 - 999

Note that the values of the weekday date part are affected by the language setting.

Get Current Date: *getdate*

The **getdate** function produces the current date and time in SQL Server internal format for *datetime* and *smalldatetime* values. **getdate** takes the NULL argument, ().

To find the current system date and time, type:

```
select getdate()

--------------------------
Jul 29 1991  2:50  PM

(1 row affected)
```

You might use **getdate** in designing a report, so that the current date and time are printed every time the report is produced. **getdate** is also useful for functions such as logging the time a transaction occurred on an account.

Find Date Parts As Numbers or Names

The **datepart** and **datename** functions produce the specified part of a *datetime* or *smalldatetime* value—the year, quarter, day, hour, and so on—as either an integer or an ASCII string. Since *smalldatetime* is

accurate only to the minute, when a *smalldatetime* value is used with either of these functions, seconds and milliseconds are always zero.

The following examples assume the July 29 date shown in the preceding example.

```
select datepart(month, getdate())
--------------
            7

(1 row affected)
select datename(month, getdate())
-------------
July

(1 row affected)
```

Calculate Intervals or Increment Dates

The **datediff** function calculates the amount of time in date parts between the second and first of two dates you specify—in other words, it finds an interval between two dates. The result is a signed integer value equal to *date2* - *date1*, in date parts.

This query uses the date November 30, 1985 and finds the number of days that elapsed between *pubdate* and that date:

```
select newdate = datediff(day, pubdate,
    "Nov 30 1985")
from titles
```

For the rows in titles having a *pubdate* of October 21, 1985, the result produced by the previous query is 40, the number of days between October 21 and November 30. To calculate an interval in months, the query is:

```
select interval = datediff(month, pubdate,
    "Nov 30 1985")
from titles
```

It produces a value of 1 for the rows with a *pubdate* in October, and a value of 5 for the rows with a *pubdate* in June. When the first date in the **datediff** function is later than the second date specified, the resulting value is negative. Since two of the rows in titles have values for *pubdate* that are assigned using the **getdate** function as a default, these values are set to the date that your *pubs* database was created, and return negative values in the two preceding queries.

If one or both of the date arguments is a *smalldatetime* value, they are converted to *datetime* values internally for the calculation. Seconds and milliseconds in *smalldatetime* values are automatically set to 0 for the purpose of the difference calculation.

Add Date Interval: *dateadd*

The **dateadd** function adds an interval to a date you specify. For example, if the publication dates of all the books in the *titles* table slipped three days, you could get the new publication dates with this statement:

```
select dateadd(day, 3, pubdate)
from titles

------------------
Jun 15 1985 12:00AM
Jun 12 1985 12:00AM
Jul  3 1985 12:00AM
Jun 25 1985 12:00AM
Jun 12 1985 12:00AM
Jun 21 1985 12:00AM
Sep 11 1986 11:02AM
Jul  3 1985 12:00AM
Jun 15 1985 12:00AM
Sep 11 1986 11:02AM
Oct 24 1985 12:00AM
Jun 18 1985 12:00AM
Oct  8 1985 12:00AM

Jun 15 1985 12:00AM
Jun 15 1985 12:00AM
Oct 24 1985 12:00AM
Jun 15 1985 12:00AM
Jun 15 1985 12:00AM

(18 rows affected)
```

If the date argument is given as a *smalldatetime*, the result is also a *smalldatetime*. You can use **dateadd** to add seconds or milliseconds to a *smalldatetime*, but it is meaningful only if the result date returned by **dateadd** changes by at least one minute.

Datatype Conversion Functions

Datatype conversions change an expression from one datatype to another and reformat date and time information. SQL Server performs certain datatype conversions automatically. These are called **implicit conversions**. For example, if you compare a *char* expression and a *datetime* expression, or a *smallint* expression and an *int* expression, or *char* expressions of different lengths, SQL Server automatically converts one datatype to another.

You must request other datatype conversions explicitly, using one of the built-in datatype conversion functions. For example, before concatenating numeric expressions, you must convert them to character expressions.

SQL Server provides three datatype conversion functions, **convert**, **inttohex**, and **hextoint**. You can use these functions in the select list, in the **where** clause, and anywhere else an expression is allowed.

SQL Server does not allow you to convert certain datatypes to certain other datatypes, either implicitly or explicitly. For example, you cannot convert *smallint* data to *datetime*, or *datetime* data to *smallint*. Unsupported conversions result in error messages.

Supported Conversions

Figure 10-1: Implicit, explicit, and unsupported datatype conversions summarizes the datatype conversions supported by SQL Server:

- Conversions marked “I” are handled implicitly. They do not require a datatype conversion function, although you can use the **convert** function on them without error.
- Conversions marked “E” must be done explicitly, with the appropriate datatype conversion function.
- Conversions marked “IE” are handled implicitly when there is no loss of precision or scale and the **arithabort numeric_truncation** option is on, but require an explicit conversion otherwise.
- Conversions marked “U” are unsupported. If you attempt such a conversion, SQL Server generates an error message.
- Conversions of a type to itself are marked “-” . In general, SQL Server does not prohibit you from explicitly converting a type to itself, but such conversions are meaningless.

From: To:	binary	varbinary	tinyint	smallint	int	decimal	numeric	real	float	char, nchar	varchar, nvarchar	smallmoney	money	bit	smalldatetime	datetime	text	image
binary	-	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	U	I
varbinary	I	-	I	I	I	I	I	I	I	I	I	I	I	I	I	I	U	I
tinyint	I	I	-	I	I	I	I	I	I	E	E	I	I	I	U	U	U	U
smallint	I	I	I	-	I	I	I	I	I	E	E	I	I	I	U	U	U	U
int	I	I	I	I	-	I	I	I	I	E	E	I	I	I	U	U	U	U
decimal	I	I	I	I	I	IE	IE	I	I	E	E	I	I	I	U	U	U	U
numeric	I	I	I	I	I	IE	IE	I	I	E	E	I	I	I	U	U	U	U
real	I	I	I	I	I	I	I	-	I	E	E	I	I	I	U	U	U	U
float	I	I	I	I	I	I	I	I	-	E	E	I	I	I	U	U	U	U
char, nchar	E	E	E	E	E	E	E	E	E	I	I	E	E	E	I	I	I	I
varchar, nvarchar	E	E	E	E	E	E	E	E	E	I	I	E	E	E	I	I	I	I
smallmoney	I	I	I	I	I	I	I	I	I	I	I	-	I	I	U	U	U	U
money	I	I	I	I	I	I	I	I	I	I	I	I	-	I	U	U	U	U
bit	I	I	I	I	I	I	I	I	I	I	I	I	I	-	U	U	U	U
smalldatetime	I	I	U	U	U	U	U	U	U	E	E	U	U	U	-	I	U	U
datetime	I	I	U	U	U	U	U	U	U	E	E	U	U	U	I	-	U	U
text	U	U	U	U	U	U	U	U	U	E	E	U	U	U	U	U	U	U
image	I	I	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U

Figure 10-1: Implicit, explicit, and unsupported datatype conversions

Using the General-Purpose Conversion Function: *convert*

The general conversion function, **convert**, is used to convert between a wide variety of datatypes and to specify a new display format for date and time information. Its syntax is:

convert(*datatype, expression* [, *style*])

Here is an example that uses **convert** in the select list:

```
select title, convert(char(5), total_sales)
from titles
where type = "trad_cook"

title
-----------------------------------     -----
Onions, Leeks, and Garlic: Cooking
    Secrets of the Mediterranean         125
Fifty Years in Buckingham Palace
    Kitchens                            15096
Sushi, Anyone?                          5405

(3 rows affected)
```

In this example, the *total_sales* column, an *int* column, is converted to a *char(5)* column so that it can be used with the **like** keyword:

```
select title, total_sales
from titles
where convert(char(5), total_sales) like "15%"
    and type = "trad_cook"

title
--------------------------------        -----
Fifty Years in Buckingham Palace
    Kitchens                            15096

(1 row affected)
```

Certain datatypes expect either a length or a precision and scale. If you do not specify a length, SQL Server uses the default length of 30 for character and binary data. If you do not specify a precision or scale, SQL Server uses the defaults of 18 and 0, respectively.

Conversion Rules

The following sections describe the rules SQL Server observes when converting different types of information:

Converting Character Data to a Noncharacter Type

Character data can be converted to a noncharacter type—such as a money, date and time, exact numeric, or approximate numeric type—if it consists entirely of characters that are valid for the new type. Leading blanks are ignored.

Syntax errors are generated when the data includes unacceptable characters. Following are some examples of characters that cause syntax errors:

- Commas or decimal points in integer data
- Commas in monetary data
- Letters in exact or approximate numeric data or bit-stream data
- Misspelled month names in date and time data

Converting from One Character Type to Another

When converting from a multibyte character set to a single-byte character set, characters with no single-byte equivalent are converted to blanks.

text columns can be explicitly converted to *char, nchar, varchar,* or *nvarchar.* You are limited to the maximum length of the character datatypes, 255 bytes. If you do not specify the length, the converted value has a default length of 30 bytes.

Converting Numbers to a Character Type

Exact and approximate numeric data can be converted to a character type. If the new type is too short to accommodate the entire string, an insufficient space error is generated. For example, the following conversion tries to store a 5-character string in a 1-character type:

```
select convert(char(1), 12.34)

Insufficient result space for explicit conversion
of NUMERIC value '12.34' to a CHAR field.
```

Rounding During Conversion to or from Money Types

The *money* and *smallmoney* types store four digits to the right of the decimal point, but round up to the nearest hundredth (.01) for display purposes. When data is converted to a money type, it is rounded up to four places.

Data converted from a money type follows the same rounding behavior if possible. If the new type is an exact numeric with less than three decimal places, the data is rounded to the scale of the new type. For example, when $4.50 is converted to an integer, it yields 4:

```
select convert(int, $4.50)
```

```
-----------
          4
```

Data converted to *money* or *smallmoney* is assumed to be in full currency units, such as dollars, rather than in fractional units, such as cents. For example, the integer value of 4 would be converted to the money equivalent of 4 dollars, not 4 cents, in us_english.

Converting Date and Time Information

Data that is recognizable as a date can be converted to *datetime* or *smalldatetime*. Incorrect month names lead to syntax errors. Dates that fall outside the acceptable range for the datatype lead to arithmetic overflow errors.

When *datetime* values are converted to *smalldatetime*, they are rounded up to the nearest minute.

Converting Between Numeric Types

Data can be converted from one numeric type to another. If the new type is an exact numeric whose precision or scale is not sufficient to hold the data, errors can occur. Use the **arithabort** and **arithignore** options to determine how these errors are handled.

➤ ***Note***

The **arithabort** and **arithignore** options have been redefined as of SQL Server release 10.0. If you use these options in your applications, examine them to make sure they are still functioning correctly.

Converting Binary-Like Data

SQL Server *binary* and *varbinary* data is platform-specific; the type of hardware you are using determines how the data is stored and interpreted. Some platforms consider the first byte after the 0x prefix to be the most significant; others consider the first byte to be the least significant.

The **convert** function treats Sybase binary data as though it were a string of characters, rather than numeric information. **convert** takes no account of byte order significance when converting a binary expression to an integer or an integer expression to a binary value.

Because of this, conversion results can vary from one platform to another.

Before converting a binary string to an integer, **convert** strips it of its 0x prefix. If the string consists of an odd number of digits, SQL Server inserts a leading zero. If the data is too long for the integer type, **convert** truncates it. If the data is too short, **convert** right-justifies and zero-pads it.

Suppose you want to convert the string 0x00000100 to an integer. On some platforms, this string represents the number 1; on others, the number 256. Depending on which platform executes the function, **convert** returns either 1 or 256 on others.

Converting Hexadecimal Data

For conversion results that are reliable across platforms, use the **hextoint** and **inttohex** functions.

hextoint accepts literals or variables consisting of digits and the uppercase and lowercase letters A through F, with or without a 0x prefix. The following are all valid uses of **hextoint**:

```
hextoint("0x00000100FFFFF")
hextoint("0x00000100")
hextoint("100")
```

hextoint strips it of the 0x prefix. If the data exceeds eight digits, **hextoint** truncates it. If the data is less than eight digits, **hextoint** right-justifies and zero-pads it. Then **hextoint** returns the platform-independent integer equivalent. The above expressions all return the same value, 256, regardless of the platform that executes the **hextoint** function.

The **inttohex** function accepts integer data and returns an 8-character **hexadecimal string** without a 0x prefix. **inttohex** always returns the same results, regardless of which platform you are using.

Converting *image* Data to *binary* or *varbinary*

You can use the **convert** function to convert an *image* column to *binary* or *varbinary*. You are limited to the maximum length of the *binary* datatypes, 255 bytes. If you do not specify the length, the converted value has a default length of 30 characters.

Conversion Errors

The following sections describe the types of errors that can occur during datatype conversions.

Arithmetic Overflow and Divide-by-Zero Errors

Divide-by-zero errors occur when SQL Server tries to divide a numeric value by zero. Arithmetic overflow errors occur when the new type has too few decimal places to accommodate the results. This happens during

- Explicit or implicit conversions to exact types with a lower precision or scale
- Explicit or implicit conversions of data that falls outside the acceptable range for a money or datetime type
- Conversions of strings longer than 4 bytes using **hextoint**

Both arithmetic overflow and divide-by-zero errors are considered serious, whether they occur during implicit or explicit conversions. Use the **arithabort arith_overflow** option to determine how SQL Server handles these errors. The default setting, **arithabort arith_overflow on**, rolls back the entire transaction or batch in which the error occurs. If you set **arithabort arith_overflow off**, SQL Server aborts the statement that causes the error but continues to process other statements in the transaction or batch.You can use the *@@error* global variable to check statement results.

Use the **arithignore arith_overflow** option to determine whether SQL Server displays a message after these errors. The default setting, **off**, displays a warning message when a divide-by-zero error or a loss of precision occurs. Setting **arithignore arith_overflow on** suppresses warning messages after these errors. The optional **arith_overflow** keyword can be omitted without any effect.

Scale Errors

When an explicit conversion results in a loss of scale, the results are truncated without warning. For example, when you explicitly convert a *float*, *numeric*, or *decimal* type to an *integer*, SQL Server assumes you really want the result to be an integer and truncates all numbers to the right of the decimal point.

During implicit conversions to *numeric* or *decimal* types, loss of scale generates a scale error. Use the **arithabort numeric_truncation** option to

determine how serious such an error is considered. The default setting, **arithabort numeric_truncation on**, aborts the statement that causes the error but continues to process other statements in the transaction or batch. If you set **arithabort numeric_truncation off**, SQL Server truncates the query results and continues processing.

Domain Errors

The **convert** function generates a domain error when the function's argument falls outside the range over which the function is defined. This should happen very rarely.

Conversions Between Binary and Integer Types

The *binary* and *varbinary* types store hexadecimal-like data consisting of a 0x prefix followed by a string of digits and letters. These strings are interpreted differently by different platforms. For example, the string 0x0000100 represents 65536 on machines that consider byte 0 most significant, and 256 on machines that consider byte 0 least significant.

The convert Function and Implicit Conversions

The binary types can be converted to integer types either explicitly, using the **convert** function, or implicitly. The data is stripped of its 0x prefix then zero-padded if it is too short for the new type or truncated if it is too long.

Both **convert** and the implicit datatype conversions evaluate binary data differently on different platforms. Because of this, results may vary from one platform to another. Use the **hextoint** function for platform-independent conversion of hexadecimal strings to integers, and the **inttohex** function for platform-independent conversion of integers to hexadecimal values.

The hextoint Function

Use the **hextoint** function for platform-independent conversions of hexadecimal data to integers. **hextoint** accepts a valid hexadecimal string, with or without a 0x prefix, enclosed in quotes, or the name of a character type column or variable.

hextoint returns the integer equivalent of the hexadecimal string. The function always returns the same integer equivalent for a given hexadecimal string, regardless of the platform on which it is executed.

The inttohex Function

Use the **inttohex** function for platform-independent conversions of integers to hexadecimal strings. **inttohex** accepts any expression that evaluates to an integer. It always returns the same hexadecimal equivalent for a given expression, regardless of the platform on which it is executed.

Converting image Columns to Binary Types

You can use the **convert** function to convert an *image* column to *binary* or *varbinary*. You are limited to the maximum length of the *binary* datatypes, 255 bytes. If you do not specify the length, the converted value has a default length of 30 characters.

Converting Other Types to bit

Exact and approximate numeric types can be converted to the bit type implicitly. Character types require an explicit **convert** function.

The expression being converted must consist only of digits, a decimal point, a currency symbol, and a plus or minus sign. The presence of other characters generates syntax errors.

The *bit* equivalent of 0 is 0. The *bit* equivalent of any other number is 1.

Changing the Display Format for Dates

The *style* parameter of **convert** provides a wide variety of date display formats when converting *datetime* or *smalldatetime* data to *char* or *varchar*. The number argument you supply as the *style* parameter determines how the data is displayed. The year can be displayed in either two digits or four digits. Add 100 to a *style* value to get a 4-digit year, including the century (*yyyy*).

Following is a table of the possible values for *style* and the variety of date formats you can use. When you use *style* with *smalldatetime*,

those styles that include seconds or milliseconds will show zeros in those positions.

Table 10-11: Converting date formats with the style parameter

Without Century (yy)	With Century (yyyy)	Standard	Output
-	0 or 100	Default	*mon dd yyyy hh:mm* AM (or PM)
1	101	USA	*mm/dd/yy*
2	2	SQL standard	*yy.mm.dd*
3	103	English / French	*dd/mm/yy*
4	104	German	*dd.mm.yy*
5	105		*dd-mm-yy*
6	106		*dd mon yy*
7	107		*mon dd, yy*
8	108		*hh:mm:ss*
-	9 or 109	Default + milliseconds	*mon dd yyyy hh:mm:sss* AM (or PM)
10	110	USA	*mm-dd-yy*
11	111	Japan	*yy/mm/dd*
12	112	ISO	*yymmdd*

The default values, style 0 or 100, and 9 or 109, always return the century (*yyyy*).

Following is an example of the use of **convert**'s *style* parameter:

```
select convert(char(12), getdate(), 3)
```

This converts the current date to style "3", *dd/mm/yy*.

11 Creating Indexes on Tables

You can create one or more indexes on a table to speed up the data retrieval process. Indexes are transparent to users accessing data from that table; SQL Server automatically decides when to use the indexes created for tables.

This chapter discusses:

- A general overview of indexes and some guidelines on when to use them
- How to create indexes for a table
- When to use clustered or nonclustered indexes
- How to specify index options
- How to drop indexes
- How to determine what indexes exist on a table

What Are Indexes?

Indexes help SQL Server locate data. They speed up data retrieval by pointing SQL Server to the location of a table column's data on disk. Tables can have more than one index.

Indexes are transparent to users. SQL includes no syntax for referring to an index in a query. You can only create or drop indexes from a table; SQL Server decides whether to use the indexes for each query submitted for that table. As the data in a table changes over time, SQL Server may change the table's indexes to reflect those modifications. Again, these changes are transparent to users, SQL Server handles this task on its own.

SQL Server supports the following types of indexes:

- **Composite indexes** – These indexes involve more than one column. Use this type of index when two or more columns are best searched as a unit because of their logical relationship.
- **Unique indexes** – These indexes do not permit any two rows in the specified columns to have the same value. SQL Server checks for duplicate values when the index is created (if data already exists) and each time data is added.
- **Clustered** or **nonclustered indexes** – Clustered indexes force SQL Server to continually sort and re-sort the rows of a table so

that their physical order is always the same as their logical (or indexed) order. You can have only one clustered index per table. Nonclustered indexes do not require the physical order of rows to be the same as their indexed order. Each nonclustered index can provide access to the data in a different sort order.

These types of indexes are described in more detail later in this chapter.

Comparing the Two Ways to Create Indexes

You can create indexes on tables either by using the **create index** statement (described in this chapter) or by using the **unique** or **primary key** integrity constraints of the **create table** command. However, these integrity constraints are limited in the following ways:

- You cannot create nonunique indexes.
- You cannot use the options provided by the **create index** command to tailor how indexes work.
- You can only drop these indexes as a constraint using the **alter table** statement.

If your application requires these features, you should create your indexes using **create index**. Otherwise, the **unique** or **primary key** integrity constraints offer a simpler way to define an index for a table. For information about the **unique** and **primary key** constraints, see Chapter 7, "Creating Databases and Tables."

Guidelines for Using Indexes

Indexes speed the retrieval of data. Putting an index on a column often makes the difference between a nearly immediate response to a query and a long wait.

So why not index every column? The most significant reason is that building an index takes time and storage space.

For example, note that nonclustered indexes are automatically re-created when a clustered index is rebuilt.

A second reason is that inserting, deleting, or updating data in indexed columns takes a little longer than in unindexed columns. But this cost is usually outweighed by the extent to which indexes improve retrieval performance.

Here are some general guidelines on when to index:

- If you plan to do manual insertions into the IDENTITY column, create a unique index to ensure that the inserts do not assign a value that has already been used.
- A column that is often accessed in sorted order, that is, specified in the **order by** clause, probably should be indexed so that SQL Server can take advantage of the indexed order.
- Columns that are regularly used in joins should always be indexed, since the system can perform the join faster if the columns are in sorted order.
- The column that stores the primary key of the table often has a clustered index, especially if it is frequently joined to columns in other tables. (Remember, there is only one clustered index per table.)
- A column that is often searched for ranges of values might be a good choice for a clustered index. Once the row with the first value in the range is found, rows with subsequent values are guaranteed to be physically adjacent. A clustered index does not offer as much of an advantage for searches on single values.

There are some cases where indexes are not useful:

- Columns that are seldom or never referenced in queries don't benefit from indexes, since the system seldom or never has to search for rows on the basis of values in these columns.
- Columns that can have only two or three values, for example, male and female or yes and no, get no real advantage from indexes.

If the system does have to search an unindexed column, it does so by looking at the rows one by one. The length of time it takes to perform this kind of scan is directly proportional to the number of rows in the table.

Creating Indexes to Speed Up Data Retrieval

Indexes are created on columns in order to speed retrieval of data. The simplest form of the **create index** command is:

```
create index index_name
   on table_name (column_name)
```

To create an index on the *au_id* column of the *authors* table, the command is:

```
create index au_id_ind
on authors(au_id)
```

The index name must conform to the rules for identifiers. The column and table name specify the column you want indexed and the table that contains it.

You cannot create indexes on columns with *bit*, *text*, or *image* datatypes.

You must be the owner of a table in order to **create** or **drop** an index. The owner of a table can **create** or **drop** an index at any time, whether or not there is data in the table. Indexes can be created on tables in another database by qualifying the table name.

create index Syntax

The complete syntax of the **create index** command is:

```
create [unique] [clustered | nonclustered]
       index index_name
   on [[database.]owner.]table_name (column_name
       [, column_name]...)
   [with {{fillfactor | max_rows_per_page}= x,
       ignore_dup_key, sorted_data,
       [ignore_dup_row | allow_dup_row]}]
   [on segment_name]
```

The following subsections explain the various options to the **create index** command.

➤ ***Note***

The **on *segment_name*** extension to **create index** allows you to place your index on a segment, which points to a specific database device or a collection of database devices. Before creating an index on a segment, see a System Administrator or the Database Owner for a list of segments that you can use. Certain segments may be allocated to specific tables or indexes for performance reasons or for other considerations.

Indexing More Than One Column: Composite Indexes

You have to specify two or more column names if you want to create a composite index on the combined values in all the specified columns.

Composite indexes are used when two or more columns are best searched as a unit. For example, the *friends_etc* table has a composite index on *pname* and *sname*. List all the columns to be included in the composite index in sort-priority order inside the parentheses after the table name, like this:

```
create index nmind
on friends_etc(pname, sname)
```

The columns in a composite index don't have to be in the same order as the columns in the **create table** statement. The order of *pname* and *sname* could be reversed in the preceding index creation statement.

Up to 16 columns can be combined into a single composite index. All the columns in a composite index must be in the same table. The maximum allowed size of the combined index values is 256 bytes. That is, the sum of the lengths of the columns that make up the composite index cannot exceed 256.

You can specify two or more column names when you create an index. These columns, together with the *sensitivity* column, form a composite index on the combined values of the columns. Composite indexes are used when two or more columns are best searched as a unit. For example, the *friends_etc* table has a composite index on *pname*, *sname*, and *sensitivity* (added automatically by SQL Server). When you specify the columns on the **create index** statement, list all the columns to be included in the composite index, except *sensitivity*, in sort-priority order inside the parentheses after the table name, like this:

```
create index nmind
on friends_etc(pname, sname)
```

The columns in a composite index do not have to be in the same order as the columns in the **create table** statement. The order of *pname* and *sname* could be reversed in the preceding index creation statement. SQL Server always adds *sensitivity* as the last column of every index.

Using the *unique* Option

A unique index is one in which no two rows are permitted to have the same index value, including NULL. The system checks for duplicate values when the index is created, if data already exists, and checks each time data is added or modified with an **insert** or **update**.

Specifying a unique index makes sense only when uniqueness is a characteristic of the data itself. For example, you would not want a unique index on a *last_name* column since there is likely to be more than one "Smith" or "Wong" in tables of even a few hundred rows.

On the other hand, a unique index on a column holding social security numbers is a good idea. Uniqueness is a characteristic of the data—each person has a different social security number. Furthermore, a unique index serves as an integrity check. For instance, a duplicate social security number probably reflects some kind of error in data entry or on the part of the government.

If you try to create a unique index on existing data that includes duplicate values, the command is aborted and SQL Server displays an error message that gives the first duplicate. You cannot create a unique index on a column that contains null values in more than one row; these are treated as duplicate values for indexing purposes.

If you try to change data on which there is a unique index, the results depend on whether you have used the **ignore_dup_key** option. See the discussion of index options later in this chapter.

You can use the **unique** keyword on composite indexes. This was not done for the *friends_etc* index we just created.

Including IDENTITY Columns in Nonunique Indexes

The **identity in nonunique index** option automatically includes an IDENTITY column in a table's index keys so that all indexes created on the table are unique. This database option makes logically nonunique indexes internally unique and allows them to be used to process updatable cursors and isolation level 0 reads.

The table must already have an IDENTITY column for the **identity in nonunique index** database option to work, either from a **create table** statement or by setting the **auto identity** database option to **true** before creating the table.

Use **identity in nonunique index** if you plan to use cursors and isolation level 0 reads on tables with nonunique indexes. A unique index

ensures that the cursor will be positioned at the correct row the next time a **fetch** is performed on that cursor.

Using the *fillfactor* and *max_rows_per_page* Options

It is seldom necessary to include the **fillfactor** or **max_rows_per_page** options in your **create index** statement. These options are provided for fine-tuning performance. They are useful only when you are creating a new index on existing data.

fillfactor

With the **fillfactor** option, the user can specify how full SQL Server should make each index page. The amount of empty space on an index page is a matter of concern because when an index page fills up after enough rows are added, the system must take the time to split it in order to make room for new rows.

The default is 0, and this value is used when you don't specify a fillfactor. A System Administrator can change the default with the system procedure **sp_configure**. See the *System Administration Guide* for more information about **fillfactor.**

Legal user-specified **fillfactor** values are between 1 and 100.

Here is a **create index** statement that uses the **fillfactor** option:

```
create index postalcode_ind
on friends_etc(postalcode)
with fillfactor = 100
```

A **fillfactor** of 100 fills every page completely, and makes sense only when you know that no index values in the table will ever change.

max_rows_per_page

The **max_rows_per_page** option limits the number of rows SQL Server can put on each index page. A low **max_rows_per_page** value reduces lock contention and makes sense only for frequently accessed tables. Low values also cause the index to take up space.

The default is 0, and this value is used when you do not specify a maximum. A user can change the value with the system procedure **sp_relimit.**

User-specified **max_rows_per_page** values are between 1 and 256.

The following **create index** statement uses the **max_rows_per_page** option:

```
create index postalcode_ind
on friends_etc(postalcode)
with max_rows_per_page = 10
```

Using Clustered or Nonclustered Indexes

With a clustered index, SQL Server sorts rows on an ongoing basis so that their physical order is the same as their logical, that is, indexed, order. The bottom or **leaf level** of a clustered index contains the actual data pages of the table. Create the clustered index before creating any nonclustered indexes, since nonclustered indexes are automatically rebuilt when a clustered index is created.

By definition, there can be only one clustered index per table. It is often created on the **primary key**—the column or columns that uniquely identify the row.

Logically, a primary key is determined by the design of the database. However, you can explicitly define primary keys, foreign keys, and common keys (pairs of keys that are frequently joined) with the system procedures **sp_primarykey, sp_foreignkey,** and **sp_commonkey.** You can display information about keys with **sp_helpkey** and about columns that are likely join candidates with **sp_helpjoins.**

Alternatively, you can specify **primary key** constraints with the **create table** or **alter table** statements to create an index and enforce the primary key attributes for table columns. You can display information about constraints with **sp_helpconstraint.**

For a definition of primary and foreign keys, see Chapter 15, "Triggers: Enforcing Referential Integrity." For complete information on system procedures, see the *SQL Server Reference Manual*.

With a nonclustered index, the physical order of the rows is not the same as their indexed order. The leaf level of a nonclustered index contains pointers to rows on data pages. More precisely, each leaf page contains an indexed value and a pointer to the row with that value. In other words, a nonclustered index has an extra level between the index structure and the data itself.

Each of the up to 249 nonclustered indexes permitted on a table can provide access to the data in a different sorted order.

Finding data using a clustered index is almost always faster than using a nonclustered index. In addition, a clustered index is advantageous when many rows with contiguous key values are being retrieved—that is, on columns that are often searched for ranges of values. Once the row with the first **key value** is found, rows

with subsequent indexed values are guaranteed to be physically adjacent, and no further searches are necessary.

If neither the **clustered** nor the **nonclustered** keyword is used, a nonclustered index is created.

Here is how the index on the *title_id* column of the *titles* table is created (if you want to try this command, you must drop the index first with **drop index**):

```
create clustered index titleidind
on titles(title_id)
```

Since you think you will often want to sort the people in *friends_etc* by postal code, you should create a nonclustered index on the *postalcode* column like this:

```
create nonclustered index postalcodeind
on friends_etc(postalcode)
```

A unique index would not make sense here, since some of your contacts are likely to have the same postal code. A clustered index would not be appropriate either, since the postal code is not the primary key.

The clustered index in *friends_etc* should be a composite index on the personal name and surname columns. In order to create this clustered index, you must drop the original nonclustered *nmind* index first:

```
drop index friends_etc.nmind
```

Then create the clustered index:

```
create clustered index nmind
on friends_etc(pname, sname)
```

➤ ***Note***

Since the leaf level of a clustered index and its data pages are the same by definition, creating a **clustered** index and using the **on *segment_name*** extension effectively moves a table from the device on which it was created to the named segment.

See a System Administrator or the Database Owner before creating tables or indexes on segments; certain segments may be reserved for performance reasons.

Specifying Index Options

The index options **ignore_dup_key, ignore_dup_row,** and **allow_dup_row** control what happens when a duplicate key or duplicate row is created with **insert** or **update.** Here is a chart of when these index options can be used:

Table 11-1: Index options

Index Type	Options
Clustered	**ignore_dup_row** | **allow_dup_row**
Unique clustered	**ignore_dup_key**
Nonclustered	None
Unique nonclustered	**ignore_dup_key**
Unique nonclustered	**ignore_dup_row**

Using the *ignore_dup_key* Option

If you try to insert a duplicate value into a column that has a unique index, the command is cancelled. You can avoid having a large transaction cancelled by including the **ignore_dup_key** option with a **unique** index.

The **unique** index can be either clustered or nonclustered. When you begin data entry, each attempt to insert a duplicate key is cancelled, with an error message. Nonduplicate keys are inserted normally.

➤ ***Note***

If an attempted **update** would create a duplicate key, the update is cancelled. After the cancellation, any transaction that was active at the time may continue as though the **update** had never taken place.

You cannot create a unique index on a column that already includes duplicate values, whether or not **ignore_dup_key** is set. If you attempt to do so, SQL Server prints an error message and a list of the duplicate values. You must eliminate duplicates before you create a unique index on the column.

Here is an example of using the **ignore_dup_key** option:

```
create unique clustered index phone_ind
on friends_etc(phone)
with ignore_dup_key
```

Using the *ignore_dup_row* and *allow_dup_row* Options

ignore_dup_row and **allow_dup_row** are options for creating a nonunique, clustered index. These options are not relevant when creating a nonunique, nonclustered index. Since a SQL Server nonclustered index attaches a unique row identification number internally, it never worries about duplicate rows—even for identical data values.

ignore_dup_row and **allow_dup_row** are mutually exclusive.

If **allow_dup_row** is set, you can create a new nonunique, clustered index on a table that includes duplicate rows, and you can subsequently create duplicate rows with **insert** or **update**.

If any index in the table is unique, the requirement for uniqueness—the most stringent requirement—takes precedence over the **allow_dup_row** option. Thus, **allow_dup_row** applies only to tables with nonunique indexes. You cannot use this keyword if a unique clustered index exists on any column in the table.

The **ignore_dup_row** option is used to eliminate duplicates from a batch of data. When you enter a duplicate row, that row is ignored and that particular **insert** is canceled, with an informational error message. Non-duplicate rows are inserted normally.

The **ignore_dup_row** applies only to tables with nonunique indexes: you cannot use this keyword if a unique index exists on any column in the table.

➤ ***Note***

An attempted **update** that creates a duplicate row causes that **update** to be cancelled. After the cancellation, any transaction that may have been active at the time may continue as though the update had never taken place.

This table illustrates how **allow_dup_row** and **ignore_dup_row** affect attempts to create a nonunique clustered index on a table that includes duplicate rows, and to enter duplicate rows into a table.

Table 11-2: Duplicate row options in indexes

Option	Has Duplicates	Enter Duplicates
Neither option set	**create index** command fails.	Enter duplicate rows command fails.
allow_dup_row set	Command completes.	Command completes.
ignore_dup_row set	Index created but duplicate rows thrown out; error message.	All rows accepted except duplicates; error message. See earlier warning.

Using the *sorted_data* Option

The **sorted_data** option speeds creation of an index when the data in the table is already in sorted order, for example, when you have used **bcp** to copy data that has already been sorted into an empty table. The speed increase becomes significant on large tables and increases to several times faster in tables larger than a gigabyte. This option can be used in conjunction with any other **create index** options with no effect on their operation.

If **sorted_data** is specified but data is not in sorted order, an error message displays and the command is aborted.

This option speeds indexing only for clustered indexes or unique nonclustered indexes. Creating a nonunique nonclustered index will, however, be successful unless there are rows with duplicate keys. If there are rows with duplicate keys, an error message displays and the command is aborted.

Using the *on segment_name* Option

The **on *segment_name*** clause allows you to specify a database segment name on which the index is to be created. A nonclustered index can be created on a different segment than the data pages. For example:

```
create index titleind
on titles(title)
on seg1
```

Dropping Indexes

The **drop index** command is used to remove an index from the database. Its syntax is:

```
drop index table_name.index_name
   [, table_name.index_name]...
```

When you issue this command SQL Server removes the specified indexes from the database, reclaiming their storage space.

Only the owner of the index can drop it. **drop index** permission cannot be transferred to other users. The **drop index** command cannot be used on any of the system tables in the *master* database or in the user database.

You might want to drop an index if it is not used for most or all of your queries.

To drop the index *phone_ind* in the *friends_etc* table, the command is:

```
drop index friends_etc.phone_ind
```

Determining What Indexes Exist on a Table

To see the indexes that exist on a table, you can use the system procedure **sp_helpindex**. Here is a report on the *friends_etc* table:

```
sp_helpindex friends_etc
index_name     index_description               index_keys
-------------- --------------------------      -------------
nmind          clustered located on default    pname, sname

postalcode_ind nonclustered located on default  postalcode

postalcodeind  nonclustered located on default  postalcode

(3 rows affected, return status = 0)
```

sp_help also provides information about the indexes on a table.

Updating Statistics About Indexes

The **update statistics** command helps SQL Server make the best decisions about which indexes to use when it processes a query, by keeping it up to date about the distribution of the key values in the indexes. It should be used when a large amount of data in an indexed column has been added, changed, or deleted.

Permission to issue the **update statistics** command defaults to the table owner, and is not transferable. Its syntax is:

update statistics *table_name* [*index_name*]

If you do not specify an index name, the command updates the distribution statistics for all the indexes in the specified table. Giving an index name updates statistics for that index only.

You can find the names of indexes by using the **sp_helpindex** system procedure. This procedure takes a table name as a parameter.

Here is how to list the indexes for the *authors* table:

```
sp_helpindex authors

index_name  index_description     index_keys
----------  -----------------     ------------------
auidind     clustered, unique     au_id
aunmind     nonclustered          au_lname, au_fname

(2 rows affected)
```

To update the statistics for all of the indexes, type:

```
update statistics authors
```

To update the statistics only for the index on the *au_id* column, type:

```
update statistics authors auidind
```

Since Transact-SQL does not require index names to be unique in a database, you must give the name of the table with which the index is associated. SQL Server runs **update statistics** automatically when you create an index on existing data.

12 Defining Defaults and Rules for Data

You can define a default value for a table column or user-defined datatype to automatically insert a value if a user does not explicitly enter a value for it. You can also define rules for that table column or datatype to restrict the types of values users can enter for it.

This chapter discusses:

- A general overview of defaults and rules
- How to create and drop defaults
- How defaults can affect null values
- How to create and drop rules
- How to get information about defaults and rules

What Are Defaults and Rules?

A **default** is a value that SQL Server inserts into a column if the user does not explicitly enter one. In the world of database management, a **rule** specifies what you are or are not allowed to enter in a particular column or in any column with a given user-defined datatype. You can use defaults and rules to help maintain the integrity of data across the database.

In a relational database management system, every data element, that is, a particular column in a particular row, must contain some value, even if that value is NULL. As discussed in Chapter 7, "Creating Databases and Tables," some columns do not accept the null value. For those columns, some other value must be entered, either a value explicitly entered by the user or a default entered by SQL Server.

Defaults allow you to specify a value that SQL Server inserts if no explicit value is entered in either a NULL or NOT NULL column. For example, you can create a default that has the value "???" or the value "fill in later."

Rules enforce the integrity of data in ways not covered solely by a column's datatype. They can be connected to a specific column, to several specific columns, or to a specified user-defined datatype.

Every time a user enters a value with an **insert** or **update** statement, SQL Server checks it against the most recent rule that has been bound

to the specified column. Data entered prior to the creation and binding of a rule is not checked.

➤ ***Note***

You can bind a character type rule to a numeric type column even though it makes no sense to do so. Rules are checked when an **insert** or **update** is attempted, not at the time of binding.

This chapter explains how to create defaults and rules using **create default** and **create rule**, and how to associate defaults and rules with a column or with a user-defined datatype, using the system procedures **sp_bindefault**, **sp_bindrule**, **sp_unbindefault**, and **sp_unbindrule**.

Comparing Defaults and Rules with Integrity Constraints

As an alternative to using defaults and rules, you can use the **default** clause and the **check** integrity constraint of the **create table** statement to accomplish some of the same tasks. However, they are specific for that table and cannot be bound to columns of other tables or to user-defined datatypes. For more information about integrity constraints, see Chapter 7, "Creating Databases and Tables."

Creating Defaults

You can create or drop defaults at any time, before or after data has been entered in a table. Create a default with the **create default** command, and drop it with **drop default.**

A default can connect to a particular column, to a number of columns, or to all the columns in the database having a given user-defined datatype. Use the system procedure **sp_bindefault** to associate a default with a column or user-defined datatype. Remove the association with **sp_unbindefault.**

Here are some things you should check when you are creating and binding defaults:

- Make sure the column is large enough for the default. A *char(2)* column will not hold a 17-byte string like "Nobody knows yet."
- Be careful when you put different defaults on a user-defined datatype and on an individual column of that type. If you bind the user-defined datatype default first and then the column default, the column default replaces the user-defined datatype

default for the named column only. The user-defined datatype default is bound to all the other columns having that datatype. However, once you bind another default to a column that had a default because of its type, that column ceases to be influenced by defaults bound to its datatype. This issue is discussed in more detail later in this chapter.

- Watch out for conflicts between defaults and rules. Be sure that the default value is allowed by the rule; otherwise, the default can be eliminated by the rule.
- If a rule allows entries between 1 and 100, for example, and the default is set to 0, the rule will force the default entry to be rejected every time and you'll get an error, unless the column accepts NULL, in which case "NULL" will be entered. Either the default or the rule will have to be changed.

create default Syntax

The syntax of the **create default** command is:

```
create default [owner.]default_name
   as constant_expression
```

Default names must follow the rules for identifiers. You can create a default in the current database only.

Within a database, default names must be unique for each user. You cannot create two defaults called *phonedflt*. However, as "guest", you can create a *phonedflt* even if *dbo.phonedflt* already exists, because the owner names make each one distinct.

Here is how a default value of "Oakland" can be created that can be used with the *city* column of *friends_etc* (the table whose creation was discussed in Chapter 7, "Creating Databases and Tables") and possibly with other columns or user datatypes. As you continue to follow this example, you can use any city name that works for the demographic distribution of the people you are planning to enter in your personal table. To create the default:

```
create default citydflt
as "Oakland"
```

After the **as**, you can use any constant. Enclose character and date constants in quotes; money, integer, and floating point constants do not require them. Binary data must be preceded by 0x, and money data should be preceded by a dollar sign ($). The default value must be compatible with the datatype of the column. You cannot use

"none," for example, as a default for a numeric column, but 0 (zero) is appropriate.

If you specify NOT NULL when you create a column and do not associate a default with it, SQL Server will produce an error message whenever anyone fails to make an entry in that column.

Often, default values are created when a table is created. However, in a session in which you want to enter many rows having the same values in one or more columns, it may be convenient to create a default tailored to that session before you begin.

Binding Defaults

After you have created a default, use the system procedure **sp_bindefault** to bind the default to a column or user-defined datatype.

```
create default phonedflt as "UNKNOWN"
```

A default value has been defined. Now you need to bind it to the appropriate column or user-defined datatype with the **sp_bindefault** system procedure.

```
sp_bindefault phonedflt, "authors.phone"
```

The default will take effect only if there is no entry in the *phone* column of the *authors* table. No entry is different from entering a null value.

➤ ***Note***

To get the defaultl, you must issue an **insert** or **update** command with a column list that does not include the column that has the default.

The default applies to new rows only. It does not retroactively change already existing rows. Of course, it only takes effect when no entry is made. If the user supplies any value for the column, including NULL, the default has no effect.

Here is how to bind *citydflt* to the *city* column in *friends_etc*:

```
sp_bindefault citydflt, "friends_etc.city"
```

Notice that the table and column name are enclosed in quotes, because of the embedded punctuation, that is, the period.

➤ ***Note***

You cannot bind a default to a system datatype, because the target would be too broad. Also, you cannot bind a default to a *timestamp* column, because SQL Server generates values for *timestamp* columns. You can bind a default to an IDENTITY column, or to a user-defined datatype with the IDENTITY property, but such defaults are ignored. Each time you insert a row into a table without specifying a value for the IDENTITY column, SQL Server assigns a value one greater than the last value assigned.

If you create a special datatype for all city columns in every table in your database, you can bind *citydflt* to that datatype and rest secure in the conviction that "Oakland" will show up only where city names are appropriate. For example, if the user datatype is called *citytype*, here is how to bind *citydflt* to it:

```
sp_bindefault citydflt, citytype
```

The **futureonly** parameter can be used when binding a default to a user datatype. This parameter prevents existing columns of that user datatype from inheriting the new default. It is never used when binding a default to a column. Here's how you create and bind the new default "Berkeley" to the datatype *citytype* for use by new table columns only. "Oakland" will continue to appear as the default for any existing table columns using *citytype*.

```
create default newcitydflt as "Berkeley"
sp_bindefault newcitydflt, citytype, futureonly
```

If most of the people in your table live in the same zip code area, you can create a default to save data entry time. Here is one that is appropriate for a section of Oakland along with its binding:

```
create default zipdflt as "94609"
sp_bindefault zipdflt, "friends_etc.postalcode"
```

Here is the complete syntax for the **sp_bindefault** system procedure:

```
sp_bindefault defname, objname [, futureonly]
```

The *defname* is the name of the default created with **create default.** The *objname* is the name of the table and column, or of the user-defined datatype, to which the default is to be bound. If the parameter is not of the form *table.column*, it is assumed to be a user-defined datatype.

All columns of a specified user-defined datatype become associated with the specified default unless one of the following is true:

- You use the optional third parameter, **futureonly**, which prevents existing columns of that user datatype from inheriting the default; or
- The column's default has previously been changed, in which case the changed default is maintained.

Defaults bound to columns always take precedence over defaults bound to user datatypes. Binding a default to a column will replace a default bound to the user-defined datatype of that column, but binding a default to a datatype will not replace a rule default to a column of that user-defined datatype. The following table indicates the precedence when binding defaults to columns and user datatypes where defaults already exist:

Table 12-1: Default precedence

New Default Bound To:	Old Default Bound To:	
	User Datatype	**Column**
User Datatype	Replaces old default	No change; old default
Column	Replaces old default	Replaces old default

Existing columns of the user-defined datatype inherit the new default unless their default has previously been changed, or the value of the optional third parameter is **futureonly**. New columns of the user-defined datatype always inherit the default.

For example, you create a table called *foes* with a column named *city* of type *citytype*, a user-defined datatype. Initially, the user-defined datatype *citytype* has no default. After creating a default called *citydflt*, you bind it to the column *foes.city*. Now you bind another default, *newcitydflt*, to the user datatype *citytype*. Although *foes.city* is a *citytype* column, the new default does not bind to it, since its default has previously been changed.

➤ ***Note***

Defaults cannot be bound to columns and used during the same batch. **sp_bindefault** cannot be in the same batch as **insert** statements that invoke the default.

Unbinding Defaults

Unbinding a default means disconnecting it from a particular column or user-defined datatype. An unbound default is still stored in the database and is available for future use. Use the system procedure **sp_unbindefault** to remove the binding between a default and a column or datatype.

Here is how you unbind the current default from the *city* column of the *friends_etc* table:

```
execute sp_unbindefault "friends_etc.city"
```

At this point the default still exists, but it has no effect on the *city* column because it isn't connected to that column.

To unbind a default from the user-defined datatype *citytype*, give this command:

```
sp_unbindefault citytype
```

The complete syntax of the **sp_unbindefault** system procedure is:

```
sp_unbindefault objname [, futureonly]
```

If the *objname* parameter you give is not of the form *table.column*, SQL Server assumes it is a user-defined datatype. When you unbind a default from a user-defined datatype, the default is unbound from all columns of that type unless:

- You give the optional second parameter **futureonly**, which prevents existing columns of that datatype from losing their binding with the default, or
- The default on a column of that user-defined datatype has been changed so that its current value is different from the default being unbound.

Here is an example that demonstrates the above situation:

1. Create a user-defined datatype called *nm*.
2. Use *nm* in the **create** statements for the tables *friends_etc* and *enemies*, to create the columns *friends_etc.pname*, *friends_etc.sname*, and *enemies.nickname*.
3. Create a default called *nmdflt* and bind it to *nm*.
4. Change the default on *enemies.nickname* by creating a new default called *nastydflt* and binding it to *enemies.nickname*.
5. Now, if you unbind *nmdflt* from *nm*, only *friends.pname* and *friends.sname* are affected. Since the original default on

enemies.nickname has been changed, that column's default is not unbound, even though it is defined as type *nm*.

Dropping Defaults

If you want to remove a default from the database entirely, use the **drop default** command. The default must be unbound from all columns and user datatypes before you can drop it. If you try to **drop** a default that is still bound, SQL Server displays an error message and the **drop default** command fails. However, you need not unbind and then **drop** a default in order to bind a new one. Simply bind another default in its place.

Here is how to remove *citydflt*:

```
drop default citydflt
```

The complete syntax of the **drop default** command is:

```
drop default [owner.]default_name
    [, [owner.]default_name] ...
```

A default can be dropped only by its owner.

How Defaults Affect Null Values

If you specify NOT NULL when you create a column and do not create a default for it, SQL Server produces an error message whenever anyone inserts a row and fails to make an entry in that column.

When you drop a default for a NULL column, NULL is inserted in that position by SQL Server each time you add rows without entering any value for that column. When you drop a default for a NOT NULL column, you will get an error message when rows are added but no value for that column is explicitly entered.

The following table illustrates the relationship between the existence of a default and the definition of a column as NULL or NOT NULL. The entries in the table show the result:

Table 12-2: Defaults and NULL values

Column Definition	No Entry, No Default	No Entry, Default	Enter Null, No Default	Enter Null, Default
NULL	Null	Default	Null	Null
NOT NULL	Error	Default	Error	Error

Creating Rules

Rules are created with the **create rule** command, and then bound to a column or user-defined datatype with the **sp_bindrule** system procedure. You can unbind a rule from the column or datatype using the **sp_unbindrule** system procedure or by binding a new rule to the column or datatype.

create rule Syntax

The syntax of the **create rule** command is:

```
create rule [owner.]rule_name
   as condition_expression
```

Rule names must follow the rules for identifiers. You can create a rule in the current database only.

Within a database, rule names must be unique for each user. A user cannot create two rules called *socsecrule*. However, two different users can create a rule named *socsecrule*, because the owner names make each one distinct.

Here is how the rule permitting five different *pub_id* numbers and one dummy value (99 followed by any two digits) was created:

```
create rule pub_idrule
as @pub_id in("1389", "0736", "0877", "1622",
"1756")
or @pub_id like "99[0-9][0-9]"
```

The **as** clause contains the name of the rule's argument, prefixed with "@", and the definition of the rule itself. The argument refers to the column value that is affected by the **update** or **insert** statement.

In the preceding example, the argument is *@pub_id*, a convenient name since this rule is to be bound to the *pub_id* column. You can use **any** name for the argument, but the first character must be "*@*." Using the name of the column or datatype to which the rule will be bound can help you remember what it is for.

The rule definition can contain any expression that is valid in a **where** clause, and can include arithmetic operators, comparison operators, **like**, **in**, **between**, and so on. However, the rule definition cannot reference any column or other database object directly. Built-in functions that do not reference database objects **can** be included.

The following example creates a rule that forces the values you enter to comply with a particular "picture." In this case, each value entered in the column must begin with the digits "415" followed by seven more characters:

```
create rule phonerule
as @phone like '415_______'
```

To make sure that the ages you enter for your friends are between 1 and 120, but never 17, try this:

```
create rule agerule
as @age between 1 and 120 and @age !=17
```

Binding Rules

After you have created a rule, use the system procedure **sp_bindrule** to link the rule to a column or user-defined datatype.

Here is the complete syntax for **sp_bindrule**:

```
sp_bindrule rulename, objname [, futureonly]
```

The *rulename* is the name of the rule created with **create rule**. The *objname* is the name of the table and column, or of the user-defined datatype to which the rule is to be bound. If the parameter is not of the form *table.column*, it is assumed to be a user datatype.

The optional third parameter, **futureonly**, makes sense only when binding a rule to a user-defined datatype. All columns of a specified user-defined datatype become associated with the specified rule unless you specify **futureonly**, which prevents existing columns of that user datatype from inheriting the rule. If the rule associated with a given user-defined datatype has previously been changed the changed rule is maintained for existing columns of that user-defined datatype.

➤ ***Note***

You can't bind a rule to a *text*, *image*, or *timestamp* datatype column.

Rules Bound to Columns

You bind a rule to a column by using the **sp_bindrule** procedure with the rule name and the quoted table name and column name. This is how *pub_idrule* was bound to *publishers.pub_id*:

```
sp_bindrule pub_idrule, "publishers.pub_id"
```

As another example, here's a rule to ensure that all the postal codes you enter will have 946 as the first three digits:

```
create rule postalcoderule946
as @postalcode like "946[0-9][0-9]"
```

Bind it to the *postalcode* column in *friends_etc* like this:

```
sp_bindrule postalcoderule946,
"friends_etc.postalcode"
```

Rules cannot be bound to columns and used during the same batch. **sp_bindrule** cannot be in the same batch as **insert** statements which invoke the rule.

Rules Bound to User-Defined Datatypes

You can't bind a rule to a system datatype, but you can bind one to a user-defined datatype. To bind *phonerule* to a user-defined datatype called *p#*, type:

```
sp_bindrule phonerule, "p#"
```

Precedence of Rules

Rules bound to columns always take precedence over rules bound to user datatypes. Binding a rule to a column will replace a rule bound to the user datatype of that column, but binding a rule to a datatype will not replace a rule bound to a column of that user datatype.

A rule bound to a user-defined datatype is activated only when you attempt to insert a value into, or update, a database column of the user-defined datatype. Because rules do not test variables, be careful not to assign a value to a user-defined datatype variable that would be rejected by a rule bound to a column of the same datatype.

The following chart indicates the precedence when binding rules to columns and user datatypes where rules already exist:

Table 12-3: Precedence of rules

New Rule Bound To	Old Rule Bound To	
	User Datatype	**Column**
User Datatype	Replaces old rule	No change
Column	Replaces old rule	Replaces old rule

When you are entering data that requires special temporary constraints on some columns, you can create a new rule to help check the data. For example, suppose that you are adding data to the *debt* column of the *friends_etc* table. You know that all the debts you want to record today are between $5 and $200. In order to avoid accidentally typing an amount outside these limits, create a rule like this one. (The rule definition allows for an entry of $0.00 in order to maintain the default previously defined for this column.)

```
create rule debtrule
as @debt = $0.00 or @debt between $5.00 and $200.00
```

Bind *debtrule* to the *debt* column like this:

```
sp_bindrule debtrule, "friends_etc.debt"
```

➤ ***Note***

After you create and bind a rule, always test it by inserting data. Many errors in creating and binding rules can be caught only by testing with an **insert** or **update**.

Unbinding Rules

Unbinding a rule means disconnecting it from a particular column or user-defined datatype. An unbound rule's definition is still stored in the database and is available for future use.

There are two ways to unbind a rule:

- Use the system procedure **sp_unbindrule** to remove the binding between a rule and a column or user-defined datatype.

- Use the system procedure **sp_bindrule** to bind a new rule to that column or datatype. The old one is automatically unbound.

Here is how to disassociate *debtrule* (or any other currently bound rule) from *friends_etc.debt*:

```
sp_unbindrule "friends_etc.debt"
```

The rule is still in the database, but it has no connection to *friends_etc.debt*.

To unbind a rule from the user-defined datatype *p#*, give this command:

```
sp_unbindrule "p#"
```

The complete syntax of the **sp_unbindrule** system procedure is:

```
sp_unbindrule objname [, futureonly]
```

If the *objname* parameter you give is not of the form *"table.column"*, SQL Server assumes it is a user-defined datatype. When you unbind a rule from a user-defined datatype, the rule is unbound from all columns of that type unless:

- You give the optional second parameter **futureonly,** which prevents existing columns of that datatype from losing their binding with the rule, or
- The rule on a column of that user-defined datatype has been changed so that its current value is different from the rule being unbound.

Dropping Rules

If you want to remove a rule from the database entirely, use the **drop rule** command. The rule must be unbound from all columns and user datatypes before you can drop it. If you try to drop a rule that is still bound, SQL Server displays an error message and the **drop rule** command fails. However, you need not unbind and then drop a rule in order to bind a new one. Simply bind a new one in its place.

Here is how to remove *phonerule* after unbinding it:

```
drop rule phonerule
```

The complete syntax for **drop rule** is:

```
drop rule [owner.]rule_name
    [, [owner.]rule_name] ...
```

After you drop a rule, new data entered into the columns that previously were governed by it goes in without these constraints. Already existing data is not affected in any way.

A rule can be dropped only by its owner.

Getting Information About Defaults and Rules

The system procedure **sp_help**, when used with a table name, displays the rules and defaults that are bound to columns. This example displays information about the *authors* table in the *pubs* database, including the rules and defaults:

```
sp_help authors
```

The **sp_help** procedure also reports on a rule bound to a user-defined datatype. To check whether a rule is bound to the user-defined datatype *p#*, give this command:

```
sp_help "p#"
```

The **sp_helptext** procedure reports the definition (the **create** statement) of a rule or default.

13 Using Batches and Control-of-Flow Language

Transact-SQL allows you to group a series of statements as a batch, either interactively or from an operating system file. You can also use the control-of-flow constructs offered by Transact-SQL to connect the statements using programming-like constructs.

This chapter discusses:

- A general overview of batches and the control-of-flow language
- The rules associated with using statements in batches
- How to use the control-of-flow language

What Are Batches and Control-of-Flow Language?

Up to this point, each example in the *Transact-SQL User's Guide* has consisted of an individual statement. You submit individual statements to SQL Server one at a time, entering statements and receiving results interactively. SQL Server can also process multiple statements submitted as a batch, either interactively or from a file.

A batch of SQL statements is terminated by an end-of-batch signal that instructs SQL Server to go ahead and execute the statements. The end-of-batch signal for the standalone SQL utility **isql** is the word "go" on a line by itself. For details, see the SQL Server utility programs manual.

Technically speaking, a single SQL statement can constitute a batch, but it is more common to think of a batch as containing multiple statements. Frequently, a batch of statements is written to an operating system file before being submitted to **isql**.

Transact-SQL provides special keywords called control-of-flow language that allow the user to control the flow of execution of statements. Control-of-flow language can be used in single statements, in batches, in stored procedures, and in triggers.

Without control-of-flow language, separate SQL statements are performed sequentially, as they occur. Correlated subqueries, discussed in Chapter 5, "Subqueries: Using Queries Within Other Queries," are a partial exception. Control-of-flow language permits statements to connect and to relate to each other using programming-like constructs.

Control-of-flow language, such as **if...else** for conditional performance of commands and **while** for repetitive execution, lets you refine and control the operation of SQL statements. Transact-SQL's control-of-flow language transforms standard SQL into a very high-level programming language.

Rules Associated with Batches

There are rules governing which SQL statements can be combined into a single batch. These batch rules include the following:

- Certain database commands **cannot** be combined with other statements in a batch. These are:

  create procedure

  create rule

  create default

  create trigger

  create view

- Commands that **can** be combined with other SQL statements in a batch include:

  create database (except that you cannot create a database, and create or access objects in the new database in a single batch)

  create table

  create index

- Rules and defaults cannot be bound to columns and used during the same batch. **sp_bindrule** and **sp_bindefault** cannot be in the same batch as **insert** statements that invoke the rule or default.
- **use** must be submitted in a prior batch before statements that reference objects in that database.
- You cannot **drop** an object and then reference or re-create it in the same batch.
- Any options set with a **set** statement take effect at the end of the batch. You can combine **set** statements and queries in the same batch, but the **set** options won't apply to the queries in that batch.

Examples of Using Batches

The examples in this section illustrate batches using the format of the **isql** utility, which has a clear end-of-batch signal—the word "go" on a line by itself. Here's a batch that contains two **select** statements in a single batch:

```
select count(*) from titles
select count(*) from authors
go

-------------
          18

(1 row affected)
-------------
          23

(1 row affected)
```

You can create a table and reference it in the same batch. This batch creates a table, inserts a row into it, and then selects everything from it:

```
create table test
  (column1 char(10), column2 int)
insert test
  values ("hello", 598)
select * from test
go

(1 row affected)
column1  column2
-------  -------
hello        598

(1 row affected)
```

A **create view** statement must be the only statement in a batch. This batch contains a single statement, which creates a view:

```
create view testview as
   select column1 from test
go
```

You can combine a **use** statement with other statements so long as objects you reference in subsequent statements are in the database in which you started. This batch selects from a table in the *master* database, and then opens the *pubs2* database. It assumes that you are in the *master* database at the beginning. After the batch is executed, *pubs2* is the current database.

```
select count(*) from sysdatabases
use pubs2
go


-------------
           9

(1 row affected)
```

You can combine a **drop** statement with other statements as long as you don't reference or re-create the dropped object in the same batch. The final batch example combines a **drop** statement with a **select** statement:

```
drop table test
select count(*) from titles
go

------------
         18
(1 row affected)
```

If there is a syntax error anywhere in the batch, none of the statements is executed. For example, here is a batch with a typing error in the last statement, and the results:

```
select count(*) from titles
select count(*) from authors
slect count(*) from publishers
go

Msg 156, Level 15, State 1:
SQL Server 'MAGOO', LIne 3:
Incorrect syntax near the keyword 'count'.
```

Batches that violate a batch rule also generate error messages. Here are some examples of illegal batches:

```
create table test
    (column1 char(10), column2 int)
insert test
    values ("hello", 598)
select * from test
create view testview as select column1 from test
go

Msg 111, Level 15, State 3:
Server 'hq', Line 6:
CREATE VIEW must be the first command in a
query batch.
```

```
create view testview as select column1 from test
insert testview values ("goodbye")
go
Msg 127, Level 15, State 1:
Server 'hq', Procedure 'testview', Line 3:
This CREATE may only contain 1 statement.
```

The next batch will work if you are already in the database you specify in the **use** statement. If you try it from another database such as *master*, however, you will get an error message.

```
use pubs2
select * from titles
go
Msg 208, Level 16, State 1:
Server 'hq', Line 2:
Invalid object name 'titles'.
drop table test
create table test
(column1 char(10), column2 int)
go
Msg 2714, Level 16, State 1:
Server 'hq', Line 2:
There is already an object named 'test' in the
database.
```

Batches Submitted As Files

You can submit one or more batches of SQL statements to **isql** from an operating system file. A file can include more than one batch—more than one collection of statements, each terminated by the word "go."

For example, an operating system file might contain the following three batches:

```
use pubs2
go
select count(*) from titles
select count(*) from authors
go
create table test
   (column1 char(10), column2 int)
insert test
   values ("hello", 598)
select * from test
go
```

Here are the results of submitting this file to the **isql** utility:

```
-------------
           18

(1 row affected)
-------------
           23

(1 row affected)
(1 row affected)
column1      column2
---------    ---------
hello              598

(1 row affected)
```

See the section on the **isql** utility in the SQL Server utility programs manual for environment-specific information on running batches stored in files.

Using Control-of-Flow Language

Control-of-flow language can be used with interactive statements, in batches, and in stored procedures. The control-of-flow and related keywords and their functions are:

Table 13-1: Control-of-flow and related keywords

Keyword	Function
if	Defines conditional execution.
...else	Defines alternate execution when the **if** condition is false.
begin	Beginning of a statement block.
...end	End of a statement block.
while	Repeat performance of statements while condition is true.
break	Exit from the end of the next outermost **while** loop.
...continue	Restart while loop.
declare	Declare local variables.
goto label	Go to *label:*, a position in a statement block.
return	Exit unconditionally.

Table 13-1: Control-of-flow and related keywords (continued)

Keyword	Function
waitfor	Set delay for command execution.
print	Print a user-defined message or local variable on user's screen.
raiserror	Print a user-defined message or local variable on user's screen and set a system flag in the global variable *@@error*.
/* *comment* ****/**	Insert a comment anywhere in a SQL statement.

if...else

The keyword **if**, with or without its companion **else**, is used to introduce a condition that determines whether the next statement is executed. The SQL statement executes if the condition is satisfied, that is, if it returns TRUE.

The **else** keyword introduces an alternate SQL statement that executes when the **if** condition returns FALSE.

The syntax for **if** and **else** is:

```
if
      boolean_expression
      statement
   [else
      [if boolean_expression]
      statement ]
```

A Boolean expression is an expression that returns TRUE or FALSE. It can include a column name, a constant, any combination of column names and constants connected by arithmetic or bitwise operators, or a subquery, as long as the subquery returns a single value. If the Boolean expression contains a **select** statement, the **select** statement must be enclosed in parentheses, and it must return a single value.

Here's an example of using **if** alone:

```
if exists (select postalcode from authors
         where postalcode = '94705')
print "Berkeley author"
```

If one or more of the zip codes in the authors table has the value "94705", the message "Berkeley author" is printed. The **select** statement in this example returns a single value, either TRUE or FALSE, because it is used with the keyword **exists**. The **exists** keyword

functions here just as it does in subqueries. See Chapter 5, "Subqueries: Using Queries Within Other Queries."

Here's an example, using both **if** and **else**, that tests for the presence of user-created objects, all of which have ID numbers that are larger than 50. If user objects exist, the **else** clause selects their names, types, and ID numbers.

```
if (select max(id) from sysobjects) < 50
   print "There are no user-created objects in
this database."
else
   select name, type, id from sysobjects
   where id > 50 and type = "U"

 (0 rows affected)

 name           type id
------------    ----------
 authors        U 1088006907
 publishers     U 1120007021
 roysched       U 1152007135
 sales          U 1184007249
 titleauthor    U 1216007363
 titles         U 1248007477
 stores         U 1280007591
 discounts      U 1312007705
 test           U 1648008902

(9 rows affected)
```

if...else constructs are frequently used in stored procedures where they test for the existence of some parameter.

if tests can nest within other **if** tests, either within another **if** or following an **else**. The expression in the **if** test can return only one value. Also, for each **if...else** construct, there can be one select statement for the **if** and one for the **else**. To include more than one select statement, use the **begin...end** keywords. The maximum number of **if** tests you can nest varies on the complexity of any **select** statements (or other language constructs) you include with each **if...else** construct.

begin...end

The **begin** and **end** keywords are used to enclose a series of statements so that they are treated as a unit by control-of-flow constructs like **if...else**. A series of statements enclosed by **begin** and **end** is called a **statement block**.

The syntax of **begin...end** is:

```
begin
      statement block
   end
```

Here's an example:

```
   if (select avg(price) from titles) < $15
   begin
      update titles
      set price = price * 2

      select title, price
      from titles
      where price > $28
   end
```

Without **begin** and **end**, the **if** condition would apply only to the first SQL statement. The second statement would execute independently of the first.

begin...end blocks can nest within other **begin...end** blocks.

while and *break...continue*

while is used to set a condition for the repeated execution of a statement or statement block. The statements are executed repeatedly as long as the specified condition is true.

The syntax is:

```
while boolean_expression
   statement
```

In this example, the **select** and **update** statements are repeated as long as the average price remains less than $30:

```
   while (select avg(price) from titles) < $30
   begin
      select title_id, price
      from titles
      where price > $20
      update titles
      set price = price * 2
   end
```

```
(0 rows affected)
title_id    price
------      -------
PC1035      22.95
PS1372      21.59
TC3218      20.95

(3 rows affected)
(18 rows affected)
(0 rows affected)
title_id    price
------      -------
BU1032      39.98
BU1111      23.90
BU7832      39.98
MC2222      39.98
PC1035      45.90
PC8888      40.00
PS1372      43.18
PS2091      21.90
PS3333      39.98
TC3218      41.90
TC4203      23.90
TC7777      29.98

(12 rows affected)
(18 rows affected)
(0 rows affected)
```

break and **continue** control the operation of the statements inside a **while** loop. **break** causes an exit from the **while** loop. Any statements that appear after the **end** keyword that marks the end of the loop are executed. **continue** causes the **while** loop to restart, skipping any statements after **continue** but inside the loop. **break** and **continue** are often activated by an **if** test.

The syntax for **break...continue** is:

```
while boolean expression
   begin
       statement
       [statement]...
       break
       [statement]...
       continue
       [statement]...
   end
```

Here is an example using **while**, **break**, **continue**, and **if** that reverses the inflation caused in the previous examples. As long as the average

price remains more than $20, the prices are all cut in half. The maximum price is then selected. If it is less than $40, the **while** loop is exited; otherwise, it will try to loop again. The **continue** allows the **print** statement to execute only when the average is over $20. After the **while** loop ends, a message and a list of the highest priced books are printed.

```
while (select avg(price) from titles) > $20
begin
    update titles
        set price = price / 2
    if (select max(price) from titles) < $40
        break
    else
        if (select avg(price) from titles) < $20
            continue
    print "Average price still over $20"
end

select title_id, price from titles
    where price > $20

print "Not Too Expensive"
```

```
(18 rows affected)
(0 rows affected)
(0 rows affected)
Average price still over $20
(0 rows affected)
(18 rows affected)
(0 rows affected)
title_id    price
--------    -------
PC1035        22.95
PS1372        21.59
TC3218        20.95

(3 rows affected)
Not Too Expensive
```

If two or more **while** loops are nested, **break** exits to the next outermost loop. First all the statements after the **end** of the inner loop execute. Then the outer loop restarts.

declare and Local Variables

A variable is an entity that is assigned a value. This value can change during the batch or **stored procedure** in which the variable is used. SQL Server has two kinds of variables: local and global. Local variables are user-defined, whereas global variables are supplied by the system and are predefined.

Local variables are declared, named, and typed using the **declare** keyword, and are assigned an initial value with a **select** statement. They must be declared, assigned a value, and used all within the same batch or procedure.

Local variables are often used in a batch or stored procedure as counters for **while** loops or **if...else** blocks. When they are used in stored procedures, they are declared for automatic, noninteractive use by the procedure when it executes.

The names of local variables must begin with the "@" sign and then follow the rules for identifiers. For each local variable, you must specify either a user-defined datatype or a system-supplied datatype other than *text*, *image*, or *sysname*.

Local variables are declared with this syntax:

```
declare @variable_name datatype
      [, @variable_name datatype]...
```

When you declare a variable, it has the value NULL. Values are assigned to local variables with a **select** statement. Here is the syntax:

```
select @variable_name = { expression |
   (select_statement) } [, @variable =
   { expression | (select_statement) } ...]
   [from clause] [where clause] [group by clause]
   [having clause] [order by clause] [compute clause]
```

Local variables must be declared and used in the same batch or procedure.

The **select** statement that assigns a value to the local variable usually returns a single value. A subquery that assigns a value to the local variable **must** return only one value. Here are some examples:

```
declare @veryhigh money
select @veryhigh = max(price)
   from titles
if @veryhigh > $20
   print "Ouch!"
```

```
declare @one varchar(18), @two varchar(18)
select @one = "this is one", @two = "this is two"
if @one = "this is one"
   print "you got one"
if @two = "this is two"
   print "you got two"
else print "nope"

declare @tcount int, @pcount int
select @tcount = (select count(*) from titles),
   @pcount = (select count(*) from publishers)
select @tcount, @pcount
```

select statements using expressions that return more than one value assign the last value that is returned to the variable.

It is more efficient in terms of both memory usage and performance to write:

```
select @a = 1, @b = 2, @c = 3
```

than to write:

```
select @a = 1
select @b = 2
select @c = 3
```

A similar rule applies to **declare** statements. It is more efficient to write:

```
declare @a int, @b char(20), @c float
```

than to write:

```
declare @a int
declare @b char(20)
declare @c float
```

The **select** statement that assigns values to variables has only that one mission. It cannot also be used to return data to the user. The first **select** statement in the following example assigns the maximum price to the local variable *@veryhigh*; the second **select** statement is needed to display the value:

```
declare @veryhigh money
select @veryhigh = max(price)
   from titles
select @veryhigh
```

If the **select** statement that assigns values to a variable returns more than one value, the last value that is returned is assigned to the variable. The following query assigns the variable the last value returned by “select advance from titles”.

```
declare @m money
select @m = advance from titles
select @m

(18 rows affected)
------------------------
               8,000.00

(1 row affected)
```

Note that the assignment statement indicates how many rows were affected (returned) by the **select** statement.

If a **select** statement that assigns values to a variable fails to return any values, the variable is left unchanged by the statement.

Local variables can be used as arguments to **print** or **raiserror.**

Variables and Null Values

Local variables are assigned the value NULL when they are declared, and may be assigned the null value by a **select** statement. The special meaning of NULL requires that comparison between null-value variables and other null values follows special rules.

This table shows the results of comparisons between null-value columns and null-value expressions using different comparison operators. (An expression can be a variable, a literal, or a combination of variables and literals and arithmetic operators.)

Table 13-2: Comparing null values

	Using =, !=, or <> operators	Using <, >, <=, !<, or !> operators
Comparing *column_value* to *column_value*	FALSE	FALSE
Comparing *column_value* to *expression*	TRUE	FALSE
Comparing *expression* to *column_value*	TRUE	FALSE
Comparing *expression* to *expression*	TRUE	FALSE

For example, this test:

```
declare @v int, @i int
if @v = @i select “null = null, true”
if @v > @i select “null > null, true”
```

shows that only the first comparison returns true:

```
-----------------
null = null, true

(1 row affected)
```

This example returns all the rows from the *titles* table where the *advance* has the value NULL:

```
declare @m money
select title_id, advance
from titles
where advance = @m

title_id advance
-------- ----------------
MC3026              NULL
PC9999              NULL
```

declare and Global Variables

Global variables are system-supplied, predefined variables. They are distinguished from local variables by having two "@" signs preceding their names—for example, *@@error*.

These are the global variables:

Table 13-3: SQL Server global variables

Variable	Contents
@@char_convert	Contains 0 if character set conversion not in effect. Contains 1 if character set conversion is in effect.
@@client_csname	Contains client's character set name. Set to NULL if client character set has never been initialized; otherwise, it contains the name of the most recently used character set.
@@client_csid	Contains client's character set ID. Set to -1 if client character set has never been initialized; otherwise, it contains the most recently used client character set's *id* from *syscharsets*.
@@connections	Contains the number of logins or attempted logins.
@@cpu_busy	Contains the amount of time, in ticks, that the CPU has spent doing SQL Server work since the last time SQL Server was started.

Table 13-3: SQL Server global variables (continued)

Variable	Contents
@@error	Contains 0 if the last transaction succeeded, otherwise contains the last error number generated by the system. The *@@error* global variable is commonly used to check the error status, whether succeeded or failed, of the most recently executed statement. A statement such as **if @@error != 0** followed by **return** causes an exit on error.
@@identity	Contains the last value inserted into an IDENTITY column by an **insert** or **select into** statement. *@@identity* is set each time a row is inserted into a table. If a statement inserts multiple rows, *@@identity* reflects the IDENTITY value for the last row inserted. If the affected table does not contain an IDENTITY column, *@@identity* is set to 0. The value of *@@identity* is not affected by the failure of an **insert** or **select into** statement, or the rollback of the transaction that contained it. *@@identity* retains the last value inserted into an IDENTITY column, even if the statement that inserted it fails to commit.
@@idle	Contains the amount of time, in ticks, that SQL Server has been idle.
@@io_busy	Contains the amount of time, in ticks, that SQL Server has spent doing input and output operations.
@@isolation	Contains the current isolation level of the Transact-SQL program. *@@isolation* takes the value of the active level (1 or 3).
@@langid	Defines the local language ID of the language currently in use as specified in *syslanguages.langid*.
@@language	Defines the name of the language currently in use as specified in *syslanguages.name*.
@@maxcharlen	Contains maximum length, in bytes, of multibyte characters in default character set.
@@max_connections	Contains the maximum number of simultaneous connections that can be made with SQL Server in this computer environment. The user can configure SQL Server for any number of connections less than or equal to the value of *@@max_connection*s with **sp_configure "number of user connections"**.
@@ncharsize	Contains average length, in bytes, of a national character.

Table 13-3: SQL Server global variables (continued)

Variable	Contents
@@nestlevel	Contains nesting level of current execution, initially zero. Each time a stored procedure or trigger calls another stored procedure or trigger, the nesting level is incremented. If the maximum of 16 is exceeded, the transaction aborts.
@@pack_received	Contains the number of input packets read by SQL Server.
@@pack_sent	Contains the number of output packets written by SQL Server.
@@packet_errors	Contains the number of errors that have occurred while SQL Server was sending and receiving packets.
@@procid	Contains stored procedure ID of currently executing procedure.
@@rowcount	Contains the number of rows affected by the last query. *@@rowcount* is set to zero by any command which does not return rows, such as an **if** statement.
@@servername	Contains the name of this SQL Server.
@@spid	Contains the server process ID number of the current process.
@@sqlstatus	Contains status information resulting from the last **fetch** statement.
@@textcolid	Contains the column ID of the column referenced by *@@textptr*. The datatype of this variable is *tinyint*.
@@textdbid	Contains the database ID of a database containing an object with the column referenced by *@@textptr*. The datatype of this variable is *smallint*.
@@textobjid	Contains the object ID of an object containing the column referenced by *@@textptr*. The datatype of this variable is *int*.
@@textptr	Contains the text pointer of the last *text* or *image* column inserted or updated by a process. The datatype of this variable is *binary(16)*. (Do not confuse this variable with the **textptr** function.)
@@textsize	Contains the limit on the number of bytes of *text* or *image* data a **select** returns. Default limit is 32K bytes for **isql**, the default depends on the client software. Can be changed for a session with **set textsize**.

Table 13-3: SQL Server global variables (continued)

Variable	Contents
@@textts	Contains the text time stamp of the column referenced by *@@textptr*. The datatype of this variable is *varbinary(8)*.
@@thresh_hysteresis	Contains the decrease in free space required to activate a threshold. This amount, also known as the hysteresis value, is measured in 2K database pages. It determines how closely thresholds can be placed on a database segment.
@@timeticks	Contains the number of microseconds per tick. The amount of time per tick is machine dependent.
@@total_errors	Contains the number of errors that have occurred while SQL Server was reading or writing.
@@total_read	Contains the number of disk reads by SQL Server since it was last started.
@@total_write	Contains the number of disk writes by SQL Server since it was last started.
@@tranchained	Contains the current transaction mode of the Transact-SQL program. *@@tranchained* returns a 0 for unchained or a 1 for chained.
@@trancount	Contains the number of currently active transactions for the current user.
@@transtate	Contains the current state of a transaction after a statement executes. However, *@@transtate* does not get cleared for each statement, unlike *@@error*.
@@version	Contains the date of the current version of SQL Server.

For information on the contents of many of these global variables, execute the system procedure **sp_monitor**. For complete information on the system procedures, see the *SQL Server Reference Manual*.

If a user declares a local variable that has the same name as a **global variable**, that variable is treated as a local variable.

goto

The **goto** keyword causes unconditional branching to a user-defined label. **goto** and labels can be used in stored procedures and batches. A label's name must follow the rules for identifiers and must be

followed by a colon when it is first given. It is not followed by a colon when it is used with **goto**.

Here is the syntax:

```
label:
   goto label
```

Here is an example that uses **goto** and a label, a **while** loop, and a local variable as a counter:

```
declare @count smallint
select @count = 1
restart:
print "yes"
select @count = @count + 1
while @count <=4
   goto restart
```

As in this example, **goto** is usually made dependent on a **while** or **if** test or some other condition, in order to avoid an endless loop between **goto** and the label.

return

The **return** keyword exits from a batch or procedure unconditionally. It can be used at any point in a batch or a procedure. When used in stored procedures, **return** can accept an optional argument to return a status to the caller. Statements after **return** are not executed.

The syntax is simply:

```
return [int_expression]
```

The following is an example of a stored procedure that uses **return** as well as **if...else** and **begin...end**:

```
create procedure findrules @nm varchar(30) = null
as
if @nm is null
begin
  print "You must give a user name"
  return
end
else
begin
   select sysobjects.name, sysobjects.id,
sysobjects.uid
   from sysobjects, master..syslogins
   where master..syslogins.name = @nm
   and sysobjects.uid = master..syslogins.suid
   and sysobjects.type = "R"
end
```

If no user name is given as a parameter when **findrules** is called, the **return** keyword causes the procedure to exit after a message has been sent to the user's screen. If a user name is given, the names of the rules owned by the user are retrieved from the appropriate system tables.

return is similar to the **break** keyword used inside **while** loops.

Examples using return values are included in Chapter 14, "Using Stored Procedures."

print

The **print** keyword, used in the previous example, displays a user-defined message or the contents of a local variable on the user's screen. The local variable must be declared within the same batch or procedure in which it is used. The message itself can be up to 255 bytes long.

The syntax is:

```
print {format_string | @local_variable |
        @@global_variable} [,arg_list]
```

Here is another example:

```
if exists (select postalcode from authors
   where postalcode = '94705')
print "Berkeley author"
```

Here is how to use **print** to display the contents of a local variable:

```
declare @msg char(50)
select @msg = "What's up doc?"
print @msg
```

print recognizes placeholders in the character string to be printed out. Format strings may contain up to 20 unique placeholders in any order. These placeholders are replaced with the formatted contents of any arguments that follow *format_string* when the text of the message is sent to the client.

To allow reordering of the arguments when format strings are translated to a language with a different grammatical structure, the placeholders are numbered. A placeholder for an argument appears in this format: *%nn!*. The components are a percent sign, followed by an integer from 1 to 20, followed by an exclamation point. The integer represents the placeholder position in the string in the original language. "%1!" is the first argument in the original version, "%2!" is the second argument, and so on. Indicating the position of the argument in this way makes it possible to translate correctly even when the order in which the arguments appear in the target language is different from their order in the source language.

For example, assume the following is an English message:

```
%1! is not allowed in %2!.
```

The German version of this message is:

```
%1! ist in %2! nicht zulässig.
```

The Japanese version of the message is:

```
%2! の中で %1! は許されません。
```

In this example, "%1!" in all three languages represents the same argument, and "%2!" also represents a single argument in all three languages. This example shows the reordering of the arguments that is sometimes necessary in the translated form.

You cannot skip placeholder numbers when using placeholders in a format string, although placeholders do not have to be used in numerical order. For example, you cannot have placeholders 1 and 3 in a format string without having placeholder 2 in the same string.

The optional *arg_list* may be a series of either variables or constants. An argument can be any datatype except *text* or *image*; it is converted to the *char* datatype before it is included in the final message. If no argument list is provided, the format string must be the message to be printed, without any placeholders.

The maximum output string length of *format_string* plus all arguments after substitution is 512 bytes.

raiserror

raiserror both displays a user-defined error or local variable message on the user's screen, and sets a system flag to record the fact that an error has occurred. As with **print**, the local variable must be declared within the same batch or procedure in which it is used. The message can be up to 255 characters long.

Here is the syntax for **raiserror**:

```
raiserror error_number
   [{format_string | @local_variable}] [, arg_list]
   [extended_value = extended_value [{,
   extended_value = extended_value}...]]
```

The *error_number* is placed in the global variable *@@error*, which stores the error number most recently generated by SQL Server, whether it is associated with a system-supplied error message or a user-defined one. Error numbers for user-defined error messages must be greater than 17,000. If the *error_number* is between 17,000 and 19,999, and *format_string* is missing or empty (""), SQL Server retrieves error message text from the *sysmessages* table in the *master* database. These error messages are used chiefly by system procedures.

The length of the *format_string* alone is limited to 255 bytes; the maximum output length of *format_string* plus all arguments is 512 bytes. Local variables used for **raiserror** messages must be *char* or *varchar*. The *format_string* or variable is optional. If one is not included, SQL Server uses the message corresponding to the *error_number* from *sysusermessages* in the default language. As with **print**, you can substitute variables or constants defined by *arg_list* in the *format_string*.

As an option, you can define extended error data for use by an Open Client™ application (when you include *extended_values* with **raiserror**). For more information about extended error data, see your Open Client documentation or **raiserror** in the *SQL Server Reference Manual*.

Use **raiserror** instead of **print** when you want an error number stored in *@@error*. For example, here is how you could use **raiserror** in the procedure **findrules**:

```
raiserror 99999 "You must give a user name"
```

The severity level of all user-defined error messages is 16. This level indicates that the user has made a nonfatal mistake.

User-Defined Messages for *print* and *raiserror*

You can call messages from *sysusermessages* for use by either **print** or **raiserror** with the system procedure **sp_getmessage**. Use the system procedure **sp_addmessage** to create a set of messages.

The example that follows uses **sp_addmessage**, **sp_getmessage**, and **print** to install a message in *sysusermessages* in both English and German, retrieve it for use in a user-defined stored procedure, and print it.

```
/*
** Install messages
** First, the English (langid = NULL)
*/
set language us_english
go
sp_addmessage 25001,
  "There is already a remote user named '%1!' for
remote server '%2!'."
go
/* Then German*/
sp_addmessage 25001,
         "Remotebenutzername '%1!' existiert
bereits auf dem Remoteserver '%2!'.","german"
go

create procedure test_proc @remotename varchar(30),
           @remoteserver varchar(30)
as
        declare @msg varchar(255)
        declare @arg1 varchar(40)
        /*
        ** check to make sure that there is not
        ** a @remotename for the @remoteserver.
        */
        if exists (select *
            from master.dbo.sysremotelogins l,
                master.dbo.sysservers s
            where l.remoteserverid = s.srvid
```

```
                and s.srvname = @remoteserver
                and l.remoteusername = @remotename)
        begin
            exec sp_getmessage 25001, @msg output
            select @arg1=isnull(@remotename,"null")
            print @msg, @arg1, @remoteserver
            return (1)
        end
return(0)
go
```

waitfor

The **waitfor** keyword specifies a specific time of day, a time interval, or an event at which the execution of a statement block, stored procedure, or transaction is to occur.

Here is the syntax:

```
waitfor {delay "time" | time "time" | errorexit |
   processexit | mirrorexit}
```

The **delay** keyword instructs SQL Server to wait until the specified amount of time has passed. **time** instructs SQL Server to wait until the specified time, given in one of the acceptable formats for *datetime* data.

However, you cannot specify dates—the date portion of the *datetime* value is not allowed. The time you specify with **waitfor time** or **waitfor delay** can include hours, minutes, and seconds—up to a maximum of 24 hours. Use the format "hh:mm:ss". For example, **waitfor time** *"16:23"* instructs SQL Server to wait for 4:23 pm. The statement **waitfor delay** *"01:30"* instructs SQL Server to wait one hour and 30 minutes. For a review of the acceptable formats for **time** values, see Chapter 8, "Adding, Changing, and Deleting Data."

errorexit instructs SQL Server to wait until a process terminates abnormally. **processexit** waits until a process terminates for any reason. **mirrorexit** waits until a read or write to a mirrored device fails.

You can use **waitfor errorexit** with a procedure that kills the abnormally terminated process in order to free system resources that would otherwise be taken up by an infected process. To find out which process is infected, check the *sysprocesses* table with the system procedure **sp_who**.

This example instructs SQL Server to wait until 2:20 p.m. Then, it updates the *chess* table with the next move and executes a stored procedure called **sendmessage**, which inserts a message into one of

Judy's tables notifying her that a new move now exists in the *chess* table. Here it is:

```
begin
waitfor time "14:20"
insert chess(next_move)
values('Q-KR5')
execute sendmessage 'judy'
end
```

To send the message to judy after 10 seconds instead of waiting until 2:20, substitute this **waitfor** statement in the preceding example:

```
waitfor delay "0:00:10"
```

After you give the **waitfor** command, you cannot use your connection to SQL Server until the time or event that you specified occurs.

Comments

The comment notation is used to attach comments to statements, batches, and stored procedures. A comment looks like this:

```
/* text of comment */
```

There is no maximum length for comments, and they can be inserted anywhere, on a line by themselves or at the end of a line. Multiple-line comments are fine, too, so long as each comment starts with a slash and an asterisk, and ends with an asterisk and a slash. Everything between "/*" and "*/" is treated as part of the comment. Comments can be nested.

A stylistic convention that's often used for multiple-line comments is to begin the first line with "/*" and subsequent lines with "**". The comment is ended with "*/" as usual. Here is what it looks like:

```
select * from titles
/* A comment here might explain the rules
** associated with using an asterisk as
** shorthand in the select list.*/
where price > $5
```

Here is a procedure that includes a couple of comments:

```
/* this procedure finds rules by user name*/

create procedure findrules2 @nm varchar(30) = null
as if @nm is null /*if no parameter is given*/
print "You must give a user name"
else
begin
   select sysobjects.name, sysobjects.id,
        sysobjects.uid
   from sysobjects, master..syslogins
   where master..syslogins.name = @nm
   and sysobjects.uid = master..syslogins.suid
   and sysobjects.type = "R"
end
```

14 Using Stored Procedures

You can group SQL statements and control-of-flow language in a stored procedure to improve the performance of SQL Server. Also, you can use a group of predefined procedures, called system stored procedures, to perform administrative tasks and to update the system tables.

This chapter discusses:

- A general overview of stored procedures
- How to create and execute stored procedures
- How to return information from stored procedures
- The rules associated with stored procedures
- How to drop and rename stored procedures
- How to use the system stored procedures
- How to get information about stored procedures

What Are Stored Procedures?

Stored procedures are collections of SQL statements and control-of-flow language. An execution plan is prepared when a procedure is run, so that subsequent execution is very fast. Stored procedures can:

- Take parameters
- Call other procedures
- Return a status value to a calling procedure or batch to indicate success or failure, and the reason for failure
- Return values of parameters to a calling procedure or batch
- Be executed on remote SQL Servers

The ability to write stored procedures greatly enhances the power, efficiency, and flexibility of SQL. Compiled procedures dramatically improve the performance of SQL statements and batches. In addition, stored procedures on other SQL Servers can be executed if your server and the remote server are both set up to allow remote logins. You can write triggers on your local SQL Server that execute procedures on a remote server whenever certain events, such as deletions, updates, or inserts, take place locally.

Stored procedures differ from ordinary SQL statements and from batches of SQL statements in that they are precompiled. The first time you run a procedure, SQL Server's query processor analyzes it and prepares an execution plan that is ultimately stored in a **system table**. Subsequently, the procedure is executed according to the stored plan. Since most of the query processing work has already been performed, stored procedures execute almost instantaneously.

SQL Server supplies a variety of stored procedures as convenient tools for the user. These stored procedures are called system procedures.

You create stored procedures with the **create procedure** command. To execute a stored procedure, either a system procedure or a user-defined procedure, use the **execute** command. Or you can use the name of the stored procedure alone, as long as it is the first word of a statement or batch.

Examples of Creating and Using Stored Procedures

The syntax for creating a simple stored procedure, without special features such as parameters, is:

```
create procedure procedure_name
   as SQL_statements
```

Stored procedures are database objects, and their names must follow the rules for identifiers.

Any number and kind of SQL statements can be included with the exception of **create** statements. See "Rules Associated with Stored Procedures" on page 14-22. A procedure can be as simple as a single statement that lists the names of all the users in a database:

```
create procedure namelist
as select name from sysusers
```

To execute a stored procedure, use the keyword **execute** and the name of the stored procedure, or just give the procedure's name, as long as it is submitted to SQL Server by itself, or is the first statement in a batch. You can execute **namelist** in any of these ways:

```
namelist

execute namelist

exec namelist
```

To execute a stored procedure on a remote SQL Server, you must give the server name. The full syntax for a remote procedure call is:

```
execute
   server_name.[database_name].[owner].procedure_name
```

The following examples all execute the procedure **namelist** in the *pubs2* database on the GATEWAY server:

```
execute gateway.pubs2..namelist

gateway.pubs2.dbo.namelist

exec gateway...namelist
```

The last example works only if *pubs2* is your default database.

The database name is optional only if the stored procedure is located in your **default database**. The owner name is optional only if the Database Owner ("dbo") owns the procedure or if you own it. Of course, you must have **permission** to execute the procedure.

A procedure can include more than one statement.

```
create procedure showall as
select count(*) from sysusers
select count(*) from sysobjects
select count(*) from syscolumns
```

When the procedure is executed, the results of each command are displayed in the order that the statement appears in the procedure.

```
showall

------------
           5
(1 row affected)

------------
          88
(1 row affected)

------------
         349
(1 row affected, return status = 0)
```

When a **create procedure** command is successfully executed, the procedure's name is stored in *sysobjects*, and its text in *syscomments*.

You can display the text of a procedure with the system procedure **sp_helptext**:

```
sp_helptext showall
```

```
# Lines of Text
--------------
             1

(1 row affected)

text
--------------------------------------
create procedure showall as
select count(*) from sysusers
select count(*) from sysobjects
select count(*) from syscolumns
(1 row affected, return status = 0)
```

Stored Procedures and Permissions

Stored procedures can serve as security mechanisms, since a user can be granted permission to execute a stored procedure even if she or he does not have permissions on tables or views referenced in it or permission to execute specific commands. For details, see the *Security Features User's Guide*.

Stored Procedures and Performance

As a database changes, you can re-optimize the original query plans used to access its tables by recompiling them with the system procedure **sp_recompile**. This saves you the work of having to find, drop, and then re-create every stored procedure and trigger. This example marks every stored procedure and trigger that accesses the table *titles* to be recompiled the next time it is executed.

```
sp_recompile titles
```

For detailed information about **sp_recompile**, see the *SQL Server Reference Manual*.

Creating and Executing Stored Procedures

The complete syntax for **create procedure** is:

```
create procedure [owner.]procedure_name[;number]
   [[(]@parameter_name datatype [= default] [output]
   [, @parameter_name datatype [= default]
   [output]]...[)]] [with recompile]
   as sql_statements
```

You can create a procedure in the current database only.

Permission to issue **create procedure** defaults to the Database Owner, who can transfer it to other users.

Here is the complete syntax statement for **execute**:

```
[execute] [@return_status = ]
   [[[server.]database.]owner.]procedure_name[;number]
       [[@parameter_name =] value |
   [@parameter_name =] @variable[output]
   [,[@parameter_name =] value |
   [@parameter_name =] @variable [output]...]]
   [with recompile]
```

➤ ***Note***

Remote procedure calls are not treated as part of a transaction. If you execute a remote procedure call after **begin transaction**, and then **rollback transaction**, any changes that the remote procedure call made on remote data are not rolled back. The stored procedure designer should be sure that all conditions which might trigger a rollback are checked before issuing a remote procedure call, which will alter remote data.

Parameters

A **parameter** is an argument to a stored procedure. One or more parameters can optionally be declared in a create procedure statement. The value of each parameter named in a **create procedure** statement must be supplied by the user when the procedure is executed.

Parameter names must be preceded by the "@" symbol and must conform to the rules for identifiers. They must be given a system datatype or a user-defined datatype, and a length if required for the datatype. Parameter names are local to the procedure that creates them; the same parameter names can be used in other procedures. Parameter names, including the "@" symbol, can be a maximum of 30 bytes long.

Here is a stored procedure that is useful in the *pubs2* database. Given an author's last and first name, it displays the names of any books that person has written and each book's publisher.

```
create proc au_info @lastname varchar(40),
  @firstname varchar(20) as
select au_lname, au_fname, title, pub_name
from authors, titles, publishers, titleauthor
where au_fname = @firstname
and au_lname = @lastname
and authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
and titles.pub_id = publishers.pub_id
```

Now we execute **au_info**:

```
au_info Ringer, Anne
au_lname au_fname title                 pub_name
-------- -------- -------------------- ----------
Ringer   Anne     The Gourmet Microwave Binnet
                                       & Hardley
Ringer   Anne     Is Anger the Enemy?   New Age
                                       Books

(2 rows affected, return status = 0)
```

The following stored procedure queries the system tables. Given a table name as the parameter, the procedure shows the table name, index name, and index ID.

```
create proc showind @table varchar(30) as
select table_name = sysobjects.name,
index_name = sysindexes.name, index_id = indid
from sysindexes, sysobjects
where sysobjects.name = @table
and sysobjects.id = sysindexes.id
```

The column headings, for example, *table_name*, were added to improve the readability of the results. Here are acceptable syntax forms for executing this stored procedure:

```
execute showind titles

exec showind titles

execute showind @table = titles

execute GATEWAY.pubs2.dbo.showind titles

showind titles
```

The last syntax form, without **exec** or **execute**, is acceptable so long as the statement is the only one or the first one in a batch.

Here are the results of executing **showind** in the *pubs2* database when *titles* is given as the parameter:

```
table_name  index_name  index_id
----------  ----------  ----------
titles      titleidind      1
titles      titleind        2

(2 rows affected, return status = 0)
```

➤ ***Note***

If you supply the parameters in the form "*@parameter = value*" you can supply them in any order. Otherwise, you must supply parameters in the order of their **create procedure** statement. If you supply one value in the form "*@parameter = value*", then all subsequent parameters must be supplied this way.

Default Parameters

You can assign a default value for the parameter in the **create procedure** statement. This value, which can be any constant, is taken as the argument to the procedure if the user does not supply one.

Here's a procedure that displays the names of all the authors that have written a book published by the publisher given as a parameter. If no publisher name is supplied, the procedure shows the authors published by Algodata Infosystems.

```
create proc pub_info
  @pubname varchar(40) = "Algodata Infosystems" as
select au_lname, au_fname, pub_name
from authors a, publishers p, titles t,
titleauthor ta
where @pubname = p.pub_name
and a.au_id = ta.au_id
and t.title_id = ta.title_id
and t.pub_id = p.pub_id
```

Note that if the default value is a character string that contains embedded blanks or punctuation, it must be enclosed in single or double quotes.

When you execute *pub_info*, you can give any publisher's name as the parameter value. If you do not supply any parameter, SQL Server uses the default, Algodata Infosystems.

```
exec pub_info
```

```
au_lname   au_fname      pub_name
-------------  ------------ --------------------
Green       Marjorie     Algodata  Infosystems
Bennet      Abraham      Algodata  Infosystems
O'Leary     Michael      Algodata  Infosystems
MacFeather  Stearns      Algodata  Infosystems
Straight    Dick         Algodata  Infosystems
Carson      Cheryl       Algodata  Infosystems
Dull        Ann          Algodata  Infosystems
Hunter      Sheryl       Algodata  Infosystems
Locksley    Chastity     Algodata  Infosystems

(9 rows affected, return status = 0)
```

You assign "titles" as the default value for the *@table* parameter in this procedure, *showind2*:

```
create proc showind2 @table varchar(30) = titles
as
select table_name = sysobjects.name,
    index_name = sysindexes.name, index_id = indid
from sysindexes, sysobjects
where sysobjects.name = @table
and sysobjects.id = sysindexes.id
```

The column headings, for example, *table_name*, clarify the results display. Here is what the procedure shows for the *authors* table:

```
showind2 authors

table_name  index_name      index_id
----------- -------------   ----------
authors     auidind              1
authors     aunmind              2

(2 rows affected, return status = 0)
```

If the user does not supply a value, SQL Server uses the default, *titles*.

```
showind2

table_name  index_name  index_id
----------- ----------- ---------
titles      titleidind         1
titles      titleind           2

(2 rows affected, return status =0)
```

If a parameter is expected but none is supplied, and a default value is not supplied in the **create procedure** statement, SQL Server displays an error message listing the parameters that the procedure expects.

NULL As Default Parameter

The default can be the value NULL. In this case, if the user does not supply a parameter, SQL Server executes the stored procedure without displaying an error message.

The procedure definition can specify an action be taken if the user does not give a parameter, by checking to see that the parameter's value is null. Here is an example:

```
create procedure showind3 @table varchar(30) = null
as
if @table is null
    print "Please give a table name"
else
   select table_name = sysobjects.name,
     index_name = sysindexes.name, index_id = indid
   from sysindexes, sysobjects
   where sysobjects.name = @table
   and sysobjects.id = sysindexes.id
```

If the user fails to give a parameter, SQL Server prints the message from the procedure on the screen.

For other examples of setting the default to NULL, examine the text of system procedures using **sp_helptext**.

Wildcard Characters in the Default Parameter

The default can include the wildcard characters (%, _, [] and [^]) if the procedure uses the parameter with the **like** keyword.

For example, **showind** can be modified to display information about the system tables if the user does not supply a parameter, like this:

```
create procedure showind4 @table varchar(30)="sys%"
as
select table_name = sysobjects.name,
    index_name = sysindexes.name, index_id = indid
from sysindexes, sysobjects
where sysobjects.name like @table
and sysobjects.id = sysindexes.id
```

Using More Than One Parameter

Here is a variant of the stored procedure *au_info* that has defaults with wildcard characters for both parameters:

```
create proc au_info2 @lastname varchar(30) = "D%",
  @firstname varchar(18) = "%" as
select au_lname, au_fname, title, pub_name
from authors, titles, publishers, titleauthor
where au_fname like @firstname
and au_lname like @lastname
and authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
and titles.pub_id = publishers.pub_id
```

If *au_info2* is executed with no parameters, all the authors with last names beginning with "D" are displayed:

```
au_info2

au_lname au_fname title                      pub_name
-------- -------  -------------------------  ------------
Dull     Ann      Secrets of Silicon Valley  Algodata Infosystems
DeFrance Michel   The Gourmet Microwave      Binnet & Hardley

(2 rows affected)
```

If defaults are available for parameters, parameters can be omitted at execution, beginning with the last parameter. You cannot skip a parameter unless NULL is its supplied default.

➤ ***Note***

If you supply parameters in the form "*@parameter* = *value*", you can supply parameters in any order. You can also omit a parameter for which a default has been supplied.

If you supply one value in the form "*@parameter* = *value*", then all subsequent parameters must be supplied this way.

As an example of omitting the second parameter when defaults for two parameters have been defined, you can find the books and publishers for all authors with the last name "Ringer" like this:

```
au_info2 Ringer

au_lname   au_fname    title                   Pub_name
--------   --------    --------------------    ------------
Ringer     Anne        The Gourmet Microwave    Binnet & Hardley
Ringer     Anne        Is Anger the Enemy?      New Age Books
Ringer     Albert      Is Anger the Enemy?      New Age Books
Ringer     Albert      Life Without Fear        New Age Books

(4 rows affected)
```

Procedure Groups

The optional semicolon and integer number after the name of the procedure in the **create procedure** and **execute** statements allow you to group procedures of the same name so that they can be dropped together with a single **drop procedure** statement.

Procedures used in the same application are often grouped this way. For example, you might create a series of procedures called **orders;1**, **orders;2**, and so on. The following statement would drop the entire group:

```
drop proc orders
```

Once procedures have been grouped by appending a semicolon and number to their names, they cannot be dropped individually. For example, the following statement is not allowed:

```
drop proc orders;2
```

with recompile in *create procedure*

In the **create procedure** statement, the optional clause **with recompile** comes just before the SQL statements. It instructs SQL Server not to save a plan for this procedure. A new plan is created each time the procedure is executed.

In the absence of **with recompile**, SQL Server stores the execution plan that it created. Usually, this execution plan is satisfactory.

However, it's possible that a change in the data or a change in parameter values supplied for subsequent executions cause SQL Server to come up with an execution plan different from the one it created when the procedure was first executed. In such situations, SQL Server needs a new execution plan.

Use **with recompile** in a **create procedure** statement when you think you need a new plan. See the *SQL Server Reference Manual* for more information.

with recompile in *execute*

In the **execute** statement, the optional clause **with recompile** comes after any parameters. It instructs SQL Server to compile a new plan. The new plan is used for subsequent executions.

Use **with recompile** when you execute a procedure if your data has changed a great deal, or if the parameter you are supplying is

atypical—that is, if you have reason to believe that the plan stored with the procedure might not be optimal for this execution of it.

➤ ***Note***

If you use **select *** in your **create procedure** statement, the procedure, even if you use the **with recompile** option to **execute**, does not pick up any new columns added to the table. You must drop the procedure and recreate it.

Nesting Procedures Within Procedures

Nesting occurs when one stored procedure or trigger calls another. The nesting level is incremented when the called procedure or trigger begins execution and it is decremented when the called procedure or trigger completes execution. Exceeding the maximum of 16 levels of nesting causes the procedure to fail. The current nesting level is stored in the *@@nestlevel* global variable.

Using Temporary Tables in Stored Procedures

You can create and use temporary tables in a stored procedure, but the temporary table exists only for the duration of the stored procedure that creates it. When the procedure completes, SQL Server automatically drops the temporary table. A single procedure can:

- Create a temporary table
- Insert, update, or delete data
- Run queries on the temporary table
- Call other procedures that reference the temporary table

Since the temporary table must exist in order to create procedures that reference it, here are the steps to follow:

1. Create the temporary table you need with a **create table** statement or a **select into** statement. For example:

   ```
   create table #tempstores
       (stor_id char(4), amount money)
   ```

2. Create the procedures that access the temporary table (but not the one that creates it).

```
create procedure inv_amounts
    as
    select stor_id, "Total Due" =sum(amount)
    from #tempstores
    group by stor_id
```

3. Drop the temporary table:

   ```
   drop table #tempstores
   ```

4. Create the procedure that creates the table and calls the procedures created in step 2:

   ```
   create procedure inv_proc
   as
   create table #tempstores
   (stor_id char(4), amount money)

   insert #tempstores
   select stor_id, sum(qty*(100-discount)/100*price)
   from salesdetail, titles
   where salesdetail.title_id = titles.title_id
   group by stor_id, salesdetail.title_id

   exec inv_amounts
   ```

You can also create temporary tables without the # prefix, using **create table *tempdb..tablename...*** from inside a stored procedure. These tables do not disappear when the procedure completes, so they can be referenced by independent procedures. Follow the above steps to create these tables.

Executing Procedures Remotely

You can execute procedures on another SQL Server from your local SQL Server. Once both servers are properly configured, you can execute any procedure on the remote SQL Server simply by using the server name as part of the identifier. For example, to execute a procedure named **remoteproc** on a server named GATEWAY:

```
exec gateway.remotedb.dbo.remoteproc
```

See the *System Administration Guide* for information on how to configure your local and remote SQL Servers for remote execution of procedures. You can pass one or more values as parameters to a remote procedure from the batch or procedure that contains the **execute** statement for the remote procedure. Results from the remote SQL Server appear on your local terminal.

The return status from procedures, described in the following sections, can be used to capture and transmit information messages about the execution status of your procedures.

◆ ***WARNING!***

Remote procedure calls are not regarded as part of a transaction. Therefore, if you execute a remote procedure call as part of a transaction, and then roll back the transaction, any changes that the remote procedure call made on a remote SQL Server are not rolled back.

Returning Information from Stored Procedures

Stored procedures report a "return status" that indicates whether they completed successfully, or the reasons for failure. This value can be stored in a variable when a procedure is called, and used in future Transact-SQL statements. SQL Server-defined return status values for failure are in the range of -1 to -99; users can define their own return status values outside this range.

Another way that stored procedures can return information to the caller is through return parameters. Parameters designated as return parameters in the **create procedure** and the **execute** statement report the parameter values back to the caller. The caller can then use conditional statements to check the returned value.

Return status and return parameters allow you to modularize your stored procedures. A set of SQL statements that are used by several stored procedures can be created as a single procedure which returns its execution status or the values of its parameters to the calling procedure. For example, many of the SQL Server-supplied system procedures execute a procedure that verifies that certain parameters are valid identifiers.

Remote procedure calls, which are stored procedures run on a remote SQL Server, also return both kinds of information. All of the examples below could be executed remotely if the syntax of the execute statement included the server, database, and owner names as well as the procedure name.

Return Status

Stored procedures can return an integer value called a **return status**. This status indicates that the procedure completed successfully or indicates the reason for failure. SQL Server has a defined set of return values. Users can also define their own return values. Here is an example of a batch that uses the form of the **execute** statement that returns the status:

```
declare @status int
execute @status = pub_info
select @status
```

The execution status of the **pub_info** procedure is stored in the variable *@status*. This example merely prints the value with a **select** statement; later examples use this return value in conditional clauses.

Reserved Return Status Values

SQL Server reserves 0 to indicate a successful return, and negative values in the range of -1 to -99 to indicate different reasons for failure. Numbers 0 and -1 to -14 are currently in use:

Table 14-1: Reserved return status values

Value	Meaning
0	Procedure executed without error
-1	Missing object
-2	Datatype error
-3	Process was chosen as deadlock victim
-4	Permission error
-5	Syntax error
-6	Miscellaneous user error
-7	Resource error, such as out of space
-8	Non-fatal internal problem
-9	System limit was reached
-10	Fatal internal inconsistency
-11	Fatal internal inconsistency
-12	Table or index is corrupt
-13	Database is corrupt
-14	Hardware error

Values -15 to -99 are reserved for future use by SQL Server.

If more than one error occurs during execution, the status with the highest absolute value is returned.

User-Generated Return Values

You can generate your own return values in stored procedures by adding a parameter to the **return** statement. Numbers from 0 to -99 are reserved for use by SQL Server; all other integers can be used. The following example returns 1 when a book has a valid contract and returns 2 in all other cases:

```
create proc checkcontract @titleid tid
as
if (select contract from titles where
        title_id = @titleid) = 1
   return 1
else
   return 2
```

The following stored procedure calls **checkcontract**, and uses conditional clauses to check the return status:

```
create proc get_au_stat @titleid tid
as
declare @retvalue int
execute @retvalue = checkcontract @titleid
if (@retvalue = 1)
   print "Contract is valid"
else
    print "There is not a valid contract"
```

Here are the results when you execute *get_au_stat* with the *title_id* of a book with a valid contract:

```
get_au_stat "MC2222"
Contract is valid
```

Checking Roles in Procedures

If a stored procedure performs system administration or security-related tasks, you may wish to ensure that only users who have been granted a specific role can execute it. (See the *Security Features User's Guide* for information about roles.) The **proc_role** function allows you to check roles when the procedure is executed. It returns 1 if the user possesses the specified role. The role names are **sa_role, sso_role,** and **oper_role.**

Here is an example using **proc_role** in the stored procedure *test_proc* to require the invoker to be a System Administrator:

```
create proc test_proc
as
if (proc_role("sa_role") = 0)
begin
    print "You don't have the right role"
    return -1
end
else
    print "You have SA role"
    return 0
```

Return Parameters

When a **create procedure** statement and **execute** statement both include the **output** option with a parameter name, the procedure returns a value to the caller. The caller can be a SQL batch or another stored procedure. The value returned can be used in additional statements in the batch or calling procedure. When return parameters are used in an **execute** statement that is part of a batch, the return values are printed with a heading before subsequent statements in the batch are executed.

This stored procedure performs multiplication on two integers. The third integer, *@result*, is defined as an **output** parameter:

```
create procedure mathtutor @mult1 int, @mult2 int,
  @result int output
as
select @result = @mult1 * @mult2
```

To use **mathtutor** to figure a multiplication problem, you must declare the *@result* variable and include it in the **execute** statement. Adding the **output** keyword to the **execute** statement displays the value of the return parameters.

```
declare @result int
exec mathtutor 5, 6, @result output
(return status = 0)

Return parameters:

-----------
         30
```

If you wanted to guess at the answer and execute this procedure by providing three integers, you wouldn't see the results of the

multiplication. The **select** statement in the procedure assigns values, but doesn't print:

```
mathtutor 5, 6, 32
(return status = 0)
```

The value for the **output** parameter must be passed as a variable, not as a constant. This example declares the *@guess* variable to store the value to pass to **mathtutor** for use in *@result*. SQL Server prints the return parameters:

```
declare @guess int
select @guess = 32
exec mathtutor 5, 6, @result = @guess output
(1 row affected)
(return status = 0)

Return parameters:

@result
-----------
        30
```

The value of the return parameter is always reported, whether its value has changed or not. Note that:

- In the example above, the **output** parameter *@result* **must** be passed as "*@parameter* = *@variable*". If it were not the last parameter passed, subsequent parameters would have to be passed as "*@parameter* = *value*".
- *@result* does not have to be declared in the calling batch; it is the name of a parameter to be passed to *mathtutor*.
- Although the changed value of *@result* is returned to the caller in the variable assigned in the **execute** statement, in this case *@guess*, it is displayed under its own heading, that is, *@result*.

If you want to use the initial value of *@guess* in conditional clauses after the **execute** statement, you must store it in another variable name during the procedure call. The following example illustrates the last two points by using *@store* to hold the value of the variable during the execution of the stored procedure, and by using the "new" returned value of *@guess* in conditional clauses:

```
declare @guess int
declare @store int
select @guess = 32
select @store = @guess
execute mathtutor 5, 6, @result = @guess output
select Your_answer = @store, Right_answer = @guess
if @guess = @store
    print "Right-o"
else
    print "Wrong, wrong, wrong!"
(1 row affected)
(1 row affected)
(return status = 0)

Return parameters:

@result
-----------
         30

 Your_answer Right_answer
 ----------- ------------
          32           30

(1 row affected)
Wrong, wrong, wrong!
```

Here is a stored procedure that checks whether new book sales would cause an author's royalty percentage to change. The *@pc* parameter is defined as an *output* parameter:

```
create proc roy_check @title tid, @newsales int,
        @pc int output
as
declare @newtotal int
select @newtotal = (select titles.total_sales +
@newsales
                  from titles where title_id =
@title)
select @pc = royalty from  roysched
   where @newtotal  >= roysched.lorange and
         @newtotal < roysched.hirange
   and roysched.title_id = @title
```

The following SQL batch calls the **roy_check** procedure, after assigning a value to the *percent* variable. The return parameters are printed before the next statement in the batch is executed:

```
declare @percent int
select @percent = 10
execute roy_check "BU1032", 1050, @pc = @percent
output
select Percent = @percent
go
(1 row affected)
(return status = 0)

Return parameters:

@pc
-----------
         12
 Percent
 -----------
          12

(1 row affected)
```

The following stored procedure calls the **roy_check** procedure, and uses the return value for *percent* in a conditional clause:

```
create proc newsales @title tid, @newsales int
as
declare @percent int
declare @stor_pc int
select @percent = (select royalty from roysched,
titles
        where roysched.title_id = @title
        and total_sales >= roysched.lorange
        and total_sales < roysched.hirange
        and roysched.title_id=titles.title_id)
select @stor_pc = @percent
execute roy_check @title, @newsales, @pc = @percent
  output
if
  @stor_pc != @percent
begin
  print "Royalty is changed"
  select Percent = @percent
end
else
  print "Royalty is the same"
```

If you execute this stored procedure with the same parameters used in the earlier batch, you see these results:

```
execute newsales "BU1032", 1050
```

```
Royalty is changed
Percent
-----------
         12

(1 row affected, return status = 0)
```

In the two preceding examples that call **roy_check**, *@pc* is the name of the parameter that is passed to **roy_check**, and *@percent* is the variable containing the output. When the **newsales** stored procedure executes **roy_check**, the value returned in *@percent* may change depending on the other parameters that are passed. If you want to compare the returned value of *percent* with the initial value of *@pc*, you must store the initial value in another variable. The preceding example saved the value in *stor_pc*.

Passing Values in Parameters

The values passed in the parameters must be passed in the form:

@parameter = @variable

You cannot pass constants; there must be a variable name to "receive" the return value. The parameters can be of any SQL Server datatype except *text* and *image*.

➤ ***Note***

If the stored procedure requires several parameters, either pass the return value parameter last in the **execute** statement, or pass all subsequent parameters in the form "@parameter = value".

The *output* Keyword

The **output** keyword can be abbreviated to **out**, just as **execute** can be shortened to **exec**.

A stored procedure can return several values; each must be defined as an **output** variable in the stored procedure and in the calling statements:

```
exec myproc @a = @myvara out, @b = @myvarb out
```

If you specify **output** while you're executing a procedure, and the parameter is not defined using *output* in the stored procedure, you will get an error message. It is not an error to call a procedure that includes return value specifications without requesting the return

values with *output*. However, you will not get the return values. The stored procedure writer has control over what information users can access, and users have control over their variables.

Rules Associated with Stored Procedures

Some additional rules for creating stored procedures are as follows:

- **create procedure** statements cannot be combined with other statements in a single batch.
- The **create procedure** definition itself can include any number and kind of SQL statements, with the exception of **use** and these **create** statements:

  create view
  create default
  create rule
  create trigger
  create procedure

- Other database objects can be created within a procedure. You can reference an object you created in the same procedure, as long as it is created before it is referenced. The **create** statement for the object must come first in the actual order of the statements within the procedure.
- Within a stored procedure, you cannot create an object, drop it, and then create a new object with the same name.
- SQL Server actually creates objects defined in a stored procedure when the procedure is executed, not when it is compiled.
- If you execute a procedure that calls another procedure, the called procedure can access objects created by the first procedure.
- You can reference temporary tables within a procedure.
- If you create a temporary table inside a procedure, the temporary table exists only for the purposes of the procedure—it disappears when you exit the procedure.
- The maximum number of parameters in a stored procedure is 255.
- The maximum number of local and global variables in a procedure is limited only by available memory.

Qualifying Names Inside Procedures

Inside a stored procedure, object names used with certain commands must be **qualified** with the object owner's name if other users are to make use of the stored procedure. These commands are: **alter table**, **create table**, **drop table**, **truncate table**, **create index**, **drop index**, **update statistics**, **dbcc**. Object names used with other statements, like **select** or **insert**, inside a stored procedure need not be qualified because the names are resolved when the procedure is compiled.

For example, user "mary", who owns table *marytab*, should qualify the name of her table when it is used with one of these commands if she wants other users to be able to execute the procedure in which the table is used:

```
create procedure p1
as
create index marytab_ind
on mary.marytab(col1)
```

The reason for this rule is that object names are resolved when the procedure is run. If *marytab* is not qualified, and user "john" tries to execute the procedure, SQL Server looks for a table called *marytab* owned by John. The preceding example shows the correct usage. It tells SQL Server to look for a table called *marytab* owned by Mary.

Dropping Stored Procedures

Procedures are removed with the **drop procedure** command. Its syntax is:

```
drop procedure [owner.]procedure_name
     [, [owner.]procedure_name]...
```

If a stored procedure that was dropped is called by another stored procedure, SQL Server displays an error message. However, if a new procedure of the same name is defined to replace the one that was dropped, other procedures that reference it can call it successfully.

A procedure group, that is, more than one procedure with the same name but different *number* suffixes, can be dropped with a single **drop procedure** statement. Once procedures have been grouped, procedures within the group cannot be dropped individually.

Renaming Stored Procedures

You can rename a stored procedure with the system procedure **sp_rename**. Here is its syntax:

```
sp_rename objname, newname
```

For example, to rename *showall* to *countall*:

```
sp_rename showall, countall
```

Of course, the new name must follow the rules for identifiers. You can change the name only of stored procedures that you own. The Database Owner can change the name of any user's stored procedure. The stored procedure must be in the current database.

Renaming Objects Referenced by Procedures

You must drop and re-create a procedure if you rename any of the objects it references. A stored procedure that references a table or view whose name has been changed may seem to work fine for a while. In fact, it only works until SQL Server recompiles it. Recompilation takes place for many reasons and without notification to the user.

Use **sp_depends** to get a report of the objects referenced by a procedure.

Using Stored Procedures As Security Mechanisms

You can use stored procedures as security mechanisms to control access to information in tables and to control the ability to perform data modification. For example, you can deny other users permission to use the **select** command on a table that you own and create a stored procedure that allows them to see only certain rows or certain columns. You can also use stored procedures to limit **update**, **delete**, or **insert** statements.

The person who owns the stored procedure must own the table or view used in the procedure. Not even a System Administrator can create a stored procedure to perform operations on another user's tables, if the System Administrator has not been granted permission on those tables.

For information about granting and revoking permissions of stored procedures and other database objects, see the *Security Features User's Guide*.

System Procedures

The system procedures are provided for your convenience as:

- Shortcuts for retrieving information from the system tables
- Mechanisms for accomplishing database administration and other tasks that involve updating system tables

Most of the time, system tables are updated **only** through stored procedures. A System Administrator can allow direct updates of system tables by changing a configuration variable and issuing the **reconfigure with override** command. See the *System Administration Guide* for details.

The names of all system procedures begin with "sp_". They are created by the **installmaster** script in the *sybsystemprocs* database during SQL Server installation.

You can run system procedures from any database. If a system procedure is executed from a database other than the *sybsystemprocs* database, any references to system tables are mapped to the database from which the procedure is being run. For example, if the Database Owner of *pubs2* runs **sp_adduser** from *pubs2*, the new user is added to *pubs2..sysusers*.

When the parameter for a system procedure is an object name, and the object name is qualified by a database name or owner name, the entire name must be enclosed in single or double quotes.

Since system procedures are located in the *sybsystemprocs* database, their permissions are also set there. Some of the system procedures can be run only by Database Owners. These procedures make sure that the user executing the procedure is the owner of the database on which they are executed.

Other system procedures can be executed by any user who has been granted **execute** permission on them, but this permission must be granted in the *sybsystemprocs* database. This situation has two consequences:

- A user can have permission to execute a system procedure either in all databases or in none of them.
- The owner of a user database cannot directly control permissions on the system procedures within his or her own database.

See the *System Administration Guide* for details.

Security Administration

This category includes procedures for:

- Adding, dropping, and reporting on logins on SQL Server
- Adding, dropping, and reporting on users, groups, and aliases in a database
- Changing passwords and default databases
- Changing the owner of a database
- Adding, dropping and reporting on remote servers that can access this SQL Server
- Adding the names of users from remote servers who can access this SQL Server

The procedures in this category are: **sp_addlogin**, **sp_addalias**, **sp_addgroup**, **sp_adduser**, **sp_changedbowner**, **sp_changegroup**, **sp_droplogin**, **sp_dropalias**, **sp_dropgroup**, **sp_dropuser**, **sp_helpgroup**, **sp_helprotect**, **sp_helpuser**, **sp_password**.

Remote Servers

This category includes procedures for:

- Adding, dropping and reporting on remote servers that can access this SQL Server
- Adding the names of users from remote servers who can access this SQL Server

The procedures in the category are: **sp_addremotelogin**, **sp_addserver**, **sp_dropremotelogin**, **sp_dropserver**, **sp_helpremotelogin**, **sp_helpserver**, **sp_remoteoption**, **sp_serveroption**.

Data Definition and Database Objects

This category includes procedures for:

- Binding and unbinding rules and defaults
- Adding, dropping, and reporting on primary keys, foreign keys, and common keys
- Adding, dropping, and reporting on user-defined datatypes
- Renaming database objects and user-defined datatypes
- Re-optimizing stored procedures and triggers

- Reporting on database objects, user-defined datatypes, dependencies among database objects, databases, indexes, and space used by tables and indexes

The procedures in this category are: **sp_bindefault**, **sp_bindrule**, **sp_unbindefault**, **sp_unbindrule**, **sp_foreignkey**, **sp_primarykey**, **sp_commonkey**, **sp_dropkey**, **sp_depends**, **sp_addtype**, **sp_droptype**, **sp_rename**, **sp_spaceused**, **sp_help**, **sp_helpdb**, **sp_helpindex**, **sp_helpjoins**, **sp_helpkey**, **sp_helptext**, **sp_indsuspect**, **sp_recompile**.

User-Defined Messages

This category includes procedures for:

- Adding user-defined messages to the *sysusermessages* table in a user database
- Dropping user-defined messages from *sysusermessages*
- Retrieving messages from either *sysusermessages* or *sysmessages* in the *master* database for use in **print** and **raiserror** statements

The procedures in this category are: **sp_addmessage**, **sp_dropmessage**, and **sp_getmessage**.

System Administration

This category includes procedures for:

- Adding, dropping, and reporting on database and dump devices
- Reporting on locks, the database options that are set, and the users currently running processes
- Changing and reporting on configuration variables
- Monitoring SQL Server activity

The procedures in this category are: **sp_addumpdevice**, **sp_dropdevice**, **sp_helpdevice**, **sp_helpsort sp_logdevice**, **sp_dboption**, **sp_diskdefault**, **sp_configure**, **sp_monitor**, **sp_lock**, **sp_who**.

More information about the system procedures that accomplish these administrative tasks is given in the *System Administration Guide*. For complete information about the system procedures, see the *SQL Server Reference Manual*.

Getting Information About Stored Procedures

Several system procedures provide information from the system tables about stored procedures.

sp_help

You can get a report on a stored procedure with the system procedure **sp_help**. For example, you can get information on the stored procedure **byroyalty**, which is part of the *pubs2* database, like this:

```
sp_help byroyalty
Name       Owner   type                Created_on
--------   ------  ----------------    -------------------
byroyalty  dbo     stored procedure    Feb  9 1987  3:56PM

Data_located_on_segment         When_created
---------------------------     --------------------
Parameter_name Type    Length   Param_order
-------------- ------  ------   -----------
@percentage    int          4           1

(return status = 0)
```

You can get help on a system procedure by executing **sp_help** when using the *sybsystemprocs* database.

sp_helptext

To display the text of the **create procedure** statement, execute the system procedure **sp_helptext**:

```
sp_helptext byroyalty
```

```
# Lines of Text
--------------
             1
(1 row affected)

text
--------------------------------------------------
create procedure byroyalty @percentage int
as
select au_id from titleauthor
where titleauthor.royaltyper = @percentage

(1 row affected, return status = 0)
```

You can view the text of a system procedure by executing **sp_helptext** when using the *sybsystemprocs* database.

sp_depends

The system procedure **sp_depends** lists all the stored procedures that reference the object you specify, or all the procedures that it is dependent upon. This command lists all the objects referenced by the user-created stored procedure *byroyalty*:

```
sp_depends byroyalty

Things the object references in the current
database.
object          type       updated      selected
--------------- ---------- ---------    --------
dbo.titleauthor  user table  no          no

(return status = 0)
```

The following statement uses **sp_depends** to list all the objects that reference the table *titleauthor*:

```
sp_depends titleauthor
```

```
Things inside the current database that reference
the object.

object          type
--------------  ------------------
dbo.titleview   view
dbo.reptq2      stored procedure
dbo.byroyalty   stored procedure

(return status = 0)
```

You must drop and recreate the procedure if any objects it references have been renamed.

System procedures are briefly discussed in “System Procedures” on page 14-25. For complete information about system procedures, see the *SQL Server Reference Manual*.

15 Triggers: Enforcing Referential Integrity

You can use triggers to enforce the referential integrity of data across the database. Triggers also allow you to "cascade" changes through related tables, to enforce column restrictions more complex than rules allow, and to compare the results of data modifications and take some resulting action.

This chapter discusses:

- An overview of triggers
- How to create and drop triggers
- How triggers can enforce referential integrity of data across the database
- The rules associated with triggers
- How to get information about triggers

What Are Triggers?

A trigger is a special kind of stored procedure that goes into effect when you insert, delete, or update data in a specified table. Triggers can help maintain the referential integrity of your data by maintaining consistency among logically related data in different tables. Referential integrity means that primary key values and corresponding foreign key values must match exactly.

The main advantage of triggers is that they are **automatic**. They work no matter what caused the data modification—a clerk's data entry or an application action. A trigger is specific to one or more of the data modification operations, **update**, **insert**, or **delete**. The trigger is executed once per SQL statement.

A trigger "fires" only after the data modification statement has completed and SQL Server has checked for any datatype, rule, or integrity constraint violations. The trigger and the statement which "fires" it are treated as a single transaction that can be rolled back from within the trigger. If a severe error is detected, the entire transaction rolls back.

In what situations are triggers most useful?

- Triggers can "cascade" changes through related tables in the database. For example, a delete trigger on the *title_id* column of the *titles* table could cause a corresponding deletion of matching

rows in other tables, using the *title_id* column as a unique key to locating rows in *titleauthor*, *sales*, and *roysched*.

- Triggers can disallow, or "roll back," changes that would violate referential integrity, canceling the attempted data modification transaction. Such a trigger might go into effect when you try to insert a foreign key that does not match its primary key. For example, you could create an insert trigger on *titleauthor* that rolled back an insert if the new *titleauthor.title_id* value did not match some value in *titles.title_id*.
- Triggers can enforce restrictions much more complex than those defined with rules. Unlike rules, triggers can reference columns or database objects. For example, a trigger can roll back updates that attempt to increase a book's price by more than 1 percent of the advance.
- Triggers can perform simple "what if" analyses. For example, a trigger can compare the state of a table before and after a data modification, and take actions based on that comparison.

This chapter summarizes trigger syntax, discusses how to use triggers, and provides examples of triggers. You may want to use these examples as templates for your own triggers. The final section of this chapter describes rules that concern the use of triggers, and explains system procedures that provide help with triggers.

➤ ***Note***

Except for the trigger named *deltitle*, the triggers discussed in this chapter are not included in the *pubs2* database shipped with your copy of SQL Server. To work with the examples shown in this chapter, create each trigger example by typing in the **create trigger** statement. Each new trigger for the same operation—**insert**, **update** or **delete**—on a table or column overwrites the previous one without warning, and old triggers are dropped automatically.

Comparing Triggers with Integrity Constraints

As an alternative to using triggers, you can use the referential integrity constraint of the **create table** statement to enforce referential integrity across tables in the database. However, referential integrity constraints differ from triggers in that they **cannot** perform the following tasks (as described above):

- "Cascade" changes through related tables in the database
- Enforce complex restrictions by referencing other columns or database objects
- Perform "what if" analyses

In addition, referential integrity constraints do not roll back the current transaction as a result of enforcing data integrity. With triggers, you can have it roll back or continue the transaction depending on how you handle referential integrity. For information about transactions, see Chapter 17, "Transactions: Maintaining Data Consistency and Recovery."

If your application requires one of the above tasks, you should use triggers. Otherwise, referential integrity constraints offer a simpler way to enforce data integrity. Note that SQL Server checks referential integrity constraints before any triggers, so a data modification statement that violates the constraint does not also fire the trigger. For more information about referential integrity constraints, see Chapter 7, "Creating Databases and Tables."

Creating Triggers

A trigger is a database object. When you create a trigger, you specify the table and the data modification commands that should "fire" or activate the trigger. Then you specify the action or actions the trigger is to take.

Here is a simple example. This trigger prints a message every time anyone tries to insert, delete, or update data in the *titles* table:

```
create trigger t1
on titles
for insert, update, delete
as
print "Now modify the titleauthor table the same
way."
```

create trigger Syntax

Here is the complete **create trigger** syntax:

```
create trigger [owner.]trigger_name
   on [owner.]table_name
   {for {insert , update , delete}
   as SQL_statements
```

Or, using the **if update** clause:

```
create trigger [owner.]trigger_name
   on [owner.]table_name
   for {insert , update}
   as
       [if update (column_name)
           [{and | or} update (column_name)]...]
           SQL_statements
       [if update (column_name)
           [{and | or} update (column_name)]...
           SQL_statements]...
```

The **create** clause creates the trigger and names it. A trigger's name must conform to the rules for identifiers.

The **on** clause gives the name of the table that activates the trigger. This table is sometimes called the **trigger table**.

A trigger is created in the current database, although it can reference objects in other databases. The owner name that qualifies the trigger name must be the same as the one on the table. No one except the table owner can create a trigger on a table. If the table owner is given with the table name in the **create trigger** clause or the **on** clause, it must also be specified in the other clause.

The **for** clause specifies which data modification commands on the trigger table activate the trigger. In the earlier example, an **insert**, **update** or **delete** to *titles* makes the message print.

The SQL statements specify **trigger conditions** and **trigger actions**. Trigger conditions specify additional criteria that determine whether the attempted **insert**, **delete**, or **update** will cause the trigger actions to be carried out. Multiple trigger actions in an **if** clause must be grouped with **begin** and **end**.

An **if update** clause tests for an insert or update to a specified column. For updates, the **if update** clause evauates to true when the column name is included in the **set** clause of an **update** statement, even if the update does not change the value of the column. **if update** is not used with **delete**. More than one column can be specified, and you can use more than one **if update** clause in a **create trigger** statement. Since you specify the table name in the **on** clause, do not use the table name in front of the column name with **if update**.

SQL Statements Not Allowed in Triggers

Since triggers execute as part of a transaction, the following statements are not allowed in a trigger:

- All **create** commands, including **create database**, **create table**, **create index**, **create procedure**, **create default**, **create rule**, **create trigger**, and **create view**
- All **drop** commands
- **alter table** and **alter database**
- **truncate table**
- **grant** and **revoke**
- **update statistics**
- **reconfigure**
- **load database** and **load transaction**
- **disk init**, **disk mirror**, **disk refit**, **disk reinit**, **disk remirror**, **disk unmirror**
- **select into**

Dropping Triggers

You can remove a trigger by dropping it, or by dropping the trigger table with which it is associated.

The **drop trigger** syntax is:

```
drop trigger [owner.]trigger_name
   [, [owner.]trigger_name]...
```

When a table is dropped any triggers associated with it are automatically dropped. **drop trigger** permission defaults to the trigger table owner and is not transferable.

Using Triggers to Maintain Referential Integrity

Triggers are used to maintain referential integrity, which assures that vital data in your database—such as the unique identifier for a given piece of data—remains accurate and can be used as the database changes. Referential integrity is coordinated through the use of primary and foreign keys.

The primary key is the column or combination of columns that uniquely identifies a row. It cannot be NULL and it must have a unique index. A table with a primary key is eligible for joins with foreign keys in other tables. The primary key table can be thought of as the **master table** in a **master-detail relationship**. There can be many such master-detail groups in a database.

You can use the **sp_primarykey** procedure to mark a primary key. This marks the key for use with **sp_helpjoins** and adds it to the *syskeys* table.

In the *pubs2* database, for example, the *title_id* column is the primary key of *titles*. It uniquely identifies the books in *titles*, and joins with *title_id* in *titleauthor*, *salesdetail*, and *roysched*. The *titles* table is the master table in relation to *titleauthor*, *salesdetail*, and *roysched*. The diagram in Chapter 1, "The pubs2 Database," of the *SQL Server Reference Supplement* shows these relationships.

The foreign key is a column or combination of columns whose values match the primary key. A foreign key doesn't have to be unique. They are often in a many-to-one relationship to a primary key. Foreign key values should be copies of the primary key values. That means no value in the foreign key should ever exist unless the same value exists in the primary key. A foreign key may be null; if any part of a composite foreign key is null, the entire foreign key must be null. Tables with foreign keys are often called **detail** or **dependent** tables to the master table.

You can use the **sp_foreignkey** procedure to mark foreign keys in your database. This flags them for use with **sp_helpjoins** and other procedures that reference the *syskeys* table.

The *title_id* columns in *titleauthor*, *salesdetail*, and *roysched* are foreign keys; these tables are detail tables.

How Triggers Work

Referential integrity triggers keep the values of foreign keys in line with those in primary keys. When a data modification affects a key column, triggers compare the new column values to related keys by using temporary work tables called **trigger test tables**. When you write your triggers, you base your comparisons on the data that is temporarily stored in the trigger test tables.

Testing Data Modifications Against the Trigger Test Tables

Two special tables are used in trigger statements: the *deleted* table and the *inserted* table. These are temporary tables used in trigger tests. Use these tables to test the effects of some data modification and to set conditions for trigger actions. You cannot directly alter the data in the trigger test tables, but you can use the tables in **select** statements to detect the effects of an **insert**, **update**, or **delete**.

- The *deleted* table stores copies of the affected rows during **delete** and **update** statements. During the execution of a **delete** or **update** statement, rows are removed from the trigger table and transferred to the *deleted* table. The *deleted* table and the trigger table ordinarily have no rows in common.
- The *inserted* table stores copies of the affected rows during **insert** and **update** statements. During an **insert** or an **update**, new rows are added to *inserted* and the trigger table at the same time. The rows in *inserted* are copies of the new rows in the trigger table.

An **update** is, effectively, a delete followed by an insert; the old rows are copied to the *deleted* table first; then the new rows are copied to the trigger table and to the *inserted* table. The following illustration shows the condition of the trigger test tables during an **insert**, a **delete**, and an **update**:

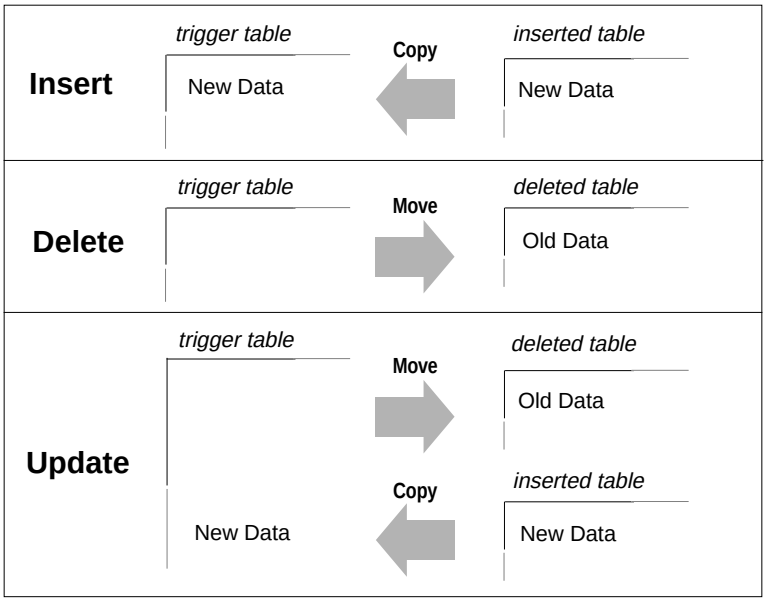


Figure 15-1: Trigger test tables during insert, delete, or update operation

When setting trigger conditions, use the trigger test tables that are appropriate for the data modification. While it is not an error to reference *deleted* while testing an **insert**, or *inserted* while testing a

delete, these trigger test tables will not contain any rows in these cases.

➤ ***Note***

A given trigger fires only once per query. If trigger actions depend on the number of rows a data modification affects, you should use tests, such as an examination of *@@rowcount*, for multirow data modifications, and take appropriate actions.

The following trigger examples will accommodate multirow data modifications where necessary. The *@@rowcount* variable, which stores the "number of rows affected" by the most recent data modification operation, tests for a multirow **insert**, **delete**, or **update**. If any other **select** statement precedes the test on *@@rowcount* within the trigger, you should use local variables to store the value for later examination. All Transact-SQL statements that do not return values reset *@@rowcount* to 0.

An Insert Trigger Example

When you insert a new foreign key row, you want to make sure the foreign key matches a primary key. The trigger should check for joins between the inserted row or rows and the rows in the primary key table, and then roll back any inserts of foreign keys that do not match a key in the primary key table. This example rolls back all changes caused by the **insert** statement; later examples show how to selectively reject certain data modifications.

On **insert**, new rows are added to the trigger table and to the *inserted* trigger test table. In order to see whether the new keys match some primary key, check for joins between *inserted* and the primary key table.

The following trigger compares the *title_id* values from the *inserted* table with those from the *titles* table. It assumes that you are making some entry for the foreign key and that you are not inserting a null value. If the join fails, the transaction is rolled back.

```
create trigger forinsertrig1
on salesdetail
for insert
as
if (select count(*)
    from titles, inserted
    where titles.title_id = inserted.title_id) !=
    @@rowcount
/* cancel the insert and print a message.*/
  begin
    rollback transaction
    print "No, a title_id does not exist in titles"
  end
/* Otherwise, allow it. */
else
  print "Added! All title_id's exist in titles."
```

In this example, *@@rowcount* refers to the number of rows added to the *salesdetail* table. This is also the number of rows added to the *inserted* table. The test of whether all the *title_id*'s added to *salesdetail* exist in the *titles* table is performed by joining *titles* and *inserted*. If the number of joined rows, which is detemined by the **select count(*)** query, differs from *@@rowcount*, then one or more of the inserts is incorrect, and the entire transaction is canceled.

This trigger prints one message if the insert is rolled back and another if it is accepted.

A Delete Trigger Example

When you delete a primary key row, you should delete corresponding foreign key rows in dependent tables. This preserves referential integrity by ensuring that detail rows are removed when their master row is deleted. If this were not done, you could end up with a database that had detail rows that could not be retrieved or identified. A trigger that performs a cascading **delete** is required.

Here is an example. When a **delete** statement on *titles* is executed, one or more rows leave the *titles* table and are added to *deleted*. A trigger can check the dependent tables—*titleauthor*, *salesdetail*, and *roysched*—to see if they have any rows with a *title_id* that matches the *title_id*s removed from *titles* and now stored in the *deleted* table. If the trigger finds any such rows, it removes them.

```
create trigger delcascadetrig
on titles
for delete
as
delete titleauthor
from titleauthor, deleted
where titleauthor.title_id = deleted.title_id
        /* Remove titleauthor rows
        ** that match deleted (titles) rows.*/

delete salesdetail
from salesdetail, deleted
where salesdetail.title_id = deleted.title_id
        /* Remove salesdetail rows
        ** that match deleted (titles) rows.*/

delete roysched
from roysched, deleted
where roysched.title_id = deleted.title_id
        /* Remove roysched rows
        ** that match deleted (titles) rows.*/
```

In actual practice, you may find that you want to keep some of the detail rows. This may be either for historical purposes (to check how many sales were made on discontinued titles while they were active) or because transactions on the detail rows are not yet complete. A well-written trigger should take these factors into consideration.

For example, the *deltitle* trigger supplied with the *pubs2* database **prevents** the deletion of a primary key if there are any detail rows for that key in the *salesdetail* table. This trigger preserves the ability to retrieve rows from *salesdetail*.

```
create trigger deltitle
on titles
for delete
as
if (select count(*)
    from deleted, salesdetail
    where salesdetail.title_id =
    deleted.title_id) > 0
  begin
    rollback transaction
    print "You can't delete a title with sales."
  end
```

In this trigger, the row or rows deleted from *titles* are tested by being joined with the *salesdetail* table. If any join is found, the transaction is canceled.

Update Trigger Examples

Since a primary key is the unique identifier for its row and for foreign key rows in other tables, an attempt to update a primary key should be taken very seriously. In this case, you want to protect referential integrity by rolling back the update unless specified conditions are met.

Generally speaking, it is best to prohibit any editing changes to a primary key, for example, by revoking all permissions on that column. But if you did want to prohibit updates only under certain circumstances, use a trigger.

This trigger prevents updates to *titles.title_id* on the weekend. The **if update** clause in *stopupdatetrig* allows you to focus on a particular column, *titles.title_id*. Modifications to the data in that column cause the trigger to go into action. Changes to the data in other columns do not. When this trigger detects an update that violates the trigger conditions, it cancels the update and prints a message. If you would like to test this one, substitute the current day of the week for "Saturday" or "Sunday".

```
create trigger stopupdatetrig
on titles
for update
as
/* If an attempt is made to change titles.title_id
** on Saturday or Sunday, cancel the update.
*/
if update (title_id)
    and datename(dw, getdate())
    in ("Saturday", "Sunday")
  begin
    rollback transaction
    print "We don't allow changes to"
    print "primary keys on the weekend!"
  end
```

You can also specify multiple trigger actions on more than one column using **if update**. The following example modifies *stopupdatetrig* to include additional trigger actions for updates to *titles.price* or *titles.advance*. Other than preventing updates to the primary key on weekends, it also prevents updates to the price or advance of a title

unless the total revenue amount for that title surpasses its advance amount. You can use the same trigger name since the modified trigger replaces the old trigger when you create it again.

```
create trigger stopupdatetrig
on titles
for update
as
if update (title_id)
  and datename(dw, getdate())
  in ("Saturday", "Sunday")
  begin
    rollback transaction
    print "We don't allow changes to"
    print "primary keys on the weekend!"
  end
if update (price) or update (advance)
  if (select count(*) from inserted
    where (inserted.price * inserted.total_sales)
    < inserted.advance) > 0
    begin
      rollback transaction
      print "We don't allow changes to price or"
      print "advance for a title until its total"
      print "revenue exceeds its latest advance."
    end
```

Updating a Foreign Key

A change or update to a foreign key by itself is probably an error. A foreign key is just a copy of the primary key. The two should never be independent. If for some reason you want to allow updates of a foreign key, you might want to protect integrity by creating a trigger that checks updates against the *master* table and rolls them back if they don't match the primary key.

In the following example, the trigger tests for two possible sources of failure: either the *title_id* is not in the *salesdetail* table to begin with, or it's not in the *titles* table.

This example uses nested **if...else** statements. The first **if** statement is true when the value in the **where** clause of the **update** statement does not match any existing value in *salesdetail*, that is, the *inserted* table will not contain any rows, and the select returns a null value. If this test is passed, the next **if** statement ascertains whether the new row or rows in the *inserted* table join with any *title_id* in the *titles* table. If any row does not join, the transaction is rolled back, and an error

message is printed. If the join succeeds, a different message is printed.

```
create trigger forupdatetrig
on salesdetail
for update
as
declare @row int
/* save value of rowcount */
select @row = @@rowcount
if update (title_id)
  begin
    if (select distinct inserted.title_id
        from inserted) is null
      begin
        rollback transaction
        print "No!  Old title_id must be in
              salesdetail"
      end
    else
      if (select count(*)
          from titles, inserted
          where titles.title_id =
          inserted.title_id) != @row
        begin
          rollback transaction
          print "No! New title_id not in titles"
        end
      else
        print "salesdetail table updated"
  end
```

Multirow Considerations

Multirow considerations are particularly important when the function of a trigger is to automatically recalculate summary values, that is, ongoing tallies.

Triggers used to maintain summary values should contain **group by** clauses, or subqueries which perform implicit grouping, to create summary values when more than one row is being inserted, updated, or deleted. Since a **group by** clause imposes extra overhead, the following examples are written to test whether the value of *@@rowcount* is equal to one, meaning that only one row in the trigger table was affected. If *@@rowcount is* equal to one, the trigger actions take effect without a **group by** clause.

This insert trigger updates the *total_sales* column in the *titles* table every time a new *salesdetail* row is added. It goes into effect whenever you record a sale by adding a row to the *salesdetail* table. It updates the *total_sales* column in the *titles* table so that *total_sales* is equal to its previous value plus the value added to *salesdetail.qty*. This keeps the totals up to date for inserts into *salesdetail.qty*.

```
create trigger intrig
on salesdetail
for insert as
    /* check value of @@rowcount */
if @@rowcount = 1
    update titles
      set total_sales = total_sales + qty
      from inserted
      where titles.title_id = inserted.title_id
else
    /* when rowcount is greater than 1,
       use a group by clause */
    update titles
      set total_sales =
        total_sales + (select sum(qty)
      from inserted
      group by inserted.title_id
      having titles.title_id = inserted.title_id)
```

The next example is a delete trigger that updates the *total_sales* column in the *titles* table every time one or more *salesdetail* rows are deleted.

```
create trigger deltrig
on salesdetail
for delete
as
    /* check value of @@rowcount */
if @@rowcount = 1
    update titles
      set total_sales = total_sales - qty
      from deleted
      where titles.title_id = deleted.title_id
else
    /* when rowcount is greater than 1,
       use a group by clause */
    update titles
      set total_sales =
        total_sales - (select sum(qty)
      from deleted
      group by deleted.title_id
      having titles.title_id = deleted.title_id)
```

This trigger goes into effect whenever a row is deleted from the *salesdetail* table. It updates the *total_sales* column in the *titles* table so that *total_sales* is equal to its previous value minus the value subtracted from *salesdetail.qty*.

The following update trigger updates the *total_sales* column in the *titles* table every time the *qty* field in a *salesdetail* row is updated. Recall that an update is an insert followed by a delete. This trigger references both the *inserted* and the *deleted* trigger test tables.

```
create trigger updtrig
on salesdetail
for update
as
if update (qty)
begin
    /* check value of @@rowcount */
    if @@rowcount = 1
        update titles
          set total_sales = total_sales +
            inserted.qty - deleted.qty
          from inserted, deleted
          where titles.title_id = inserted.title_id
          and inserted.title_id = deleted.title_id
    else
    /* when rowcount is greater than 1,
       use a group by clause */
    begin
        update titles
          set total_sales = total_sales +
            (select sum(qty)
                from inserted
                group by inserted.title_id
                having titles.title_id =
                inserted.title_id)
        update titles
          set total_sales = total_sales -
            (select sum(qty)
                from deleted
                group by deleted.title_id
                having titles.title_id =
                deleted.title_id)
    end
end
```

A Conditional Insert Trigger

The triggers examined so far have looked at each data modification statement as a whole. If one row of a four-row insert was unacceptable, the whole insert was unacceptable and the transaction was rolled back.

However, you do not have to roll back all data modifications simply because some of them are unacceptable. Using a correlated subquery in a trigger can force the trigger to examine the modified rows one by one. See Chapter 5, "Subqueries: Using Queries Within Other Queries," for more information on correlated subqueries. The trigger can then take different actions on different rows.

The trigger example that follows assumes the existence of a table called *newsales*. Here is its **create** statement:

```
create table newsales
(stor_id    char(4)     not null,
ord_num     varchar(20) not null,
title_id    tid         not null,
qty         smallint    not null,
discount    float       not null)
```

You should insert four rows in the *newsales* table, in order to test the conditional trigger. Two of the *newsales* rows have *title_ids* that do not match any of those already in the *titles* table. Here is the data:

```
newsales

stor_id ord_num     title_id    qty     discount
------- ----------  ---------   ----    ---------
7066    BA27619     PS1372        75    40.000000
7066    BA27619     BU7832       100    40.000000
7067    NB-1.242    PSxxxx        50    40.000000
7131    Asoap433    PSyyyy        50    40.000000
```

When you insert data from *newsales* into *salesdetail*, the statement looks like this:

```
insert salesdetail
select * from newsales
```

What if you want to examine each of the records you are trying to insert? The trigger *conditionalinsert* analyzes the insert row by row, and deletes the rows that do not have a *title_id* in *titles*. Here's how:

```
create trigger conditionalinsert
on salesdetail
for insert as
if
(select count(*) from titles, inserted
where titles.title_id = inserted.title_id)
    != @@rowcount
begin
  delete salesdetail from salesdetail, inserted
    where salesdetail.title_id = inserted.title_id
    and inserted.title_id not in
    (select title_id from titles)
  print "Only records with matching title_ids
    added."
end
```

The trigger test is the same as the one in the *intrig* example, but the transaction is not rolled back. Instead, the trigger deletes the unwanted rows. This ability to delete rows that have just been inserted relies on the order in which processing occurs when triggers are fired. First, the rows are inserted into the table and the *inserted* table; then, the trigger fires.

Rolling Back Triggers

You can roll back triggers using either the **rollback trigger** statement or the **rollback transaction** statement (if the trigger is fired as part of a transaction). However, **rollback trigger** rolls back only the effect of the trigger and the statement which caused the trigger to fire. **rollback transaction** rolls back the entire transaction. For example:

```
begin tran
insert into publishers (pub_id) values ('9999')
insert into publishers (pub_id) values ('9998')
commit tran
```

If the second **insert** statement causes a trigger on *publishers* to issue a **rollback trigger**, only that **insert** is affected; the first **insert** is not rolled back. If that trigger issues a **rollback transaction** instead, both **insert** statements are rolled back as part of the transaction.

The following is the syntax for **rollback trigger:**

```
rollback trigger
   [with raiserror_statement]
```

The syntax for **rollback transaction** is described in Chapter 17, "Transactions: Maintaining Data Consistency and Recovery."

The *raiserror_statement* specifies a **raiserror** statement which prints a user-defined error message and sets a system flag to record that an error condition has occurred. This provides the ability to raise an error to the client when the **rollback trigger** is executed, so that the transaction state in the error reflects the rollback. For example:

```
rollback trigger with raiserror 25002
    "title_id does not exist in titles table."
```

For more information about **raiserror**, see Chapter 13, "Using Batches and Control-of-Flow Language."

When the **rollback trigger** is executed, SQL Server aborts the currently executing command and halts execution of the rest of the trigger. If the trigger that issues the **rollback trigger** is nested within other triggers, SQL Server rolls back all work done in these triggers up to and including the update which caused the first trigger to fire.

The following example of an insert trigger performs a similar task to the trigger *forinsertrig1* described on page 15-9. However, this trigger uses a **rollback trigger** instead of a **rollback transaction** to raise an error when it rolls back the insertion but not the transaction.

```
create trigger forinsertrig2
on salesdetail
for insert
as
if (select count(*) from titles, inserted
    where titles.title_id = inserted.title_id) !=
    @@rowcount
   rollback trigger with raiserror 25003
     "Trigger rollback: salesdetail row not added
     because a title_id does not exist in titles."
```

When triggers that include **rollback transaction** statements are executed from a batch, they abort the entire batch. In the following example, if the **insert** statement fires a trigger that includes a **rollback transaction** (such as *forinsertrig1*), the **delete** statement will not be executed, since the batch will be aborted:

```
insert salesdetail values ("7777", "JR123",
    "PS9999", 75, 40)
delete salesdetail where stor_id = "7067"
```

If triggers that include **rollback transaction** statements are fired from within a **user-defined transaction**, the **rollback transaction** rolls back the entire batch. In the following example, if the **insert** statement fires a trigger that includes a **rollback transaction**, the **update** statement will also be rolled back:

```
begin tran
update stores set payterms = "Net 30"
    where stor_id = "8042"
insert salesdetail values ("7777", "JR123",
    "PS9999", 75, 40)
```

See Chapter 17, "Transactions: Maintaining Data Consistency and Recovery," for information on user-defined transactions.

SQL Server ignores a **rollback trigger** executed outside of a trigger and does not issue a **raiserror** associated with the statement. However, a **rollback trigger** executed outside of a trigger but inside a transaction generates an error which causes SQL Server to roll back the transaction and abort the current statement batch.

Nesting Triggers

Triggers can nest to a depth of 16 levels. Nesting is enabled at installation. A System Administrator can turn trigger nesting on and off with **sp_configure**. To disable nesting:

```
sp_configure "allow nested triggers", 0
```

If nested triggers are enabled, a trigger that changes a table on which there is another trigger fires the second trigger, which can in turn fire a third trigger, and so forth. If any trigger in the chain sets off an infinite loop, the nesting level is exceeded and the trigger aborts. You can use nested triggers to perform useful housekeeping functions such as storing a backup copy of rows affected by a previous trigger.

For example, you can create a trigger on *titleauthor* that saves a backup copy of *titleauthor* rows that the *delcascadetrig* trigger deleted. With the *delcascadetrig* trigger in effect, deleting the *title_id* "PS2091" from *titles* also deletes the corresponding row or rows from *titleauthor*. To save the data, you can create a **delete** trigger on *titleauthor* that saves the deleted data in another table, *del_save*:

```
create trigger savedel
on titleauthor
for delete
as
insert del_save
select * from deleted
```

It is not a good idea to use nested triggers in an order-dependent sequence. Use separate triggers to cascade data modifications, as in the earlier example of *delcascadetrig*.

➤ ***Note***

When triggers are put into a transaction, a failure at any level of a set of nested triggers (including the error message that the nesting level has been exceeded) cancels the entire transaction. All data modifications are rolled back. Supply your triggers with **print** or **raiserror** statements in order to determine where the failure occurred.

A **rollback transaction** in a trigger at any nesting level rolls back the effects of each trigger and cancels the entire transaction. A **rollback trigger** affects only the nested triggers and the data modification statement that caused the initial trigger to fire.

Trigger Self-Recursion

By default, a trigger does not call itself recursively. That is, an update trigger does not call itself in response to a second update to the same table within the trigger. If an update trigger on one column of a table results in an update to another column, the update trigger fires only once, rather than repeatedly. However, you can turn on the **self_recursion** option of the **set** command to allow triggers to call themselves recursively. The **allow nested triggers** configuration variable must also be enabled for self-recursion to occur.

The **self_recursion** setting remains in effect only for the duration of a current client session. If the option is set as part of a trigger, its effect is limited by the scope of the trigger that sets it. If the trigger that sets **self_recursion on** returns or causes another trigger to fire, this option reverts to **off**. Once a trigger turns on the **self_recursion** option, it can repeatedly loop if its own actions cause itself to fire again, but it cannot exceed the limit of 16 nesting levels.

For example, assume the following *new_budget* table exists in *pubs2*:

```
select * from new_budget

unit            parent_unit     budget
--------------- --------------- -------
one_department  one_division    10
one_division    company_wide    100
company_wide    NULL            1000

(3 rows affected)
```

You can create a trigger that recursively updates *new_budget* whenever its *budget* column is changed as follows:

```
create trigger budget_change
on new_budget
for update as
if exists (select * from inserted
            where parent_unit is not null)
begin
    set self_recursion on
    update new_budget
    set new_budget.budget = new_budget.budget +
        inserted.budget - deleted.budget
    from inserted, deleted, new_budget
    where new_budget.unit = inserted.parent_unit
        and new_budget.unit = deleted.parent_unit
end
```

If a user updates *new_budget.budget* by increasing the budget of unit *one_department* by 3, SQL Server behaves as follows (assuming nested triggers are enabled):

1. Increasing *one_department* from 10 to 13 fires the *budget_change* trigger.
2. The trigger updates the budget of the parent of *one_department* (in this case *one_division*) from 100 to 103, which fires the trigger again.
3. The trigger updates the parent of *one_division* (in this case *company_wide*) from 1000 to 1003, which causes the trigger to fire for the third time.
4. The trigger attempts to update the parent of *company_wide*, but since none exists (NULL) the last **update** never occurs and the trigger is not fired, ending the self recursion. You can query *new_budget* to see the final results as follows:

```
select * from new_budget

unit            parent_unit     budget
--------------- --------------- -------
one_department  one_division    13
one_division    company_wide    103
company_wide    NULL            1003

(3 rows affected)
```

A trigger can also be recursively executed in other ways. A trigger calls a stored procedure that performs actions that causes the trigger to fire again (it is reactivated only if nested triggers are enabled). Unless there are conditions within the trigger that limit the number of recursions, this causes a nesting-level overflow.

For example, if an update trigger calls a stored procedure that performs an update, the trigger and stored procedure execute exactly once if **nested triggers** is set off. If **nested triggers** is set on, and the number of updates is not limited to less than 16 by some condition in the trigger or procedure, this loop will continue until it exceeds the 16-level maximum nesting value.

Rules Associated with Triggers

Apart from anticipating the effects of a multirow data modification, trigger rollbacks, and trigger nesting, there are some other factors to consider when writing triggers.

Triggers and Permissions

A trigger is defined on a particular table. Only the owner of the table has **create trigger** and **drop trigger** permissions for the table. These permissions cannot be transferred to others.

SQL Server will accept a trigger definition that attempts actions for which you do not have permission. The existence of such a trigger aborts any future attempt to modify the trigger table, since the trigger will fire and fail due to the incorrect permissions. The transaction will be canceled. You must rectify the permissions situation or drop the trigger.

For example, Jose owns *salesdetail* and creates a trigger on it. The trigger is supposed to update *titles.total_sales* when *salesdetail.qty* is updated. However, Mary is the owner of *titles*, and has not granted Jose permission on *titles*. When Jose tries to update *salesdetail*, SQL Server detects the trigger and Jose's lack of permissions on *titles*, and rolls back the update transaction. Jose must either get update permission on *titles.total_sales* from Mary, or drop the trigger on *salesdetail*.

Trigger Restrictions

The following describes some of the limitations or restrictions imposed on triggers by SQL Server:

- A table can have a maximum of three triggers: one update trigger, one insert trigger, and one delete trigger.
- Each trigger can apply to only one table. However, a single trigger may apply to all three user actions: **update**, **insert**, and **delete**.

- You cannot create a trigger on a view or on a temporary table, though triggers can reference views or temporary tables.
- Although a **truncate table** statement is, in effect, like a **delete** without a **where** clause because it removes all rows, it cannot "fire" a trigger because individual row deletions are not logged.
- Triggers are not allowed on system tables. Although no error message appears if you create a trigger on a system table, the trigger will not be used.

Implicit and Explicit Null Values

if update(*column_name*) is true for an **insert** statement whenever the column is assigned a value in the select list or in the **values** clause. An explicit NULL or a default assigns a value to a column, and thus activates the trigger. An implicit NULL, however, does not.

These examples clarify the situation:

```
create table junk
(a int null,
 b int not null)

create trigger junktrig
on junk
for insert
as
if update(a) and update(b)
        print "FIRING"

    /*"if update" is true for both columns.
      The trigger is activated.*/
insert junk (a, b) values (1, 2)

    /*"if update" is true for both columns.
      The trigger is activated.*/
insert junk values (1, 2)

    /*Explicit NULL:
      "if update" is true for both columns.
      The trigger is activated.*/
insert junk values (NULL, 2)
```

```
    /* If default exists on column a,
      "if update" is true for either column.
      The trigger is activated.*/
insert junk (b) values (2)

    /* If no default exists on column a,
      "if update" is not true for column a.
      The trigger is not activated.*/
insert junk (b)values (2)
```

The same results would be produced using only the clause:

```
if update(a)
```

To create a trigger that disallows the insertion of implicit nulls, you can use:

```
if update(a) or update(b)
```

SQL statements in the trigger can then test to see if *a* or *b* is null.

Triggers and Performance

In performance terms, trigger overhead is usually very low. The time involved in running a trigger is spent mostly in referencing other tables, which may be either in memory or on the database device.

The *deleted* and *inserted* trigger test tables are always in active memory. The location of other tables referenced by the trigger determines the amount of time the operation takes.

set Commands in Triggers

You can use the **set** command inside a trigger. The **set** option you invoke remains in effect during the execution of the trigger, and then reverts to its former setting.

Renaming and Triggers

If you change the name of an object referenced by a trigger, you must drop the trigger and re-create it so that its text reflects the new name of the object it references. Use **sp_depends** to obtain a report of the objects that a trigger references. The safest course of action is not to rename any tables or views that are referenced by a trigger.

Getting Information About Triggers

As database objects, triggers are listed in *sysobjects* by name. The *type* column of *sysobjects* identifies triggers with the abbreviation "TR." This query finds the triggers that exist in a database:

```
select *
from sysobjects
where type = "TR"
```

The **create trigger** statement for each trigger is stored in *syscomments*. You can display the trigger definition with the system procedure **sp_helptext**.

Execution plans for triggers are stored in *sysprocedures*. Several system procedures provide information from the system tables about triggers.

sp_help

You can get a report on a trigger with the system procedure **sp_help**. For example, you can get information on *deltitle* like this:

```
sp_help deltitle
Name        Owner   Type
----------- ------- -----------
deltitle    dbo     trigger

Data_located_on_segment  When_created
-----------------------  -----------------
not applicable           Feb 9 1987  3:56PM

(return status = 0)
```

sp_helptext

To display the text of the **create trigger** statement, execute the system procedure **sp_helptext**:

```
sp_helptext deltitle
# Lines of Text
---------------
              1
```

```
text
--------------------------------------------

create trigger deltitle
on titles
for delete
as
if (select count(*) from deleted, salesdetail
where salesdetail.title_id = deleted.title_id) >0
begin
rollback transaction
print "You can't delete a title with sales."
end
```

sp_depends

The system procedure **sp_depends** lists all the triggers that reference the object or all the tables or views that the trigger affects. This example shows how to use **sp_depends** to get a list of all the objects referenced by the trigger *deltitle*:

```
sp_depends deltitle

Things the object references in the current
database.

object              type       updated  selected
----------------    ----------  ------- --------
dbo.salesdetail     user table  no      no
dbo.titles          user table  no      no

(return status = 0)
```

This statement lists all the objects that reference the *salesdetail* table:

```
sp_depends salesdetail
Things inside the current database that reference
the object.

object                        type
----------------------------  ----------------
dbo.deltitle                  trigger
dbo.history_proc              stored procedure
dbo.insert_salesdetail_proc   stored procedure
dbo.totalsales_trig           trigger

(return status = 0)
```

16 Cursors: Accessing Data Row by Row

A **select** statement returns zero or more rows. If your **select** statement returns several rows, you can manipulate each row individually using cursors.

This chapter discusses:

- A general overview of cursors
- How to declare and open cursors
- How to get data using cursors
- How to update or delete data using cursors
- How to close and deallocate cursors
- An example of using cursors
- How locking affects cursors
- How to get information about cursors

What Are Cursors?

A **cursor** is a symbolic name that is associated with a Transact-SQL **select** statement through a declaration statement. It consists of the following parts:

- **cursor result set** – the set (table) of rows resulting from the execution of a query associated with the cursor
- **cursor position** – a pointer to one row within the cursor result set

The cursor position indicates the current row of the cursor. You can explicitly modify or delete that row using **delete** or **update** statements with a clause naming the cursor. You change the current cursor position through an operation called a **fetch**. A fetch moves the current cursor position one or more rows down the cursor result set.

A cursor behaves much like a file pointer to a series of file records, where the cursor acts as a pointer to the query results. However, cursors only support forward (or sequential) movement through the query results. Once you fetch several rows, you cannot backtrack through the cursor result set to access them again. This process allows you to traverse the query results row by row.

After you declare the cursor, it is in one of two states:

- Closed – The cursor result set does not exist, so you cannot read information from it. Cursors are initially in this state. You must explicitly open the cursor before you can use it. Once it is opened, you can explicitly close it when you are finished. SQL Server can implicitly close a cursor for several reasons, which are described later in this chapter.
- Open – The rows within the cursor result set are available for reading or updating.

You can close a cursor and then reopen it. Reopening a cursor re-creates the cursor result set and positions the cursor right before the first row. This allows you to process through a cursor result set as many times as needed. You can close the cursor at any time; you do not have go through the entire result set.

All cursor operations, like fetching or updating the row, are accomplished in reference to the current cursor position. Updating a cursor row involves changing data in the row or deleting the row completely. You cannot use cursors to insert rows. All updates through a cursor affect the corresponding base tables included in the cursor result set.

How SQL Server Processes Cursors

When accessing data using cursors, SQL Server divides the process into the following operations:

- Declaring the cursor

  SQL Server creates the cursor structure and compiles the query defined for that cursor. It stores the compiled query plan but does not execute it.

- Opening the cursor

  SQL Server executes the query plan. It scans the base tables (as much as is needed, just like a normal **select**) and creates the cursor result set. It prepares any temporary tables generated by the query and allocates resources (such as memory) to support the cursor structure. It also positions the cursor before the first row of the cursor result set.

- Fetching from the cursor

  SQL Server moves the cursor position one or more rows down the cursor result set. It retrieves the data from each row of the result set and stores the current position, allowing further fetches until it reaches the end of the result set.

- Updating or deleting through the cursor

  SQL Server updates or deletes the data in the cursor result set (and corresponding base tables that derived the data) at the current cursor position after a fetch. This operation is optional.

- Closing the cursor

  SQL Server closes the cursor result set, removes any remaining temporary tables, and releases the server resources held for the cursor structure. However, it keeps the query plan for the cursor so that it can be opened again.

- Deallocating the cursor

  SQL Server dumps the query plan from memory and eliminates all trace of the cursor structure. You must declare the cursor again before using it.

Declaring Cursors

You must declare a cursor before you can use it. The declaration specifies the query that defines the cursor result set. You can explicitly define a cursor as updatable or read-only by using the **for update** or **for read only** keywords. If you omit either one, SQL Server determines whether the cursor is updatable based on the type of query that defines the cursor result set. You cannot use the **update** or **delete** statements on the result set of a read-only cursor.

declare cursor Syntax

The syntax of the **declare cursor** statement is:

```
declare cursor_name cursor
   for select_statement
   [for {read only | update [of column_name_list]}]
```

The **declare cursor** statement must precede any **open** statement for that cursor. You cannot combine **declare cursor** with other statements in the same Transact-SQL batch, except when using a cursor in a stored procedure.

The *select_statement* is the query that defines the cursor result set for *cursor_name*. In general, *select_statement* may use the full syntax and semantics of a Transact-SQL **select** statement, including the **holdlock** keyword. However, it cannot contain a **compute**, **for browse**, or **into** clause.

For example, the following **declare cursor** statement defines a result set for the *authors_crsr* cursor that contains all authors that do not reside in California:

```
declare authors_crsr cursor
for select au_id, au_lname, au_fname
from authors
where state != 'CA'
```

The *select_statement* can contain references to Transact-SQL parameter names or local variables. However, the names can only reference parameters and local variables defined in a stored procedure that contains the **declare cursor** statement. If the cursor is used in a trigger, the *select_statement* can also reference the *inserted* and *deleted* temporary tables that are used in triggers. For information on using the **select** statement, see Chapter 2, "Queries: Selecting Data from a Table."

Cursor Scope

A cursor is defined by its **scope**, which determines the region in which the cursor is known. Once a cursor's scope no longer exists, its cursor name no longer exists. Cursor scopes are defined by the following regions:

- Session – This region starts when a client logs onto SQL Server and ends when it logs off. This region is distinct from regions defined by stored procedures or triggers.
- Stored procedure – This region starts when a stored procedure begins execution and ends when it completes execution. If a stored procedure calls another stored procedure, SQL Server starts a new region and treats it as a subregion of the first procedure.
- Trigger – This region starts when a trigger begins execution and ends when it completes execution.

A cursor name must be unique within a given scope. Since each scope is distinct, a cursor name defined in one region can also be defined in another region or in its own subregion. You cannot access a cursor defined in one region from another region. However, SQL Server does allow access to a cursor in a subregion if no other cursor with that same name exists in the subregion.

SQL Server detects name conflicts within a particular scope only during run time. A stored procedure or trigger can define two cursors with the same name if only one is executed. For example:

```
create procedure proc1 (@flag int)
as
if (@flag)
    declare names_crsr cursor
    for select au_fname from authors
else
    declare names_crsr cursor
    for select au_lname from authors
return
```

This stored procedure works because only one *names_crsr* cursor is defined in its scope.

Cursor Scans and the Cursor Result Set

Cursor result set rows may not reflect the values in the actual base table rows. For example, a cursor declared with an **order by** clause usually requires the creation of an internal table to order the rows for the cursor result set. SQL Server does not lock the rows in the base table that correspond to the rows in the internal table, which permits other clients to update these base table rows. In that case, the rows returned to the client from the cursor result set would not be in sync with the base table rows.

A cursor result set is generated as the rows are returned through a **fetch** of that cursor. This means that a cursor **select** query is processed like a normal **select** query. This process, known as **cursor scans**, provides a faster turnaround time and eliminates the need to read rows the application does not require.

SQL Server requires that cursor scans use a unique index of a table, particularly for isolation level 0 reads. If the table has an IDENTITY column and you need to create a nonunique index on it, use the **identity in nonunique index** database option to automatically include an IDENTITY column in the table's index keys so that all indexes created on the table are unique. This database option makes logically nonunique indexes internally unique and allows the indexes to be used to process updatable cursors for isolation level 0 reads.

You can still use cursors that reference tables without indexes, if none of those tables are updated by another process that causes the current row position to move. For example:

```
declare storinfo_crsr cursor
for select stor_id, stor_name, payterms
    from stores
    where state = 'CA'
```

The table *stores* specified with the above cursor does not have any indexes. SQL Server allows the declaration of cursors on tables without unique indexes, but any **update** or **delete** in these tables that moves the position of the rows closes all cursors on such tables.

Making Cursors Updatable

You can update or delete a row returned by a cursor if the cursor is updatable. If the cursor is read-only, you can only read the data; you cannot update or delete it. By default, SQL Server attempts to determine if a cursor is updatable before designating it as read-only.

You can explicitly specify whether a cursor is updatable using the **read only** or **update** keywords in the **declare** statement. For example, the following defines an updatable result set for the *pubs_crsr* cursor:

```
declare pubs_crsr cursor
for select pub_name, city, state
from publishers
for update of city, state
```

The above example includes all the rows from the *publishers* table, but it explicitly defines only the *city* and *state* columns for update.

Unless you plan to update or delete rows through a cursor, you should declare a cursor as read-only. If you do not explicitly specify **read only** or **update**, the cursor is implicitly updatable when the **select** statement does **not** contain any of the following:

- **distinct** option
- **group by** clause
- Aggregate function
- Subquery
- **union** operator
- **at isolation read uncommitted** clause

You cannot specify the **for update** clause if a cursor's *select_statement* contains one of the above constructs. SQL Server also defines a cursor as read-only if you declare certain types of cursors that include an **order by** clause as part of their *select_statement*. For more information, see "Cursors" in the *SQL Server Reference Manual*.

If you do not specify a *column_name_list* with the **for update** clause, all the specified columns in the query are updatable. As described earlier for cursor scans, SQL Server attempts to use unique indexes for updatable cursors when scanning the base table. For cursors, SQL

Server considers an index containing an IDENTITY column to be unique, even if it is not declared so.

SQL Server allows you to include columns in the *column_name_list* that are not specified in the list of columns of the cursor's *select_statement*, but that are part of the tables specified in the *select_statement*.

In the following example, SQL Server uses the unique index on the *pub_id* column of *publishers* (even though *pub_id* is not included in the definition of *newpubs_crsr*):

```
declare newpubs_crsr cursor
for select pub_name, city, state
from publishers
for update
```

If you do not specify the **for update** clause, SQL Server chooses any unique index, although it can also use other indexes or table scans if no unique index exists for the specified table columns. However, when you specify the **for update** clause, SQL Server must use a unique index defined for one or more of the columns to scan the base table. If no unique index exists, it returns an error.

Any columns of the base table you specify in the *column_name_list* of **for update** should include only those columns you need to update, and not any columns included in at least one unique index. This allows SQL Server to use that unique index for its cursor scan, which helps prevent an update anomaly called the **Halloween Problem**.

This problem occurs when a client updates a column of a cursor result set row that defines the order in which the rows are returned from the base tables. For example, if SQL Server accesses a base table using an index and the index key is updated by the client, the updated index row can move within the index and be read again by the cursor. This is a result of an updatable cursor only logically creating a cursor result set. The cursor result set is actually the base tables that derive the cursor.

Opening Cursors

After you declare the cursor, you must open it to **fetch**, **update**, or **delete** rows. Opening a cursor causes SQL Server to evaluate the **select** statement that defines the cursor and make the cursor result set available for processing. The syntax for **open** is:

open ***cursor_name***

After you open a cursor, it is positioned before the first row of the cursor result set. You must use **fetch** to access that first row.

SQL Server does not allow you to open a cursor if it is already open or if the cursor has not been defined with the **declare cursor** statement. You can reopen a closed cursor to reset the cursor position to the beginning of the cursor result set.

Fetching Data Rows Using Cursors

After declaring and opening a cursor, you can fetch rows from that cursor's result set with the **fetch** command. A **fetch** returns one or more rows to the client which is responsible for extracting the column data from the row. Optionally, you can include Transact-SQL parameters or local variables with **fetch** to store column values.

fetch Syntax

The syntax for the **fetch** statement is:

fetch *cursor_name* [into *fetch_target_list*]

For example, after declaring and opening the *authors_crsr* cursor, you can **fetch** the first row of its result set as follows:

```
fetch authors_crsr
au_id        au_lname            au_fname
-----------  ------------------- ---------------
341-22-1782  Smith               Meander
```

Each subsequent **fetch** retrieves another row from the cursor result set. For example:

```
fetch authors_crsr
au_id       au_lname            au_fname
----------- ------------------- ---------------
527-72-3246 Greene              Morningstar
```

After you **fetch** all the rows, the cursor points to the last row of the result set. If you **fetch** again, SQL Server returns a warning through the *@@sqlstatus* variable (described below) indicating there is no more data. The cursor position remains unchanged.

You cannot **fetch** a row that has already been fetched. There is no way to backtrack through a cursor result set. You can close and reopen the cursor to generate the cursor result set again and start fetching from the beginning.

The **into** clause specifies that SQL Server returns column data into the specified variables. The *fetch_target_list* must consist of previously declared Transact-SQL parameters or local variables.

For example, after declaring the *@name*, *@city*, and *@state* variables, you can fetch rows from the *pubs_crsr* cursor as follows:

```
fetch pubs_crsr into @name, @city, @state
```

SQL Server expects a one-to-one correspondence between the variables in the *fetch_target_list* and the target list expressions specified by the *select_statement* that defines the cursor. The datatypes of the variables or parameters must be compatible with the datatypes of the columns in the cursor result set.

Checking the Cursor Status

SQL Server returns a status value after each fetch. You can access the value through the global variable *@@sqlstatus*. The following table lists possible *@@sqlstatus* values and their meaning:

Table 16-1: @@sqlstatus values

Value	Meaning
0	Indicates successful completion of the **fetch** statement.
1	Indicates that the **fetch** statement resulted in an error.
2	Indicates that there is no more data in the result set. This warning can occur if the current cursor position is on the last row in the result set and the client submits a **fetch** statement for that cursor.

The following example determines the *@@sqlstatus* for the currently open *authors_crsr* cursor:

```
select @@sqlstatus

---------
        0


(1 row affected)
```

Only a **fetch** statement can set *@@sqlstatus*. All other statements have no effect on *@@sqlstatus*.

Checking the Number of Rows Fetched

SQL Server also provides the *@@rowcount* global variable. *@@rowcount* allows you to monitor the number of rows of the cursor result set returned to the client up to the last fetch. In other words, it represents the total number of rows seen by the cursor at any one time.

Once all the rows are read from a cursor result set, *@@rowcount* represents the total number of rows in that result set. Each open cursor is associated with a specific *@@rowcount* variable. The variable is dropped when you close the cursor. Checking *@@rowcount* after a **fetch** provides you with the number of rows read for the cursor specified in that **fetch**.

The following example determines the *@@rowcount* for the currently open *authors_crsr* cursor:

```
select @@rowcount
---------
        1

(1 row affected)
```

Getting Multiple Rows with Each *fetch*

By default, the **fetch** command only brings back one row at a time. You can use the **cursor rows** option of the **set** command to change the number of rows **fetch** returns. However, this option does not affect a **fetch** containing an **into** clause.

The syntax for **set** is:

```
set cursor rows number for cursor_name
```

where *number* specifies the number of rows for the cursor. The default setting is 1 for each cursor you declare. You can **set** the **cursor rows** option for a cursor whether it is open or closed.

For example, you can change the number of rows fetched for the *authors_crsr* cursor as follows:

```
set cursor rows 3 for authors_crsr
```

Afterwards, each **fetch** of *authors_crsr* returns 3 rows from the cursor result set:

```
fetch authors_crsr
```

```
au_id       au_lname            au_fname
----------- ------------------- ---------------
648-92-1872 Blotchet-Halls      Reginald
712-45-1867 del Castillo        Innes
722-51-5424 DeFrance            Michel
```

The cursor is positioned on the last row fetched (the author Michel DeFrance in the above example).

Fetching several rows at a time works especially well for client applications. If you fetch more than one row, Open Client or Embedded SQL™ automatically buffer the rows sent to the client application. The client still sees a row-by-row access, but each **fetch** results in fewer calls to SQL Server, which improves performance.

Updating and Deleting Rows Using Cursors

If the cursor is updatable, you can use the **update** and **delete** statements to update or delete rows. SQL Server determines if the cursor is updatable by checking the *select_statement* that defines the cursor. You can also explicitly define a cursor as updatable with the **for update** clause of the **declare cursor** statement. See "Making Cursors Updatable" on page 16-6 for more information.

Deleting Cursor Result Set Rows

Using the **where current of** clause of the **delete** statement, you can delete the row at the current cursor position. When you delete a row from the cursor's result set, the row is deleted from the underlying database table. You can delete only one row at a time using the cursor.

The syntax for **delete...where current of** is:

```
delete [from]
   [[database.]owner.]{table_name|view_name}
   where current of cursor_name
```

The *table_name* or *view_name* specified with a **delete...where current of** must be the table or view specified in the first **from** clause of the **select** statement that defines the cursor.

For example, you can delete the row that the *authors_crsr* cursor currently points to as follows:

```
delete from authors
where current of authors_crsr
```

The **from** keyword in the above example is optional.

➤ ***Note***

You cannot delete a row from a cursor defined by a **select** statement containing a join, even if the cursor is updatable.

After you delete a row from a cursor, SQL Server positions the cursor before the row following the deleted row in the cursor's result set. You must still use **fetch** to access that next row. If the deleted row is the last row in the cursor result set, SQL Server positions the cursor after the last row of the result set.

For example, after deleting the current row in the above example (the author Michel DeFrance), you can fetch the next three authors in the cursor result set (assuming **cursor rows** is still set to 3):

```
fetch authors_crsr
au_id        au_lname            au_fname
-----------  ------------------- ---------------
807-91-6654  Panteley            Sylvia
899-46-2035  Ringer              Anne
998-72-3567  Ringer              Albert
```

You can, of course, delete a row from the base table without referring to a cursor. The cursor result set changes as changes are made to the base table.

Updating Cursor Result Set Rows

Using the **where current of** clause of the **update** statement, you can update the row at the current cursor position. Any update to the cursor result set also affects the base table row from which the cursor row is derived.

The syntax for **update...where current of** is:

```
update [[database.]owner.]{table_name | view_name}
   set [[[database.]owner.]{table_name.|view_name.}]
       column_name1 =
          {expression1|NULL|(select_statement)}
       [, column_name2 =
          {expression2|NULL|(select_statement)}]...
   where current of cursor_name
```

The **set** clause specifies the cursor's result set column name and assigns the new value. When more than one column name and value pair is listed, you must separate them with commas.

The *table_name* or *view_name* must be the table or view specified in the first **from** clause of the **select** statement that defines the cursor. If that **from** clause references more than one table or view (using a join), you can specify only the table or view actually being updated.

For example, you can update the row that the *pubs_crsr* cursor currently points to as follows:

```
update publishers
set city = "Pasadena",
    state = "CA"
where current of pubs_crsr
```

After the update, the cursor position remains unchanged. You can continue to update the row at that cursor position, as long as another SQL statement does not move the position of that cursor.

SQL Server allows you to update columns that are not specified in the list of columns of the cursor's *select_statement*, but are part of the tables specified in that statement. However, when you specify a *column_name_list* with **for update**, you can update only those columns.

Closing and Deallocating Cursors

When you are finished with the result set of a cursor, you can **close** it. The syntax for **close** is:

```
close cursor_name
```

Closing the cursor does not change its definition. You can **open** the cursor again, and SQL Server creates a new cursor result set using the same query as before. For example:

```
close authors_crsr
open authors_crsr
```

You can then fetch from *authors_crsr*, starting from the beginning of its cursor result set. Any conditions associated with that cursor (such as the number of rows fetched defined by **set cursor rows**) remain in effect.

For example:

```
fetch authors_crsr
```

```
au_id       au_lname            au_fname
----------- ------------------- ---------------
341-22-1782 Smith               Meander
527-72-3246 Greene              Morningstar
648-92-1872 Blotchet-Halls      Reginald
```

If you want to discard the cursor, you must **deallocate** it. The syntax for **deallocate** is:

deallocate cursor *cursor_name*

Deallocating a cursor frees up any resources associated with the cursor, including the cursor name. You cannot reuse a cursor name until you deallocate it. If you deallocate an open cursor, SQL Server automatically closes it. Terminating a client connection to a server also closes and deallocates any open cursors.

An Example Using a Cursor

The following cursor example uses this query:

```
select author = au_fname + " " + au_lname, au_id
from authors
order by au_lname
```

The results of the query are

```
author                      au_id
-------------------------   -----------
Abraham Bennet              409-56-7008
Reginald Blotchet-Halls     648-92-1872
Cheryl Carson               238-95-7766
Michel DeFrance             722-51-5454
Ann Dull                    427-17-2319
Marjorie Green              213-46-8915
Morningstar Greene          527-72-3246
Burt Gringlesby             472-27-2349
Sheryl Hunter               846-92-7186
Livia Karsen                756-30-7391
Chastity Locksley           486-29-1786
Stearns MacFeather          724-80-9391
Heather McBadden            893-72-1158
```

```
Michael O'Leary             267-41-2394
Sylvia Panteley             807-91-6654
Anne Ringer                 899-46-2035
Albert Ringer               998-72-3567
Meander Smith               341-22-1782
Dick Straight               274-80-9391
Dirk Stringer               724-08-9931
Johnson White               172-32-1176
Akiko Yokomoto              672-71-3249
Innes del Castillo          712-45-1867
```

The following steps show how to use a cursor with the above query:

1. You must declare the cursor. This **declare cursor** statement defines a cursor using the **select** statement shown above:

   ```
   declare newauthors_crsr cursor for
   select author = au_fname + " " + au_lname, au_id
   from authors
   order by au_lname
   ```

2. Once you declare the cursor, you can open it:

   ```
   open newauthors_crsr
   ```

3. Now you can fetch rows using the cursor:

   ```
   fetch newauthors_crsr

   author                      au_id
   -------------------------   -----------
   Abraham Bennet              409-56-7008
   ```

4. You can fetch more than one row at a time by specifying the number of rows with the **set** command:

   ```
   set cursor rows 5 for newauthors_crsr
   fetch newauthors_crsr

   author                      au_id
   -------------------------   -----------
   Reginald Blotchet-Halls     648-92-1872
   Cheryl Carson               238-95-7766
   Michel DeFrance             722-51-5454
   Ann Dull                    427-17-2319
   Marjorie Green              213-46-8915
   ```

   Each subsequent **fetch** brings back five more rows:

   ```
   fetch newauthors_crsr
   ```

```
author                      au_id
-------------------------   -----------
Morningstar Greene          527-72-3246
Burt Gringlesby             472-27-2349
Sheryl Hunter               846-92-7186
Livia Karsen                756-30-7391
Chastity Locksley           486-29-1786
```

5. Once you are finished with the cursor, you can close it:

   ```
   close newauthors_crsr
   ```

   Closing the cursor releases the result set, but the cursor is still defined. If you **open** the cursor again, SQL Server reruns the query and places the cursor before the first row in the result set. The cursor is still set to return five rows with each **fetch**.

6. Use the **deallocate** command to make the cursor undefined:

   ```
   deallocate cursor newauthors_crsr
   ```

   You cannot reuse the cursor name until you **deallocate** it.

Using Cursors in Stored Procedures

Cursors are particularly useful in stored procedures. They allow you to accomplish the same task using only one query that would otherwise require several queries. However, all cursor operations must execute within a single procedure. A stored procedure cannot **open**, **fetch**, or **close** a cursor that was not declared in the procedure. The cursor is undefined outside of the scope of the stored procedure.

For example, the following stored procedure *au_sales* checks the *sales* table to see if any books by a particular author have sold well:

```
create procedure au_sales (@author_id id)
as

/* declare local variables used for fetch */
declare @title_id tid
declare @title varchar(80)
declare @ytd_sales int
declare @msg varchar(120)
```

```
/* declare the cursor to get each book written
    by given author */
declare author_sales cursor for
select ta.title_id, t.title, t.total_sales
from titleauthor ta, titles t
where ta.title_id = t.title_id
and ta.au_id = @author_id

open author_sales

fetch author_sales
        into @title_id, @title, @ytd_sales

if (@@sqlstatus = 2)
begin
    print "We do not sell books by this author."
    close author_sales
    return
end

/* if cursor result set is not empty, then process
    each row of information */
while (@@sqlstatus = 0)
begin
    if (@ytd_sales = NULL)
    begin
        select @msg = @title +
           " had no sales this year."
        print @msg
    end
    else if (@ytd_sales < 500)
    begin
        select @msg = @title +
            " had poor sales this year."
        print @msg
    end
    else if (@ytd_sales < 1000)
    begin
```

```
            select @msg = @title +
                " had mediocre sales this year."
            print @msg
        end
        else
        begin
            select @msg = @title +
                " had good sales this year."
            print @msg
        end

        fetch author_sales into @title_id, @title,
        @ytd_sales
    end

    /* if error occurred, call a designated handler */
    if (@@sqlstatus = 1) exec error_handle

    close author_sales

    deallocate cursor author_sales

    return
```

For more information about stored procedures, see Chapter 14, "Using Stored Procedures."

Cursors and Locking

Cursor **locking** methods are similar to the current locking methods for SQL Server. In general, statements that read data (such as **select** or **readtext**) use shared locks on each data page to avoid reading changed data from an uncommitted transaction. Update statements use **exclusive locks** on each page they change. To reduce deadlocks and improve concurrency, SQL Server often precedes an exclusive lock with an update lock, which indicates that the client intends to change data on the page.

For updatable cursors, SQL Server uses update locks by default when scanning tables or views referenced with the **for update** clause of **declare cursor.** If the **for update** clause is included, but the list is empty, all tables and views referenced in the **from** clause of the *select_statement* receive update locks by default. If the **for update** clause is not included, the referenced tables and views receive shared locks. You can instruct

SQL Server to use shared locks instead of update locks by adding the **shared** keyword to the **from** clause. Specifically, you should add **shared** after each table name for which you prefer a **shared lock**.

For information about SQL Server locking, see the *System Administration Guide*. For more information about cursors and locking, see the *SQL Server Reference Manual*.

Getting Information About Cursors

SQL Server provides the system procedure **sp_cursorinfo**, which displays information about the cursor name, its current status (such as open or closed), and its result columns. The following example displays information about the *authors_crsr* cursor:

```
sp_cursorinfo 0, authors_crsr

Cursor name 'authors_crsr' is declared at nesting
    level '0'.
The cursor id is 327681
The cursor has been successfully opened 1 times
The cursor was compiled at isolation level 1.
The cursor is not open.
The cursor will remain open when a transaction is
    committed or rolled back.
The number of rows returned for each FETCH is 1.
The cursor is updatable.
There are 3 columns returned by this cursor.

The result columns are:
Name = 'au_id', Table = 'authors', Type = ID,
    Length = 11 (updatable)
Name = 'au_lname', Table = 'authors', Type =
    VARCHAR, Length = 40 (updatable)
Name = 'au_fname', Table = 'authors', Type =
    VARCHAR, Length = 20 (updatable)
```

For more information about **sp_cursorinfo**, see the *SQL Server Reference Manual*.

17 Transactions: Maintaining Data Consistency and Recovery

Transactions provide a way to group Transact-SQL statements so that they are treated as a unit. Either all statements in the group are executed or no statements are executed.

This chapter discusses:

- An overview of transactions
- How to use group statements in a transaction
- How to define transaction modes and isolation levels
- How stored procedures and triggers work with transactions
- How cursors work with transactions
- Backup and recovery of transactions

What Are Transactions?

A transaction is a mechanism for ensuring that a set of one or more SQL statements is treated as a single unit of work. SQL Server automatically manages all data modification commands, including single-step change requests, as transactions. By default, each **insert**, **update**, and **delete** statement is considered a single transaction.

You can group a set of SQL statements into a user-defined transaction with the **begin transaction**, **commit transaction**, and **rollback transaction** commands. **begin transaction** marks the beginning of a transaction block. All subsequent statements, up to a **rollback transaction** or a matching **commit transaction**, are included as part of the transaction.

Transactions allow SQL Server to guarantee:

- Consistency – Simultaneous queries and change requests cannot collide with each other, and users never see or operate on data that is part way through a change.
- Recovery – In case of system failure, database recovery is complete and automatic.

To support SQL standards-compliant transactions, SQL Server provides options that allow you to select the mode and isolation level for your transactions. Applications that require SQL standards-compliant transactions should set those options at the beginning of

every session. Transaction modes and isolation levels are described later in this chapter.

Transactions and Consistency

In a multiuser environment, SQL Server must prevent simultaneous queries and data modification requests from interfering with each other. This is important because if the data being processed by a query could be changed by another user's update while the query runs, the results of the query would be ambiguous.

SQL Server automatically sets the appropriate level of locking for each transaction. You can make shared locks more restrictive on a query-by-query basis by including the **holdlock** keyword in a **select** statement.

User-defined transactions allow users to instruct SQL Server to process any number of SQL statements as a single unit. They are discussed in a later section.

Transactions and Recovery

A transaction is both a unit of work and a unit of recovery. The fact that SQL Server handles single-step change requests as transactions means that the database can be recovered completely in case of failures.

SQL Server's recovery time is measured in seconds and minutes. You can specify the maximum acceptable recovery time.

The SQL commands related to recovery and backup are discussed in "Backup and Recovery of Transactions" on page 17-21.

Using Transactions

begin transaction and **commit transaction** tell SQL Server to process any number of single commands as a single unit. **rollback transaction** undoes the transaction, either back to its beginning, or back to a savepoint. You define a **savepoint** inside a transaction with the **save transaction** command.

User-defined transactions give you control over transaction management. They also improve performance, since system overhead is incurred once per transaction, rather than once for each individual command.

➤ ***Note***

Grouping large numbers of Transact-SQL commands into one long-running transaction may affect recovery time. If SQL Server fails before the transaction commits, recovery is longer, because SQL Server must undo the transaction.

Any user can define a transaction. No permission is required for any of the transaction commands.

The following sections discuss general transaction topics and transaction commands, with examples. For more information about transactions, see the *SQL Server Reference Manual*.

Allowing Data Definition Commands in Transactions

You can use certain data definition language commands in transactions by setting **sp_dboption**'s **ddl in tran** option to true. If **ddl in tran** is true in a particular database, you can issue commands such as **create table**, **grant**, and **alter table** inside transactions in that database. If **ddl in tran** is true in the *model* database, you can issue the commands inside transactions in all databases created after **ddl in tran** was set to true in *model*.

◆ ***WARNING!***

The only scenario in which using data definition language commands inside transactions is justified is in create schema. Data definition language commands hold locks on system tables such as *sysobjects*. If you use data definition language commands inside transactions, keep the transactions short.

In particular, avoid using any data definition language commands on *tempdb* within transactions, lest your system grind to a halt. Always leave ddl in tran set to false in *tempdb*.

To set **ddl in tran** to true, type:

```
sp_dboption mydb,"ddl in tran", true
```

The first parameter specifies the name of the database in which to set the option. You must be using the *master* database to execute **sp_dboption**. Any user can execute **sp_dboption** with no parameters to

display the current option settings. To set options, however, you must be either a System Administrator or the Database Owner.

The following commands are allowed inside a user-defined transaction only if the **ddl in tran** option to **sp_dboption** is set to true:

Table 17-1: DDL commands allowed in transactions

alter table (clauses other than **partition** and **unpartition** are allowed)	**create default**	**drop default**	**grant**
	create index	**drop index**	**revoke**
	create procedure	**drop procedure**	
	create rule	**drop rule**	
	create schema	**drop table**	
	create table	**drop trigger**	
	create trigger	**drop view**	
	create view		

System procedures that change the *master* database or create temporary tables cannot be used inside user-defined transactions.

Never use the following commands inside a user-defined transaction:

Table 17-2: DDL commands not allowed in transactions

alter database	**disk init**	**load transaction**	**select into**
alter table...partition	**dump database**	**load database**	**update statistics**
alter table...unpartition	**dump transaction**	**reconfigure**	**truncate table**
create database	**drop database**		

You can check the current setting of **ddl in tran** with **sp_helpdb**.

Beginning and Committing Transactions

The **begin transaction** and **commit transaction** commands can enclose any number of SQL statements and stored procedures. The syntax for both statements is:

```
begin {transaction | tran} [transaction_name]
commit {transaction | tran | work} [transaction_name]
```

transaction_name is the name assigned to the transaction. It must conform to the rules for identifiers.

The keywords **transaction**, **tran**, and **work** (in **commit transaction**) are synonymous; you can use one in the place of the others. However,

transaction and **tran** are Transact-SQL extensions; only **work** is SQL standards-compliant.

Here is a skeletal example:

```
begin tran
    statement
    procedure
    statement
commit tran
```

commit transaction does not affect SQL Server if a transaction is not currently active.

Rolling Back and Saving Transactions

If a transaction must be canceled before it is committed—either because of some failure or because of a change by the user—all of its completed statements or procedures must be undone.

You can cancel or roll back a transaction with the **rollback transaction** command at any time before the **commit transaction** command has been given. Using savepoints, you can cancel either an entire transaction or part of it. However, you cannot cancel a transaction after it has been committed.

The syntax of the **rollback transaction** command is:

```
rollback {transaction | tran | work}
   [transaction_name | savepoint_name]
```

A **savepoint** is a marker that the user puts inside a transaction to indicate a point to which it can be rolled back.

Savepoints are inserted by putting a **save transaction** command within the transaction. The syntax is:

```
save {transaction | tran} savepoint_name
```

The savepoint name must conform to the rules for identifiers.

If no *savepoint_name* or *transaction_name* is given with the **rollback transaction** command, the transaction is rolled back to the first **begin transaction** in a batch.

Here is how you can use the **save transaction** and **rollback transaction** commands:

```
begin tran transaction_name
  statement
  statement
  procedure
save tran savepoint_name
  statement
rollback tran savepoint_name
  statement
  statement
rollback tran
```

The first **rollback transaction** command rolls the transaction back to the savepoint inside the transaction. The second **rollback transaction** rolls the transaction back to its beginning. If a transaction is rolled back to a savepoint, it must still proceed to completion or else be canceled altogether.

Until you issue a **commit transaction**, SQL Server considers all subsequent statements to be part of the transaction, unless it encounters another **begin transaction** statement. At that point, SQL Server considers all subsequent statements to be part of this new nested transaction. Nested transactions are described in the next section.

rollback transaction or **save transaction** does not affect SQL Server and does not return an error message if a transaction is not currently active.

Checking the State of Transactions

The global variable *@@transtate* keeps track of the current state of a transaction. SQL Server determines what state to return by keeping track of any transaction changes after a statement executes. *@@transtate* may contain the following values:

Table 17-3: @@transtate values

Value	Meaning
0	Transaction in progress. An explicit or implicit transaction is in effect; the previous statement executed successfully.
1	Transaction succeeded. The transaction completed and committed its changes.
2	Statement aborted. The previous statement was aborted; no effect on the transaction.
3	Transaction aborted. The transaction aborted and rolled back any changes.

In a transaction, you can use *@@transtate* after a statement (such as an **insert**) to determine whether it was successful or aborted, and to determine its effect on the transaction. The following example checks *@@transtate* during a transaction (after a successful **insert**) and after the transaction commits:

```
begin transaction

insert into publishers (pub_id) values ('9999')
(1 row affected)
select @@transtate
----------
         0

(1 row affected)
commit transaction

select @@transtate
----------
         1

(1 row affected)
```

This next example checks *@@transtate* after an unsuccessful **insert** (due to a rule violation) and after the transaction rolls back:

```
begin transaction

insert into publishers (pub_id) values ('7777')
Msg 552, Level 16, State 1:
A column insert or update conflicts with a rule
bound to the column. The command is aborted. The
conflict occured in database 'pubs2', table
'publishers', rule 'pub_idrule', column 'pub_id'.
select @@transtate
----------
         2

(1 row affected)
rollback transaction
select @@transtate
```

```
----------
         3

(1 row affected)
```

Unlike *@@error*, however, SQL Server does not clear *@@transtate* after every statement. It changes *@@transtate* only in response to an action taken by a transaction.

Nested Transactions

You can nest transactions within other transactions. When you nest **begin transaction** and **commit transaction** statements, the outermost pair actually begin and commit the transaction. The inner pairs just keep track of the nesting level. SQL Server does not commit the transaction until the **commit transaction** that matches the outermost **begin transaction** is issued.

SQL Server provides a global variable, *@@trancount*, that keeps track of the current nesting level for transactions. An initial implicit or explicit **begin transaction** sets *@@trancount* to 1. Each subsequent **begin transaction** increments *@@trancount*, and a **commit transaction** decrements it. Firing a trigger also increments *@@trancount*, and the transaction begins with the statement that causes the trigger to fire. Nested transactions are not committed until *@@trancount* equals 0.

For example, the following nested groups of statements are not committed by SQL Server until the final **commit transaction**:

```
begin tran
    select @@trancount
    /* @@trancount = 1 */

    begin tran
        select @@trancount
        /* @@trancount = 2 */

        begin tran
            select @@trancount
            /* @@trancount = 3 */
        commit tran

    commit tran

commit tran
```

```
select @@trancount
/* @@ trancount = 0 */
```

When you nest a **rollback transaction** statement without including a transaction or savepoint name, it always rolls back to the outermost **begin transaction** statement and cancels the transaction.

Example of a User-Defined Transaction

This example shows how a user-defined transaction might be specified:

```
begin transaction royalty_change

/* A user sets out to change the royalty split */
/* for the two authors of The Gourmet Microwave. */
/* Since the database would be inconsistent */
/* between the two updates, they must be grouped */
/* into a transaction. */

update titleauthor
set royaltyper = 65
from titleauthor, titles
where royaltyper = 75
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"

update titleauthor
set royaltyper = 35
from titleauthor, titles
where royaltyper = 25
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"

save transaction percent_changed

/* After updating the royaltyper entries for */
/* the two authors, the user inserts the */
/* savepoint “percent_changed,” and then checks */
/* to see how a 10 percent increase in the
/* price would affect the authors’ royalty */
/* earnings. */
```

```
update titles
set price = price * 1.1
where title = "The Gourmet Microwave"


select (price * royalty * total_sales) * royaltyper
from titles, titleauthor, roysched
where title = "The Gourmet Microwave"
and titles.title_id = titleauthor.title_id
and titles.title_id =roysched.title_id

rollback transaction percent_changed

/* The transaction rolls back to the savepoint */
/* with the rollback transaction command. */
/* Without a savepoint, it would roll back to */
/* the begin transaction. */

commit transaction
```

Selecting Transaction Mode and Isolation Level

SQL Server provides two options you can set to support SQL standard-compliant transactions. These options define the transaction mode and transaction isolation level. You should set these options at the beginning of every session that requires SQL standards-compliant transactions.

SQL Server supports the following transaction modes:

- The default mode, called **unchained** or Transact-SQL mode, requires explicit **begin transaction** statements paired with **commit transaction** or **rollback transaction** statements to complete the transaction.
- The SQL standards-compatible mode, called **chained** mode, implicitly begins a transaction before any data retrieval or modification statement. These statements include: **delete**, **insert**, **open**, **fetch**, **select**, and **update**. You must still explicitly end the transaction with **commit transaction** or **rollback transaction**.

You can set either mode using the **chained** option of the **set** command. However, you should not mix these transaction modes in your applications. The behavior of stored procedures and triggers can

vary depending on the mode, and you may require special action to run a procedure in one mode that was created in the other.

SQL Server supports the following transaction isolation levels:

- Level 0 – SQL Server ensures that data written by one transaction represents the actual data. This level prevents other transactions from writing over the same data until the transaction commits. The other transactions can still read the uncommitted data.
- Level 1 – SQL Server ensures that data read by one transaction represents the actual data, not the data in the process of another uncommitted transaction. This is the default isolation level supported by SQL Server.
- Level 3 – SQL Server ensures that data read by one transaction is valid until the end of that transaction. It supports this level through the **holdlock** keyword of the **select** statement which applies a read-lock on the specified data.

You can set the isolation level for your session using the **transaction isolation level** option of the **set** command. You can enforce the isolation level for just a query as opposed to using the **at isolation** clause of the **select** statement.

The following sections describe these options in more detail.

Choosing a Transaction Mode

The SQL standards require every SQL data-retrieval and data-modification statement to occur inside of a transaction. A transaction automatically starts with the first data-retrieval or data-modification statement after the start of a session or after the previous transaction commits or aborts. This is the chained transaction mode.

You can set this mode for your current session by turning on the **chained** option of the **set** statement. For example:

```
set chained on
```

However, you cannot execute the **set chained** command within a transaction. To return to the unchained transaction mode, set the **chained** option **off**. The default transaction mode is unchained.

In the chained transaction mode, SQL Server implicitly executes a **begin transaction** statement just before the following data retrieval or modification statements: **delete**, **insert**, **open**, **fetch**, **select**, and **update**. For example, the following group of statements produce different results depending on which mode you use:

```
insert into publishers
    values ('9999', null, null, null)
begin transaction
delete from publishers where pub_id = '9999'
rollback transaction
```

In unchained mode, the **rollback** affects only the **delete** statement, so *publishers* still contains the inserted row. In chained mode, the **insert** statement implicitly begins a transaction, and the rollback affects all statements up to the beginning of that transaction, including the **insert**.

Although chained mode implicitly begins transactions with data retrieval or modification statements, you can nest transactions only by explicitly using **begin transaction** statements. Once the first transaction implicitly begins, further data retrieval or modification statements no longer begin transactions until after the first transaction commits or aborts. For example, in the following query, the first **commit transaction** commits all changes in chained mode; the second commit is unnecessary:

```
insert into publishers
    values ('9999', null, null, null)
    insert into publishers
        values ('9997', null, null, null)
    commit transaction
commit transaction
```

➤ ***Note***

In chained mode, a data retrieval or modification statement begins a transaction whether or not it executes successfully. Even a **select** that does not access a table begins a transaction.

You can check the global variable *@@tranchained* to determine SQL Server's current transaction mode. **select** *@@tranchained* returns a 0 for unchained mode or a 1 for chained mode.

Choosing an Isolation Level

The SQL92 standard defines four levels of isolation for transactions. Each isolation level specifies the kinds of actions that are not permitted while concurrent transactions are executing. Higher levels include the restrictions imposed by the lower levels:

- Level 0 prevents other transactions from changing data that has already been modified (through an **insert**, **delete**, **update**, and so on) by an uncommitted transaction. The other transactions are blocked from modifying that data until the transaction commits. However, other transactions can still read the uncommitted data, which results in **dirty reads**.
- Level 1 prevents dirty reads. Such reads occur when one transaction modifies a row, and then a second transaction reads that row before the first transaction commits the change. If the first transaction rolls back the change, the information read by the second transaction becomes invalid.
- Level 2 prevents **nonrepeatable reads**. Such reads occur when one transaction reads a row and a second transaction modifies that row. If the second transaction commits its change, subsequent reads by the first transaction yield different results than the original read.
- Level 3 prevents **phantoms**. Phantoms occur when one transaction reads a set of rows that satisfy a search condition, and then a second transaction modifies the data (through an **insert**, **delete**, **update**, and so on). If the first transaction repeats the read with the same search conditions, it obtains a different set of rows.

By default, SQL Server's transaction isolation level is 1. The SQL92 standard requires that level 3 be the default isolation for all transactions. This prevents dirty reads, nonrepeatable reads, and phantoms. To enforce this default level of isolation, Transact-SQL provides the **transaction isolation level 3** option of the **set** statement. This option instructs SQL Server to automatically apply a **holdlock** to all **select** operations in a transaction. For example:

```
set transaction isolation level 3
```

Applications that use **transaction isolation level 3** should set that isolation level at the beginning of each session. However, setting **transaction isolation level 3** causes SQL Server to hold any read-locks for the duration of the transaction. If you also use the chained transaction mode, that isolation level remains in effect for any data retrieval or modification statement that implicitly begins a transaction. In both cases, this can lead to concurrency problems for some applications, since more locks may be held for longer periods of time.

To return your session to the SQL Server default isolation level:

```
set transaction isolation level 1
```

Applications that are not impacted by dirty reads may see better concurrency and reduced deadlocks when accessing the same data

by setting **transaction isolation level 0** at the beginning of each session. An example is an application that finds the momentary average balance for all savings accounts stored in a table. Since it requires only a snapshot of the current average balance, which probably changes frequently in an active table, the application should query the table using isolation level 0. Other applications that require data consistency, such as deposits and withdrawals to specific accounts in the table, should avoid using isolation level 0.

Queries executing at isolation level 0 do not acquire any read locks for their scans, so they do not block other transactions from writing to the same data, and vice versa. However, even if you set your isolation level to 0, utilities (like **dbcc**) and data modification statements (like **update**) still acquire read locks for their scans, because they must maintain the database integrity by ensuring that the correct data has been read before modifying it.

The global variable *@@isolation* contains the current isolation level of your Transact-SQL session. Querying *@@isolation* returns the value of the active level (0, 1, or 3). For example:

```
select @@isolation

--------
       1


(1 row affected)
```

For more information about isolation levels and locking, see the *Performance and Tuning Guide*.

Changing the Isolation Level for a Query

You can change the isolation level for a query by using the **at isolation** clause with the **select** or **readtext** statements. The **read uncommitted**, **read committed**, and **serializable** options of **at isolation** represent each isolation level as defined below:

at isolation Option	Isolation Level
read uncommited	0
read committed	1
serializable	3

For example, the following two statements query the same table at isolation levels 0 and 3, respectively:

```
select *
from titles
at isolation read uncommitted

select *
from titles
at isolation serializable
```

The **at isolation** clause is valid only for single **select** and **readtext** queries or in the **declare cursor** statement. SQL Server returns a syntax error if you use **at isolation** as follows:

- With a query using the **into** clause
- Within a subquery
- With a query in the **create view** statement
- With a query in the **insert** statement
- With a query using the **for browse** clause

If there is a **union** operator in the query, you must specify the **at isolation** clause after the last **select**.

The SQL92 standard defines **read uncommitted**, **read committed**, and **serializable** as options for **at isolation** (and **set transaction isolation level** as well). A Transact-SQL extension also allows you to specify 0, 1, or 3 for **at isolation**. To simplify the discussion of isolation levels, the **at isolation** examples in this manual do not use this extension.

You can also enforce isolation level 3 using the **holdlock** keyword of the **select** statement. However, you cannot specify **holdlock**, **noholdlock**, or **shared** in a query that also specifies **at isolation read uncommitted**. When you use different ways to set an isolation level, the **holdlock** keyword takes precedence over the **at isolation** clause (except for isolation level 0), and **at isolation** takes precedence over the session level defined by **set transaction isolation level**.

Cursors and Isolation Levels

You can use the **select** statement's **at isolation** clause to change the isolation level with a cursor. For example:

```
declare commit_crsr cursor
for select *
from titles
at isolation read committed
```

This statement makes the cursor operate at isolation level 1, regardless of the isolation level of the transaction or session. If you declare a cursor at isolation level 0 (**read uncommitted**), SQL Server also

defines the cursor as read-only. You cannot specify the **for update** clause along with **at isolation read uncommitted** in a **declare cursor** statement.

SQL Server decides a cursor's isolation level when you open it, not when it is declared. Once you open the cursor, SQL Server determines its isolation level based on the following:

- If the cursor was declared with the **at isolation** clause, that isolation level overrides the transaction isolation level in which it is opened.
- If the cursor was **not** declared with **at isolation**, the cursor uses the isolation level in which it is opened. If you close the cursor and reopen it later, the cursor acquires the current isolation level of the transaction.

The caveat to the last point is that certain types of cursors (language and client) declared in a transaction with isolation level 1 or 3 cannot be opened in a transaction with isolation level 0. For more information about this restriction and about the different types of cursors, see the *SQL Server Reference Manual.*

Stored Procedures and Isolation Levels

The Sybase system-stored procedures always operate at isolation level 1, regardless of the transaction or session isolation level. User stored procedures operate at the isolation level of the transaction that executes it. If the isolation level changes within a stored procedure, the new isolation level remains in effect only during the execution of the stored procedure.

Triggers and Isolation Levels

Since triggers are fired by data modification statements (like **insert**), all triggers execute at either the transaction's isolation level or isolation level 1, whichever is higher. So, if a trigger fires in a transaction at level 0, SQL Server sets the trigger's isolation level to 1 before executing its first statement.

Using Transactions in Stored Procedures and Triggers

You can use transactions in stored procedures and triggers just as with statement batches. If a transaction in a batch or stored procedure invokes another stored procedure or trigger containing a transaction, that second transaction is nested into the first one.

The first explicit or implicit (using chained mode) **begin transaction** starts the transaction in the batch, stored procedure, or trigger. Each subsequent **begin transaction** increments the nesting level. Each subsequent **commit transaction** decrements the nesting level until it reaches 0. SQL Server then commits the entire transaction. A **rollback transaction** aborts the entire transaction up to the first **begin transaction** regardless of the nesting level or the number of stored procedures and triggers it spans.

In stored procedures and triggers, the number of **begin transaction** statements must match the number of **commit transaction** statements. This also applies to stored procedures that use chained mode. The first statement that implicitly begins a transaction must also have a matching **commit transaction.**

The following illustration demonstrates what can happen when you nest transaction statements within stored procedures:

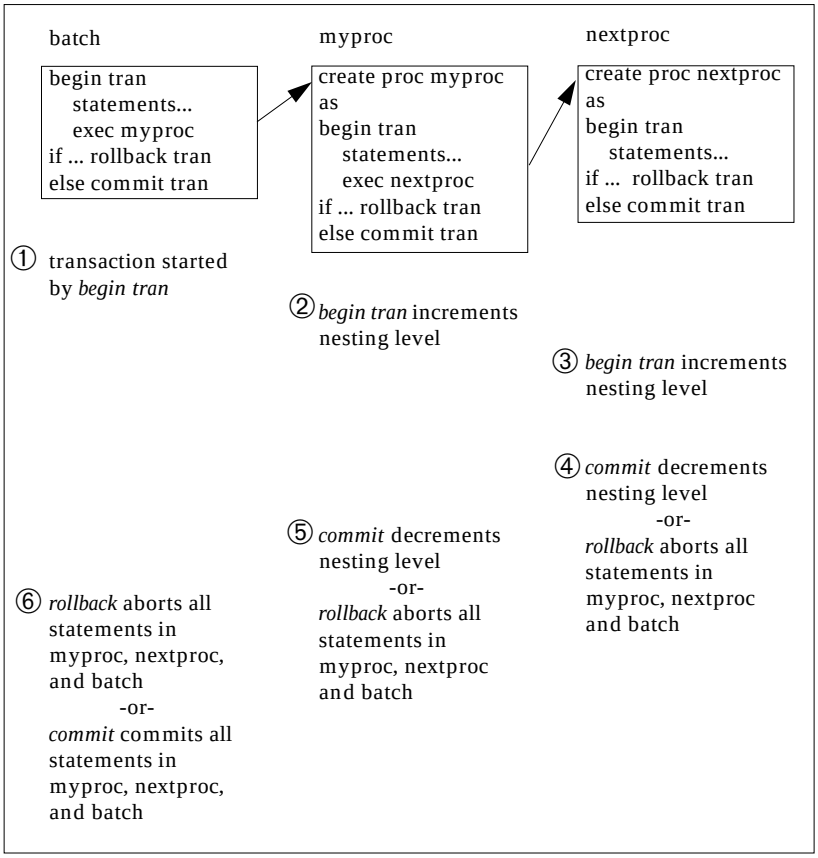


Figure 17-1: Nesting transaction statements

rollback transaction statements in stored procedures do not affect subsequent statements in the procedure or batch that originally called the procedure. SQL Server executes subsequent statements in the stored procedure or batch. However, **rollback transaction** statements in triggers do abort the batch so that subsequent statements are not executed.

For example, the following batch calls the stored procedure *myproc* which includes a **rollback transaction** statement:

```
begin tran
update titles set ...
insert into titles ...
execute myproc
delete titles where ...
```

The **update** and **insert** statements are rolled back and the transaction is aborted. SQL Server continues the batch and executes the **delete** statement. However, if there is an **insert** trigger on a table that includes a **rollback transaction**, the entire batch is aborted and the **delete** is not executed. For example:

```
begin tran
update authors set ...
insert into authors ...
delete authors where ...
```

Different transaction modes or isolation levels for stored procedures have certain requirements, which are described in the next section. Triggers are not affected by the current transaction mode since they are always called as part of a data modification statement.

Transaction Modes and Stored Procedures

Stored procedures written to use the unchained transaction mode may be incompatible with other transactions using chained mode, and vice versa. For example, following is a valid stored procedure using chained transaction mode:

```
create proc myproc
as
insert into publishers
    values ('9999', null, null, null)
commit work
```

A program using unchained transaction mode would fail if it called this procedure because the **commit** does not have a corresponding **begin**. You may encounter other problems:

- Applications that start a transaction using chained mode may create impossibly long transactions, or may hold data locks for the entire length of their session. This behavior degrades SQL Server performance.
- Applications may nest transactions at unexpected times. This can produce different results depending on the transaction mode.

As a rule, applications using one transaction mode should call stored procedures written to use that same mode. The exceptions to that

rule are Sybase system-stored procedures (not including **sp_procxmode**, described below), which can be invoked by sessions using any transaction mode. If no transaction is active when you execute a system-stored procedure, SQL Server turns off chained mode for the duration of the procedure. Before returning, it resets the mode its original setting.

SQL Server tags all procedures with the transaction mode ("chained" or "unchained") of the session in which they are created. This helps avoid problems associated with transactions using one mode invoking other transactions using the other mode. A stored procedure tagged as "chained" is not executable in sessions using unchained transaction mode, and vice versa.

◆ ***WARNING!***

When using transaction modes, be aware of the effects each setting can have on your applications.

Setting Transaction Modes for Stored Procedures

You can use the **sp_procxmode** system stored procedure to change the tag value associated with a stored procedure. SQL Server also provides a third tag, "anymode", which you can use with **sp_procxmode** to indicate stored procedures that can run under either transaction mode. For example:

```
sp_procxmode byroyalty, "anymode"
```

Use **sp_procxmode** without any parameter values to get the transaction modes for all stored procedures in the current database:

```
sp_procxmode
```

```
procedure name               transaction mode
--------------------------   --------------------
byroyalty                    Unchained
discount_proc                Unchained
insert_sales_proc            Unchained
insert_salesdetail_proc      Unchained
storeid_proc                 Unchained
storename_proc               Unchained
title_proc                   Unchained
titleid_proc                 Unchained

(8 rows affected, return status = 0)
```

You can use **sp_procxmode** only in unchained transaction mode.

To change a procedure's transaction mode, you must be a System Administrator, the Database Owner, or the owner of the procedure.

Using Cursors in Transactions

By default, SQL Server does not change a cursor's state (open or closed) when a transaction ends through a commit or roll back. The SQL standards, however, associate an open cursor with its active transaction. Committing or rolling back that transaction automatically closes any open cursors associated with it.

To enforce this SQL standards-compliant behavior, SQL Server provides the **close on endtran** option of the **set** command. In addition, if you set chained mode on, SQL Server starts a transaction when you open a cursor, and it closes that cursor when the transaction is committed or rolled back.

For example, the following sequence of statements produces an error by default:

```
open cursor test
commit tran
open cursor test
```

If you set either the **close on endtran** or **chained** options, the cursor's state changes from open to closed after the commit. This allows the cursor to be reopened.

Any exclusive locks acquired by a cursor in a transaction are held until the end of that transaction. This also applies to shared locks when using the **holdlock** keyword, the **at isolation serializable** clause, or the **set isolation level 3** option. However, if you do not set the **close on endtran** option, the cursor remains open past the end of the transaction, and its current page lock remains in effect. It could also continue to acquire locks as it fetches additional rows.

Backup and Recovery of Transactions

Every change to the database, whether it is the result of a single **update** statement or a grouped set of SQL statements, is automatically recorded in the system table *syslogs*. This table is called the **transaction log**.

Some commands that change the database are not logged, such as **truncate table**, bulk copy into a table that has no indexes, **select into**, **writetext** and **dump transaction with no_log**.

The transaction log records **update**, **insert**, or **delete** statements on a moment-to-moment basis. When a transaction begins, a **begin transaction** event is recorded in the log. As each data modification statement is received, it is recorded in the log.

The change is always recorded in the log before any change is made in the database itself. This type of log, called a write-ahead log, ensures that the database can be recovered completely in case of a failure.

Failures can be due to hardware or media problems, system software problems, application software problems, program-directed cancellations of transactions, or user decisions to cancel a transaction.

In case of any of these failures, the transaction log can be played back against a copy of the database restored from a backup made with the **dump** commands.

To recover from a failure, transactions that were in progress but not yet committed at the time of the failure must be undone, because a partial transaction is not an accurate change. Completed transactions must be redone if there is no guarantee that they have been written to the database device.

If there are active, long-running transactions that are not committed when SQL Server fails, undoing the changes may require as much time as the transaction has been running. Such cases include transactions that do not contain a **commit transaction** or **rollback transaction** to match a **begin transaction**. This prevents SQL Server from writing any changes and increases recovery time.

SQL Server's dynamic dump allows the database and transaction log to be backed up while use of the database continues. Make frequent backups of your database transaction log. The more often you back up your data, the less work will be lost if a system failure occurs.

The owner of each database or a user with OPER authorization is responsible for backing up the database and its transaction log with the **dump** commands, though permission to execute them can be transferred to other users. Permission to use the **load** commands, however, defaults to the Database Owner and cannot be transferred.

Once the appropriate **load** commands are issued, SQL Server handles all aspects of the recovery process. SQL Server also automatically

controls the checkpoint interval, which is the point at which all data pages that have been changed are guaranteed to have been written to the database device. Users can force a checkpoint if necessary with the **checkpoint** command.

For more information, see the *SQL Server Reference Manual* and the *System Administration Guide.*

Glossary

aggregate function

A function that works on a set of cells to produce a single answer or set of answers, one for each subset of cells. The aggregate functions available in Transact-SQL are: average (**avg**), maximum (**max**), minimum (**min**), sum (**sum**), and count of the number of items (**count**).

alias

Allows a SQL Server user to be known in a database as another user. Create aliases with **sp_addalias**.

allocation unit

A logical unit of 1/2 megabyte. The **disk init** command initializes a new database device for SQL Server and divides it into 1/2 megabyte pieces called allocation units.

argument

A value supplied to a function or procedure that is required to evaluate the function.

arithmetic expression

An expression that contains only numeric operands and returns a single numeric value. In Transact-SQL, the operands can be of any SQL Server numeric datatype. They can be functions, variables, parameters, or they can be other arithmetic expressions. Synonymous with **numeric expression**.

arithmetic operators

Addition (+), subtraction (-), division (/), and multiplication (*) can be used with numeric columns. Modulo (%) can be used with *int*, *smallint*, and *tinyint* columns only.

base date

January 1, 1900; the date supplied by SQL Server when a user does not specify a value for a date column.

base tables

The permanent tables on which a view is based. Also called “underlying” tables.

batch

One or more Transact-SQL statements terminated by an end-of-batch signal, which submits them to SQL Server for processing.

Boolean expression

An expression that evaluates to TRUE (1), or FALSE (0). Boolean expressions are often used in control of flow statements, such as **if** or **while** conditions.

built-in functions

A wide variety of functions that take one or more parameters and return results. The built-in functions include mathematical functions, system functions, string functions, text functions, date functions, and type conversion functions.

Cartesian product

All the possible combinations of the rows from each of the tables specified in a join. The number of rows in the Cartesian product is equal to the number of rows in the first table times the number of rows in the second table. Once the Cartesian product is formed, the rows that do not satisfy the join conditions are eliminated.

cascading delete

A delete operation which affects related data in other tables.

chained transaction mode

Determines whether or not SQL Server automatically starts a new transaction on the next data retrieval or data modification statement. When **set chained** is turned **on** outside a transaction, the next data retrieval or data modification statement begins a new transaction. This mode is SQL standard-compliant: it ensures that every SQL data retrieval and data modification statement occur inside a transaction. Chained transaction mode may be incompatible with existing Transact-SQL programs. The default value is **off**. Applications which require standard SQL (such as the Embedded SQL precompiler) should automatically set the **chained** option **on** at the beginning of each session.

character expression

An expression that returns a single character-type value. It can include literals, concatenation operators, functions, and column identifiers.

check constraint

A **check** constraint limits what values users can insert into a column of a table. A **check** constraint specifies a *search_condition* which any value must pass before it is inserted into the table.

checkpoint

The point at which all data pages that have been changed are guaranteed to have been written to the database device.

clauses

A set of keywords and parameters that tailor a Transact-SQL command to meet a particular need. Also called a **keyword phrase**.

client cursor

A cursor declared through Open Client calls or Embedded SQL. The Open Client keeps track of the rows returned from SQL Server and buffers them for the application. Updates and deletes to the result set of client cursors can only be done through the Open Client calls.

clustered index

An index in which the physical order and the logical (indexed) order is the same. The leaf level of a clustered index represents the data pages themselves.

column

The logical equivalent of a field. A column contains an individual data item within a row or record.

column-level constraint

Limit the values of a specified column. Place column-level constraints after the column name and datatype in the **create table** statement, before the delimiting comma.

command

An instruction that specifies an operation to be performed by the computer. Each command or SQL statement begins with a keyword, such as **insert**, that names the basic operation performed. Many SQL commands have one or more **keyword phrases**, or **clauses**, that tailor the command to meet a particular need.

command permissions

Permissions that apply to commands. See also **object permissions**.

command terminator

The end-of-batch signal that sends the batch to SQL Server for processing.

comparison operators

Used to compare one value to another in a query. Comparison operators include equal to (=), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), not equal to (!=), not greater than (!>), and not less than (!<).

compatible datatypes

Types that SQL Server automatically converts for implicit or explicit comparison.

composite indexes

Indexes which involve more than one column. Use composite indexes when two or more columns are best searched as a unit because of their logical relationship.

concatenation

Combine expressions to form longer expressions. The expressions can include any combination of binary or character strings, or column names.

constant expression

An expression that returns the same value each time the expression is used. In Transact-SQL syntax statements, *constant_expression* does not include variables or column identifiers.

context-sensitive protection

Protection that provides certain permissions or privileges depending on the identity of the user. This type of protection can be provided using views and the **user_id** built-in function.

control-break report

A report or data display that breaks data into groups and generates summary information for each break. The breaks control the generation of summary data.

control-of-flow language

Transact-SQL's programming-like constructs (such as **if**, **else**, **while**, **goto** label) that control the flow of execution of Transact SQL statements.

correlated subquery

A **subquery** that cannot be evaluated independently, but that depends on the outer query for its results. Also called a repeating subquery, since the subquery is executed once for each row that might be selected by the outer query. See also **nested queries**.

correlation names

Distinguish the different roles a particular table plays in a query, especially a correlated query or **self-join**. Assign correlation names in the from clause and specify the correlation name after the table name:

```
select au1.au_fname, au2.au_fname
from authors au1, authors au2
where au1.zip = au2.zip
```

cursor

A symbolic name associated with a Transact-SQL **select** statement through a declaration statement. Cursors consist of two parts: the **cursor result set** and the **cursor position**.

cursor name space

The region in which the cursor is known. One of three types:

session – The region starts when the client logs onto SQL Server and ends when the client logs off. This region does not include any regions defined by stored procedures or triggers.

stored procedure – The region starts when a stored procedure begins execution and ends when it completes.

trigger – The region starts when a trigger begins execution and ends when it completes.

Cursor names must be unique within a given scoping region. For example, a stored procedure cannot declare two cursors with the same name.

cursor position

Indicates the current row of the cursor. You can explicitly reference that row using statements designed to support cursors, such as **delete** or **update**. Change the current cursor position through **fetch**, which moves the current cursor position one or more rows down the cursor result set.

cursor result set

The set of rows resulting from the execution of the **select** statement associated with the cursor.

cursor scans

The process of generating a cursor result set.

cursor scope

The region in which the cursor is known. The cursor name exists only so long as its scope exists. The scope is one of three types:

Session - the region starts when the client logs onto SQL Server and ends when the client logs off. This region does not include any regions defined by stored procedures or triggers

Stored procedure - the region starts when a stored procedure begins execution and ends when it completes.

Trigger - the region starts when a trigger begins execution and ends when it completes.

Cursor names must be unique within their scope. For example, a stored procedure cannot declare two cursors with the same name.

cursor stability

A locking or isolation level in which SQL Server has a shared lock on the base table pages which contain a current cursor row. The page remains locked until the cursor is no longer positioned on the page (as a result of fetches). If the base table has an index, the corresponding index pages have shared locks as well.

data definition

The process of setting up databases and creating database objects such as tables, indexes, rules, defaults, procedures, triggers, and views.

data dictionary

The system tables that contain descriptions of the **database objects** and how they are structured.

data integrity

The correctness and completeness of data within a database.

data modification

Adding, deleting, or changing information in the database with the **insert**, **delete**, and **update** commands.

data retrieval

Requesting data from the database and receiving the results. Also called a **query**.

database

A set of related data tables and other database objects that are organized and presented to serve a specific purpose.

database object

One of the components of a database: table, view, index, procedure, trigger, column, default, or rule.

Database Owner

The user who creates a database. A Database Owner has control over all the database objects in that database. The login name for the Database Owner is "dbo".

datatype

Specifies what kind of information each column will hold, and how the data will be stored. Datatypes include *char*, *int*, *money*, and so on. Users can construct their own datatypes based on the SQL Server system datatypes.

datatype conversion function

A function which is used to convert expressions of one datatype into another datatype, whenever these conversions are not performed automatically by SQL Server.

datatype hierarchy

The hierarchy that determines the results of computations using values of different datatypes.

date function

A function that displays information about dates and times, or manipulates date or time values. The date functions include **getdate**, **datename**, **datepart**, **datediff**, and **dateadd**.

date part

Parts of a date, such as day, month, or year, recognized by the Transact-SQL date functions.

deadlock

A situation which arises when two users, each having a **lock** on one piece of data, attempt to acquire a lock on the other's piece of data. The SQL Server detects deadlocks, and kills one user's process.

default

The option chosen by the system when no other option is specified.

default clause

Specifies the default value for a column in the **create table** statement.

default database

The database that a user connects with when he or she logs in.

delimited identifiers

Object names enclosed in double quotes which avoid certain restrictions on object names.

demand lock

A demand lock prevents any more shared locks from being set on a data resource (table or data page). Any new shared lock request has to wait for the demand lock request to finish.

dependent

Data is logically dependent on other data when master data in one table must be kept synchronized with detail data in another table in order to protect the logical consistency of the database.

detail

Data that logically depends on data in another table. For example, in the *pubs2* database, the *salesdetail* table is a detail table. Each order in the *sales* table can have many corresponding entries in *salesdetail*. Each item in *salesdetail* is meaningless without a corresponding entry in the *sales* table.

dirty read

Occurs when one transaction modifies a row, and then a second transaction reads that row before the first transaction commits the change. If the first transaction rolls back the change, the information read by the second transaction becomes invalid.

disk allocation pieces

Disk allocation pieces are the groups of allocation units from which SQL Server constructs a new database file. The minimum size for a disk allocation piece is one **allocation unit**, or 256 2KB pages.

display precision

The number of significant binary digits offered by the default display format for *real* and *float* values. Internally, *real* and *float* values are stored with a precision less than or equal to that of the platform-specific datatypes on which they are built. For display purposes, Sybase *real* values have 9 digits of precision; Sybase *float* values, 17.

dynamic dump

A dump made while the database is active.

equijoin

A join based on equality.

error message

A message that SQL Server issues, usually to the user's terminal, when it detects an error condition.

error state number

The number attached to a SQL Server error message that allows unique identification of the line of SQL Server code at which the error was raised.

exclusive locks

Locks which prevent any other transaction from acquiring a lock until the original lock is released at the end of a transaction, always applied for update (**insert**, **update**, **delete**) operations.

execute cursor

A cursor which is a subset of Open Client cursors, whose result set is defined by a stored procedure which has a single **select** statement. The stored procedure can use parameters. The values of the parameters are sent through Open Client calls.

expression

A combination of one or more constants, literals, functions, column identifiers, or variables separated by operators that returns a single value. An expression can be arithmetic, relational, logical (Boolean), or a character string.

fatal errors

Errors with severity levels of 19 and above. They terminate the user's work session, so that it is necessary to log in again.

fetch

A fetch moves the current cursor position down the cursor result set. Also called a cursor fetch.

field

A data value that describes one characteristic of an entity. Also called a **column**.

FIPS flagger

Setting the FIPS (Federal Information Processing Standards) flagger requires that all nonstandard enhancements to the SQL language be flagged. FIPS recognizes SQL89 as the base standard.

foreign key

A key column in a table that logically depends on a **primary key** column in another table. Also, a column (or combination of columns) whose values are required to match a primary key in some other table.

fragment

When you allocate only a portion of the space on a device with **create** or **alter database**, that portion is called a fragment.

functions

See **built-in functions**.

global variable

System-defined variables that SQL Server updates on an ongoing basis. For example, *@@error* contains the last error number generated by the system.

guest

If the user name "guest" exists in the *sysusers* table of a database, any user with a valid SQL Server login can use that database, with limited privileges.

Halloween problem

An anomaly associated with cursor updates whereby a row seems to appear twice in the result set. This happens when the index key is updated by the client and the updated index row moves farther down in the result set.

hexadecimal string

A hexadecimal-encoded binary string that begins with the prefix 0x and can include the digits 0 through 9 and the upper- and lowercase letters A through F. The interpretation of hexadecimal strings is platform specific. For some systems, the first byte after the prefix is the most significant; for others, the last byte. For example, the string 0x0100 is interpreted as 1 on some systems and as 256 on others.

identifier

A string of characters used to identify a database object, such as a table name or column name.

IDENTITY column

A column that contains system-generated values that uniquely identify each row in a table. IDENTITY columns store unique numbers, such as invoice numbers or employee numbers, that are generated automatically by SQL Server. The value of the IDENTITY column uniquely identifies each row in a table.

implicit conversions

Datatype conversions that SQL Server automatically performs to compare datatypes.

inner query

Another name for a **subquery**.

int

A signed 32 bit integer value.

integrity constraints

Form a model to describe the database integrity in the **create table** statement. Database integrity has two complementary components: **validity**, which guarantees that all false information is excluded from the database, and **completeness**, which guarantees that all true information is included in the database.

intent lock

Indicates the intention to acquire a share or exclusive lock on a data page.

isolation level

Specifies the kinds of actions that are not permitted while the current transactions execute; also called "locking level." The SQL standard defines four levels of isolation for SQL transactions. Level 0 prevents other transactions from changing data already modified by an uncommitted transaction. Level 1 prevents **dirty reads**. Level 2 (not supported by SQL Server) also prevents **non-repeatable reads**. Level 3 prevents both types of reads and **phantoms**; it is equivalent to doing all queries with **holdlock**. The user controls the isolation level with the **set** option **transaction isolation level** or with the **at isolation** clause of **select** or **readtext**. The default is level 1.

join

A basic operation in a relational system which links the rows in two or more tables by comparing the values in specified columns.

key

A field used to identify a record, often used as the index field for a table.

key value

Any value that is indexed.

keyword

A word or phrase that is reserved for exclusive use by Transact-SQL. Also known as a **reserved word**.

keyword phrases

A set of keywords and parameters that tailor a Transact-SQL command to meet a particular need. Also called a **clause**.

language cursor

A cursor declared in SQL without using Open Client. As with SQL Server cursors, Open Client is completely unaware of the cursors and the results are sent back to the client in the same format as a normal **select**.

leaf level

The level of an index at which all key values appear in order. For SQL Server clustered indexes, the leaf level and the data level are the same. For nonclustered indexes, the last index level above the data level is the leaf level, since key values for all of the data rows appear there in sorted order.

livelock

A request for an **exclusive lock** that is repeatedly denied because a series of overlapping **shared locks** keeps interfering. SQL Server detects the situation after four denials, and refuses further shared locks.

local variables

User-defined variables defined with a **declare** statement.

locking

The process of restricting access to resources in a multiuser environment to maintain security and prevent concurrent access problems. SQL Server automatically applies locks to tables or pages.

locking level

See **isolation level**.

logical expression

An expression that evaluates to TRUE (1), FALSE (0), or UNKNOWN (NULL). Logical expressions are often used in control of flow statements, such as **if** or **while** conditions.

logical operators

The operators **and**, **or**, and **not**. All three can be used in **where** clauses. The operator **and** joins two or more conditions and returns results when all of the conditions are true; **or** connects two or more conditions and returns results when any of the conditions is true.

login

The name a user uses to log onto SQL Server. A login is valid if SQL Server has an entry for that user in the system table *syslogins*.

master database

Controls the user databases and the operation of SQL Server as a whole. Known as *master*, it keeps track of such things as user accounts, ongoing processes, and system error messages.

master table

A table that contains data on which data in another table logically depends. For example, in the *pubs2* database, the *sales* table is a master table. The *salesdetail* table holds detail data which depends on the master data in *sales*. The detail table typically has a foreign key that joins to the primary key of the master table.

master-detail relationship

A relationship between sets of data where one set of data logically depends on the other. For example, in the *pubs2* database, the *sales* table and *salesdetail* table have a master-detail relationship. See **detail** and **master table**.

message number

The number that uniquely identifies an error message.

modulo

An **arithmetic operator** represented by the percent (%) sign which gives the integer remainder after a division operation on two integers. For example, 21 % 9 = 3 because 21 divided by 9 equals 2 with a remainder of 3.

natural join

A **join** in which the values of the columns being joined are compared on the basis of equality, and all the columns in the tables are included in the results, except that only one of each pair of joined columns is included.

nested queries

select statements that contain one or more subqueries.

nested select statements

See **nested queries**.

nonclustered index

An **index** that stores key values and pointers to data. The **leaf level** points to data pages rather than containing the data itself.

nonrepeatable read

Occurs when one transaction reads a row and then a second transaction modifies that row. If the second transaction commits its change, subsequent reads by the first transaction yield different results than the original read.

normalization rules

The standard rules of database design in a relational database management system.

not-equal join

A join on the basis of inequality.

null

Having no explicitly assigned value. NULL is not equivalent to zero, or to blank. A value of NULL is not considered to be greater than, less than, or equivalent to any other value, including another value of NULL.

numeric expression

An expression that contains only numeric values and returns a single numeric value. In Transact-SQL, the operands can be of any SQL Server numeric datatype. They can be functions, variables, parameters, or they can be other arithmetic expressions. Synonymous with **arithmetic expression**.

object permissions

Permissions that regulate the use of certain commands (data modification commands, plus **select**, **truncate table** and **exec**ute) to specific tables, views or columns. See also **command permissions**.

objects

See **database objects**.

operating system

A group of programs that translates your commands to the computer, so that you can perform such tasks as creating files, running programs, and printing documents.

operators

Symbols that act on two values to produce a third. See comparison operators, logical operators, or arithmetic operators.

outer join

A join in which both matching and nonmatching rows are returned. The operators *= and =* are used to indicate that all the rows in the first or second tables should be returned, regardless of whether or not there is a match on the join column.

outer query

Another name for the principal query in a statement containing a subquery.

parameter

An argument to a stored procedure.

permission

The authority to perform certain actions on certain database objects or to run certain commands.

phantoms

Occur when one transaction reads a set of rows that satisfy a search condition, and then a second transaction modifies the data (through an **insert**, **delete**, **update**, and so on). If the first transaction repeats the read with the same search conditions, it obtains a different set of rows.

precision

The maximum number of decimal digits that can be stored by *numeric* and *decimal* datatypes. The precision includes **all** digits, both to the right and to the left of the decimal point.

primary key

The column or columns whose values uniquely identify a row in a table.

primary key constraint

A primary key constraint is a **unique** constraint which does not permit null values for the component key columns. There can only be one primary key constraint per table. The primary key constraint creates a unique index on the specified columns to enforce this data integrity.

privilege

The authority to perform certain actions on certain database objects or to run certain commands. Synonymous to **permission**.

projection

One of the basic query operations in a relational system. A projection is a subset of the columns in a table.

qualified

The name of a database object can be qualified, or preceded, by the name of the database and the object owner.

query

1. A request for the retrieval of data with a **select** statement.

2. Any SQL statement that manipulates data.

referential integrity

The rules governing data consistency, specifically the relationships among the primary keys and foreign keys of different tables. SQL Server addresses referential integrity with user-defined triggers.

referential integrity constraint

Referential integrity constraints require that data inserted into a "referencing" table which defines the constraint must have matching values in a "referenced" table. You cannot delete rows or update column values from a referenced table that match values in a referencing table. Also, you cannot drop the referenced table until the referencing table is dropped or the referential integrity constraint is removed.

relational expression

A type of Boolean or logical expression of the form:

```
arith_expression
relational_operator arith_expression
```

In Transact-SQL, a relational expression can return TRUE, FALSE or UNKNOWN. The results can evaluate to UNKNOWN if one or both of the expressions evaluates to NULL.

relational operator

An operator that compares two operands and yields a truth value, such as "5 <7" (TRUE), "ABC" = "ABCD" (FALSE) or "@value > NULL" (UNKNOWN).

restriction

A subset of the rows in a table. Also called a **selection**, it is one of the basic query operations in a relational system.

return status

A value that indicates that the procedure completed successfully or indicates the reason for failure.

roles

Provide individual accountability for users performing system administration and security-related tasks in SQL Server. The System Administrator, System Security Officer, and Operator roles can be granted to individual server login accounts.

rollback transaction

A Transact-SQL statement used with a user-defined transaction (before a **commit transaction** has been received) that cancels the transaction and undoes any changes that were made to the database.

row

A set of related **columns** that describes a specific entity. Also called a **record**.

row aggregate function

Functions (**sum**, **avg**, **min**, **max**, and **count**) that generate a new row for summary data when used with **compute** in a **select** statement.

rule

A specification that controls what data may be entered in a particular column, or in a column of a particular user-defined datatype.

savepoint

A marker that the user puts inside a **user-defined transaction**. The user can later use the **rollback transaction** command with the savepoint name to cancel any commands up to the savepoint, or **commit transaction** to actually complete the commands.

scalar aggregate

An aggregate function that produces a single value from a **select** statement that does not include a **group by** clause. This is true whether the aggregate function is operating on all the rows in a table or on a subset of rows defined by a **where** clause. See also **vector aggregate**.

scale

The maximum number of digits that can be stored to the right of the decimal point by a *numeric* or *decimal* datatype. The scale must be less than or equal to the **precision**.

schema

Consists of the collection of objects associated with a particular schema name and schema authorization identifier. The objects are tables, views, domains, constraints, assertions, privileges, and so on. A schema is created by a **create schema** statement.

select list

The columns specified in the main clause of a **select** statement. In a dependent view, the target list must be maintained in all underlying views if the dependent view is to remain valid.

selection

A subset of the rows in a table. Also called a restriction, it is one of the basic query operations in a relational system.

self-join

A join used for comparing values within a column of a table. Since this operation involves a join of a table with itself, you must give the table two temporary names, or **correlation names**, which are then used to qualify the column names in the rest of the query.

server cursor

A cursor declared inside a stored procedure. The client executing the stored procedure is not aware of the presence of these cursors. Results returned to the client for a **fetch** appear exactly the same as the results from a normal **select**.

server user ID

The ID number by which a user is known to SQL Server.

severity level number

The severity of an error condition: errors with severity levels of 19 and above are fatal errors.

shared lock

A **lock** created by non-update (“read”) operations. Other users may read the data concurrently, but no transaction can acquire an **exclusive** lock on the data until all the shared locks have been released.

statement

Begins with a **keyword** that names the basic operation or command to be performed.

statement block

A series of Transact-SQL statements enclosed between the keywords **begin** and **end** so that they are treated as a unit.

stored procedure

A collection of SQL statements and optional control-of-flow statements stored under a name. SQL Server-supplied stored procedures are called **system procedures**.

string function

A function that operates on strings of characters or binary data. **substring** and **charindex** are Transact-SQL string functions.

subquery

A **select** statement that is nested inside another **select**, **insert**, **update** or **delete** statement, or inside another subquery.

System Administrator

A user authorized to handle SQL Server system administration, including creating user accounts, assigning permissions, and creating new databases.

system databases

The databases on a newly installed SQL Server: *master*, which controls user databases and the operation of the SQL Server; *tempdb*, used for temporary tables; *model*, used as a template to create new user databases; and *sybsystemprocs*, which stores the system procedures.

system function

A function that returns special information from the database, particularly from the system tables.

system procedures

Stored procedures that SQL Server supplies for use in system administration. These procedures are provided as shortcuts for retrieving information from the system tables, or mechanisms for accomplishing database administration and other tasks that involve updating system tables.

system table

One of the data dictionary tables. The system tables keep track of information about the SQL Server as a whole and about each user database. The Master Database contains some system tables that are not in user databases.

table

A collection of rows (records) that have associated columns (fields). The logical equivalent of a database file.

table-level constraint

Limits values on more than one column of a table. Enter table-level constraints as separate comma-delimited clausesin the **create** statement. You must declare constraints that operate on more than one column as table-level constraints.

theta join

Joins which use the comparison operators as the join condition. Comparison operators include equal (=), not equal (!=), greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=).

transaction

A mechanism for ensuring that a set of actions is treated as a single unit of work. See also **user-defined transaction**.

transaction log

A system table (*syslogs*) in which all changes to the database are recorded.

trigger

A special form of **stored procedure** that goes into effect when a user gives a change command such as **insert**, **delete**, or **update** to a specified table or column. Triggers are often used to enforce referential integrity.

trigger actions

The action for which a trigger is specified.

trigger conditions

The conditions that cause a trigger to take effect.

trigger table

The table to which a trigger is attached.

trigger test tables

When a data modification affects a key column, triggers compare the new column values to related keys by using temporary work tables called trigger test tables.

unique constraint

A constraint requiring that all non-null values in the specified columns must be unique. No two rows in the table are allowed to have the same value in the specified column. The **unique** constraint creates a unique index on the specified columns to enforce this data integrity.

unique indexes

Indexes which do not permit any two rows in the specified columns to have the same value. SQL Server checks for duplicate values when you create the index (if data already exists) and each time data is added.

update

An addition, deletion, or change to data, involving the **insert**, **delete**, **truncate table**, or **update** statements.

update locks

Locks which ensure that only one operation can change data on a page. Other transactions are allowed to read the data through shared locks. SQL Server applies update locks when an **update** or **delete** operation begins.

user-defined datatype

A definition of the type of data a column can contain, that is created by the user. These datatypes are defined in terms of the existing system datatypes. Rules and defaults can be bound to user-defined datatypes (but not to system datatypes).

user-defined transaction

See **transaction**.

variable

An entity that is assigned a value. SQL Server has two kinds of variables, called **local variables** and **global variables**.

vector aggregate

A value that results from using an aggregate function with a **group by** clause. See also **scalar aggregate**.

view

An alternative way of looking at the data in one or more tables. Usually created as a subset of columns from one or more tables.

view resolution

In queries that involve a view, the process of verifying the validity of database objects in the query, and combining the query and the stored definition of the view.

wildcard

Special character used with the Transact-SQL **like** keyword that can stand for one (the underscore, _) or any number of (the percent sign, %) characters in pattern-matching.

Index

The index is divided into two sections:

- Symbols

  Indexes each of the symbols used in Sybase SQL Server documentation.

- Subjects

  Indexes subjects alphabetically.

Page numbers in **bold** are primary references.

Symbols

Numerics

A

B

C

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y