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

Adaptive Server Enterprise Version 12

Document ID: 32300-01-1200-01

Last Revised: October, 1999

Principal author: Server Publications Group

Document ID: 32300-01-1200

This publication pertains to Adaptive Server Enterprise Version 12 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

Sybase, the Sybase logo, APT-FORMS, Certified SYBASE Professional, Data Workbench, First Impression, InfoMaker, PowerBuilder, Powersoft, Replication Server, S-Designor, SQL Advantage, SQL Debug, SQL SMART, SQL Solutions, Transact-SQL, VisualWriter, and VQL are registered trademarks of Sybase, Inc. Adaptable Windowing Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Monitor, ADA Workbench, AnswerBase, Application Manager, AppModeler, APT-Build, APT-Edit, APT-Execute, APT-Library, APT-Translator, APT Workbench, Backup Server, BayCam, Bit-Wise, ClearConnect, Client-Library, Client Services, CodeBank, Column Design, Connection Manager, DataArchitect, Database Analyzer, DataExpress, Data Pipeline, DataWindow, DB-Library, dbQ, Developers Workbench, DirectConnect, Distribution Agent, Distribution Director, Dynamo, Embedded SQL, EMS, Enterprise Client / Server, Enterprise Connect, Enterprise Manager, Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, EWA, Formula One, Gateway Manager, GeoPoint, ImpactNow, InformationConnect, InstaHelp, InternetBuilder, iScript, Jaguar CTS, jConnect for JDBC, KnowledgeBase, Logical Memory Manager, MainframeConnect, Maintenance Express, MAP, MDI Access Server, MDI Database Gateway, media.splash, MetaWorks, MethodSet, Net-Gateway, NetImpact, Net-Library, ObjectConnect, ObjectCycle, OmniConnect, OmniSQL Access Module, OmniSQL Toolkit, Open Client, Open ClientConnect, Open Client / Server, Open Client / Server Interfaces, Open Gateway, Open Server, Open

Restricted Rights

Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608.

Table of Contents

About This Book

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. Using Extended Stored Procedures

16. Triggers: Enforcing Referential Integrity

17. Cursors: Accessing Data Row by Row

18. Transactions: Maintaining Data Consistency and Recovery

19. Locking Commands and Options

A. The *pubs2* Database

B. The *pubs3* Database

Index

List of Figures

List of Tables

About This Book

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® Adaptive Server™ Enterprise database management systems who are unfamiliar with SQL can consider 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 advanced topics.

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 Concepts 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 Concepts" 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 10–18. 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 are based on the *pubs2* and, where noted, *pubs3* sample databases. 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* and *pubs3*. For a complete description of these databases, see Appendix A, "The pubs2 Database," and Appendix B, "The pubs3 Database."

You can use Transact-SQL with the Adaptive Server stand-alone program **isql**. The **isql** program is a utility program called directly from the operating system.

Adaptive Server Enterprise Documents

The following documents comprise the Sybase Adaptive Server Enterprise documentation:

- The *Release Bulletin* for your platform – contains last-minute information that was too late to be included in the books.

  A more recent version of the *Release Bulletin* may be available on the World Wide Web. To check for critical product or document information that was added after the release of the product CD, use the Sybase Technical Library.
- The Adaptive Server installation documentation for your platform – describes installation, upgrade, and configuration procedures for all Adaptive Server and related Sybase products.
- *What's New in Adaptive Server Enterprise?* – describes the new features in Adaptive Server version 12, the system changes added to support those features, and the changes that may affect your existing applications.
- *Transact-SQL User's Guide* – documents Transact-SQL, Sybase's enhanced version of the relational database language. This manual serves as a textbook for beginning users of the database management system. This manual also contains descriptions of the *pubs2* and *pubs3* sample databases.
- *System Administration Guide* – provides in-depth information about administering servers and databases. This manual includes instructions and guidelines for managing physical resources,

security, user and system databases, and specifying character conversion, international language, and sort order settings.

- *Adaptive Server Reference Manual* – contains detailed information about all Transact-SQL commands, functions, procedures, and datatypes. This manual also contains a list of the Transact-SQL reserved words and definitions of system tables.
- *Performance and Tuning Guide* – explains how to tune Adaptive Server for maximum performance. This manual includes information about database design issues that affect performance, query optimization, how to tune Adaptive Server for very large databases, disk and cache issues, and the effects of locking and cursors on performance.
- The *Utility Programs* manual for your platform – documents the Adaptive Server utility programs, such as **isql** and **bcp**, which are executed at the operating system level.
- *Error Messages* and *Troubleshooting Guide* – explains how to resolve frequently occurring error messages and describes solutions to system problems frequently encountered by users.
- *Component Integration Services User's Guide* – explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.
- *Java in Adaptive Server Enterprise* – describes how to install and use Java classes as datatypes and user-defined functions in the Adaptive Server database.
- *Using Sybase Failover in a High Availability System* – provides instructions for using Sybase's Failover to configure an Adaptive Server as a companion server in a high availability system.
- *Using Adaptive Server Distributed Transaction Management Features* – explains how to configure, use, and troubleshoot Adaptive Server DTM Features in distributed transaction processing environments.
- *XA Interface Integration Guide for CICS, Encina, and TUXEDO* provides instructions for using Sybase's DTM XA Interface with X/Open XA transaction managers.
- *Adaptive Server Glossary* – defines technical terms used in the Adaptive Server documentation.

Other Sources of Information

Use the SyBooks™ and SyBooks-on-the-Web online resources to learn more about your product:

- SyBooks documentation is on the CD that comes with your software. The DynaText browser, also included on the CD, allows you to access technical information about your product in an easy-to-use format.

  Refer to *Installing SyBooks* in your documentation package for instructions on installing and starting SyBooks.

- SyBooks-on-the-Web is an HTML version of SyBooks that you can access using a standard Web browser.

  To use SyBooks-on-the-Web, go to http://www.sybase.com, and choose Documentation.

Conventions

The following sections describe 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.

Font and Syntax Conventions

Table 1 shows the conventions for syntax statements that appear in this manual::

Table 1: Font and syntax conventions for this manual

Element	Example
Command names, command option names, utility names, utility flags, and other keywords are **bold**.	**select** **sp_configure**

Table 1: Font and syntax conventions for this manual (continued)

Element	Example
Database names, datatypes, file names and path names are in *italics*.	*master* database
Variables, or words that stand for values that you fill in, are in *italics*.	`select` *column_name* `from` *table_name* `where` *search_conditions*
Parentheses are to be typed as part of the command.	`compute` *row_aggregate* `(`*column_name*`)`
Curly braces indicate that you must choose at least one of the enclosed options. Do not type the braces.	`{cash, check, credit}`
Brackets mean choosing one or more of the enclosed options is optional. Do not type the brackets.	`[anchovies]`
The vertical bar means you may select only one of the options shown.	`{die_on_your_feet | live_on_your_knees | live_on_your_feet}`
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.	`[extra_cheese, avocados, sour_cream]`

An ellipsis (...) means that you can **repeat** the last unit as many times as you like.	`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: 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.

- Syntax statements (displaying the syntax and all options for a command) appear as follows:

```
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. Italic font shows user-supplied words.

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

  ```
  select * from publishers
  ```

- Examples of output from the computer appear as follows:

  ```
  0736    New Age Books            Boston       MA
  0877    Binnet & Hardley         Washington   DC
  1389    Algodata Infosystems     Berkeley     CA
  ```

Case

In this manual, most of the examples are in lowercase. However, you can disregard case when typing Transact-SQL keywords. For example, **SELECT**, **Select**, and **select** are the same.

Adaptive Server's sensitivity to the case of database objects, such as table names, depends on the sort order installed on Adaptive Server. You can change case sensitivity for single-byte character sets by reconfiguring the Adaptive Server sort order. See the *System Administration Guide* for more information.

Expressions

Adaptive Server syntax statements use the following types of expressions.

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

Table 2: Types of expressions used in syntax statements (continued)

Usage	Definition
binary_expression	An expression that returns a single *binary* or *varbinary* value

If You Need Help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

1 Introduction

This chapter discusses:

Overview of Adaptive Server and Its Components

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 provides important extra capabilities and functions, such as summary calculations, stored procedures (predefined SQL statements), and error handling.

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.

➤ ***Note***

If Java is enabled on your Adaptive Server, you can install and use Java classes in the database. You can invoke Java operations and store Java classes using standard Transact-SQL commands. Refer to *Java in Adaptive Server Enterprise* for a complete description of this feature.

Queries, Data Modification, and Commands

In this manual, **query** means a request for the retrieval of data, using the **select** command. For example, the following query asks for a listing of authors who live in the state of California:

```
select au_lname, city, state
from authors
where state = "CA"
```

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:

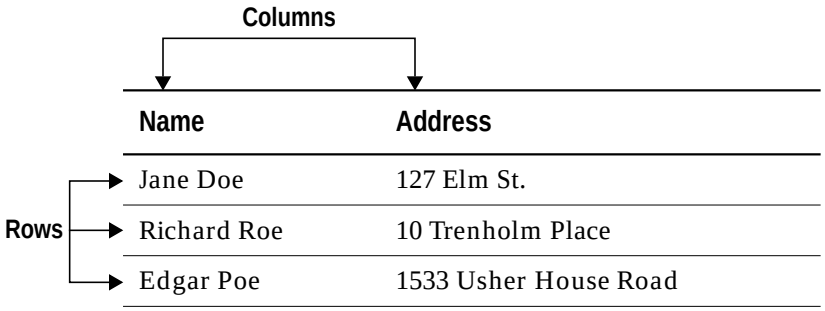
```
drop table authors
```

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.



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, suppose you have one table containing information about authors, including the columns *au_id* (author identification number) and *au_lname* (author's last name). A second table contains 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.

Compiled Objects

Adaptive Server uses **compiled objects** to contain vital information about each database and to help you access and manipulate data. A compiled object is any object that requires entries in the *sysprocedures* table, including:

- Check constraints

- Defaults
- Rules
- Stored procedures
- Extended stored procedures
- Triggers
- Views

Compiled objects are created from **source text**, the SQL statements that describe and define the compiled object. When a compiled object is created, Adaptive Server:

1. Parses the source text, catching any syntactic errors, to generate a parsed tree.
2. Normalizes the parsed tree to create a normalized tree, which represents the user statements in a binary tree format. This is the compiled object.
3. Stores the compiled object in the *sysprocedures* table.
4. Stores the source text in the *syscomments* table.

Saving Source Text

In Adaptive Server release 11.5 and in SQL Server releases prior to that release, the source text was saved in *syscomments* only so that it could be returned to a user who executed **sp_helptext**. Because this was the only purpose of saving the text, users often deleted the source text from the *syscomments* table to save disk space and to remove confidential information from this public area. However, you should not delete the source text because this may pose a problem for future upgrades of Adaptive Server. If you have removed the source text from *syscomments*, use the procedures in this section to restore the source text.

Restoring Source Text

If a compiled object does not have matching source text in the *syscomments* table, you can restore the source text to *syscomments* using any of the following methods:

- Load the source text from a backup.
- Recreate the source text manually
- Reinstall the application that created the compiled object

Verifying and Encrypting Source Text

Adaptive Server release 11.5 provides functionality that allows you to verify the existence of source text, and encrypt the text if you choose. Use the following commands when you are working with source text:

- **sp_checkresource** – verifies that source text is present in *syscomments* for each compiled object.
- **sp_hidetext** – encrypts the source text of a compiled object in the *syscomments* table to prevent casual viewing.
- **sp_helptext** – displays the source text if it is present in *syscomments*, or notifies you of missing source text.
- **dbcc checkcatalog** – notifies you of missing source text.

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

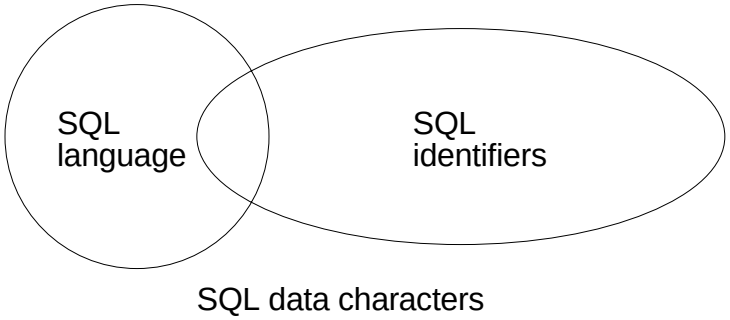


Figure 1-1: Characters used for different parts of SQL statements

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 Adaptive 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 Adaptive 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 Adaptive Server.

The @ sign or _(underscore character) can also be used. The @ sign as the first character of an identifier indicates a **local variable**.

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

After the first character, identifiers can include characters declared as alphabetic, numeric, or the character $, #, @, _, ¥ (yen), or £ (pound sterling). However, you cannot use two @@ symbols together at the beginning of a named object, as in "*@@myobject*." This naming convention is reserved for **global variables**, which are system-defined variables that Adaptive Server updates on an ongoing basis.

The case sensitivity of Adaptive 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 create only one of these objects, but you can use any of these combinations of case to refer to 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 *Adaptive Server Reference Manual*.

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

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

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

Using Multibyte Character Sets

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.

Although Hankaku Katakana characters are legal in identifiers on Japanese systems, they are not recommended for use in heterogeneous systems. These characters cannot be converted between the EUC-JIS and Shift-JIS character sets.

The same is true for some 8-bit European characters. For example, the character "Œ," the OE ligature, is part of the Macintosh character set (codepoint 0xCE). This character does not exist in the ISO 8859-1 (iso_1) character set. If "Œ" exists in data being converted from the Macintosh to the ISO 8859-1 character set, it causes a conversion error.

If an object identifier contains a character that cannot be converted, the client loses direct access to that object.

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 Adaptive 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 with bcp, may not be supported by all front-end products, and can produce unexpected results when used with system procedures.

While the **quoted_identifier** option is set **on**, do not use double quotes around character or date strings; use single quotes instead. Delimiting these strings with double quotes causes Adaptive Server to treat them as identifiers. For example, to insert a character string into *col1* of *1one*, use:

```
insert "1one"(col1) values ('abc')
```

not:

```
insert "1one"(col1) values ("abc")
```

To insert a single quote into a column, use two consecutive single quotation marks. For example, to insert the characters "a'b" into *col1*, use:

```
insert "1one"(col1) values('a''b')
```

Uniqueness and Qualification 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 Adaptive 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 have already used in the same database, Adaptive 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 of the **set** command is **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, Adaptive 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 Adaptive 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
```

You must include the starting element, in this case, *database*, particularly if you are using this syntax when creating tables. If you omit the starting element, you could, for example, create a table named *..mytable*. This naming convention prevents you from performing certain actions on such a table, such as cursor updates.

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.sa.publishers.city
from pubs2..publishers

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

Identifying Remote Servers

You can execute stored procedures on a remote Adaptive 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, Adaptive 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 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.
`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 *System Administration Guide* for information on setting up Adaptive Server for remote access. A remote server name (GATEWAY in the previous example) must match a server name in

your local Adaptive Server's *interfaces* file. If the server name in *interfaces* is in uppercase letters, you must use it in uppercase letters in the remote procedure call.

Expressions in Adaptive Server

An **expression** is a combination of one or more constants, literals, functions, column identifiers, and variables, separated by operators, that returns a single value. Expressions can be of several types, including **arithmetic**, **relational**, **logical** (or **Boolean**), and **character string**. In some Transact-SQL clauses, a subquery can be used in an expression. A case expression can be used in an expression.

Use parentheses to group the elements in an expression. When "*expression*" is given as a variable in a syntax statement, a simple expression is assumed. "Logical expression" is specified when only a logical expression is acceptable.

Arithmetic and Character Expressions

The general pattern for arithmetic and character expressions is:

```
{constant | column_name | function | (subquery)
   | (case_expression)}
      [{arithmetic_operator | bitwise_operator |
       string_operator | comparison_operator }
   {constant | column_name | function | (subquery)
   | case_expression}]...
```

Operator Precedence

Operators have the following precedence levels, where 1 is the highest level and 6 is the lowest:

1. unary (single argument) **- + ~**
2. *** / %**
3. binary (two argument) **+ - & | ^**
4. **not**
5. **and**
6. **or**

When all operators in an expression are of the same level, the order of execution is left to right. You can change the order of execution

with parentheses—the most deeply nested expression is handled first.

Arithmetic Operators

Adaptive Server uses the following arithmetic operators:

Table 1-2: Arithmetic operators

Operator	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulo (Transact-SQL extension)

Addition, subtraction, division, and multiplication can be used on exact numeric, approximate numeric, and *money* type columns.

The modulo operator cannot be used on *smallmoney, money, float* or *real* columns. Modulo finds the integer remainder after a division involving two whole numbers. For example, 21 % 11 = 10 because 21 divided by 11 equals 1 with a remainder of 10.

When you perform arithmetic operations on mixed datatypes (for example, *float* and *int*) Adaptive Server follows specific rules for determining the type of the result. See Chapter 6, "Using and Creating Datatypes," for more information.

Bitwise Operators

The bitwise operators are a Transact-SQL extension for use with integer type data. These operators convert each integer operand into its binary representation and then evaluate the operands column by column. A value of 1 corresponds to true; a value of 0 corresponds to false.

The following tables summarize the results for operands of 0 and 1. If either operand is NULL, the bitwise operator returns NULL:

Table 1-3: Truth tables for bitwise operations

& (and)	1	0
1	1	0
0	0	0

| (or)	1	0

Table 1-3: Truth tables for bitwise operations (continued)

1	1	1
0	1	0

^ (exclusive or)	1	0
1	0	1
0	1	0

~ (not)	
1	FALSE
0	0

The following examples use two *tinyint* arguments, A = 170 (10101010 in binary form) and B = 75 (01001011 in binary form.

Table 1-4: Examples of bitwise operations

Operation	Binary Form	Result	Explanation
(A & B)	10101010 01001011 ---------- 00001010	10	Result column equals 1 if both A and B are 1. Otherwise, result column equals 0.
(A | B)	10101010 01001011 ---------- 11101011	235	Result column equals 1 if either A or B, or both, is 1. Otherwise, result column equals 0
(A ^ B)	10101010 01001011 ---------- 11100001	225	Result column equals 1 if either A or B, but not both, is 1
(~A)	10101010 ---------- 01010101	85	All 1's are changed to 0's and all 0's to 1's

The String Concatenation Operator

The string operator **+** can concatenate two or more character or binary expressions. For example:

1.
```
select Name = (au_lname + ", " + au_fname)
from authors
```

   Displays author names under the column heading *Name* in last-name first-name order, with a comma after the last name; for example, “Bennett, Abraham.”

2.
```
select "abc" + "" + "def"
```

   Returns the string “abc def”. The empty string is interpreted as a single space in all *char, varchar, nchar, nvarchar,* and *text* concatenation, and in *varchar* insert and assignment statements.

When concatenating non-character, non-binary expressions, use **convert**:

```
select "The date is " +
    convert(varchar(12), getdate())
```

The Comparison Operators

Adaptive Server uses the following comparison operators:

Table 1-5: Comparison operators

Operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to
!=	Not equal to (Transact-SQL extension)
!>	Not greater than (Transact-SQL extension)
!<	Not less than (Transact-SQL extension)

In comparing character data, **<** means closer to the beginning of the server’s sort order and **>** means closer to the end of the sort order. Uppercase and lowercase letters are equal in a sort order that is not case sensitive. Use **sp_helpsort** to see the sort order for your Adaptive Server. Trailing blanks are ignored for comparison purposes. So, for example, “Dirk” is the same as “Dirk ”.

In comparing dates, **<** means earlier and **>** means later.

Put single or double quotes around all character and *datetime* data used with a comparison operator:

```
= "Bennet"
> "May 22 1947"
```

Nonstandard Operators

The following operators are Transact-SQL extensions:

- Modulo operator: **%**
- Negative comparison operators: **!>**, **!<**, **!=**
- Bitwise operators: **~**, **^**, **|**, **&**
- Join operators: ***=** and **=***

Comparing Character Expressions

Adaptive Server treats character constant expressions as *varchar*. If they are compared with non-*varchar* variables or column data, the datatype precedence rules are used in the comparison (that is, the datatype with lower precedence is converted to the datatype with higher precedence). If implicit datatype conversion is not supported, you must use the **convert** function.

Comparison of a *char* expression to a *varchar* expression follows the datatype precedence rule; the "lower" datatype is converted to the "higher" datatype. All *varchar* expressions are converted to *char* (that is, trailing blanks are appended) for the comparison.

Using the Empty String

The empty string ("") or ('') is interpreted as a single blank in **insert** or assignment statements on *varchar* data. In concatenation of *varchar, char, nchar, nvarchar* data, the empty string is interpreted as a single space; for example:

```
"abc" + "" + "def"
```

is stored as "abc def". The empty string is never evaluated as NULL.

Including Quotation Marks in Character Expressions

There are two ways to specify literal quotes within a *char* or *varchar* entry. The first method is to double the quotes. For example, if you begin a character entry with a single quote, but you want 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). 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?"'
```

Relational and Logical Expressions

A logical expression or relational expression returns TRUE, FALSE, or UNKNOWN. The general patterns are:

```
expression comparison_operator [any | all] expression

expression [not] in expression

[not] exists expression

expression [not] between expression and expression

expression [not] like "match_string"
   [escape "escape_character"]

not expression like "match_string"
   [escape "escape_character"]

expression is [not] null

not logical_expression

logical_expression {and | or} logical_expression
```

Using *any*, *all*, and *in*

any is used with **<**, **>**, or **=** and a subquery. It returns results when any value retrieved in the subquery matches the value in the **where** or **having** clause of the outer statement. **all** is used with **<** or **>** and a subquery. It returns results when all values retrieved in the subquery are less than (**<**) or greater than (**>**) the value in the **where** or **having** clause of the outer statement. See Chapter 5, "Subqueries: Using Queries Within Other Queries," for more information.

in returns results when any value returned by the second expression matches the value in the first expression. The second expression must be a subquery or a list of values enclosed in parentheses. **in** is equivalent to **= any**.

Connecting Expressions with *and* and *or*

and connects two expressions and returns results when both are true. **or** connects two or more conditions and returns results when either of the conditions is true.

When more than one logical operator is used in a statement, **and** is evaluated before **or**. You can change the order of execution with parentheses.

Table 1-6 shows the results of logical operations, including those that involve null values:

Table 1-6: Truth tables for logical expressions

and	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
NULL	UNKNOWN	FALSE	UNKNOWN

or	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
NULL	TRUE	UNKNOWN	UNKNOWN

not	
TRUE	FALSE
FALSE	TRUE
NULL	UNKNOWN

The result UNKNOWN indicates that one or more of the expressions evaluates to NULL, and that the result of the operation cannot be determined to be either TRUE or FALSE.

Transact-SQL Extensions

Transact-SQL enhances the power of SQL and minimizes 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 Transact-SQL **compute** clause extension 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. A number of predefined global variables are supplied by the system.

Transact-SQL also supports **case** expressions, which includes the keywords **case**, **when**, **then**, **coalesce**, **nullif**. **case** expressions replace the **if** statements of standard SQL. **case** expressions are allowed anywhere a value expression is used.

Stored Procedures

A stored procedure is a named collection of SQL statements and optional control-of-flow statements stored under a name. 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.

Adaptive Server-supplied stored procedures, called **system procedures,** are provided for your use in Adaptive 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 *Adaptive 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.

Extended Stored Procedures

An **extended stored procedure** (ESP) has the interface of a stored procedure, but instead of containing SQL statements and control-of-flow statements, it executes procedural language code that has been compiled into a dynamic link library (DLL).

The procedural language in which an ESP function is written can be any language capable of calling C language functions and manipulating C datatypes.

ESPs allow Adaptive Server to perform a task outside the RDBMS in response to an event occurring within the database. For example, an email notification or network-wide broadcast could be sent in response to an event occurring within the RDBMS.

There are some Adaptive Server-supplied ESPs, called **system extended stored procedures**. One of these, **xp_cmdshell**, allows you to execute an operating system command from within Adaptive Server. Chapter 15, "Using Extended Stored Procedures," describes ESPs in detail. See also the *Adaptive Server Reference Manual* for more information on the system extended stored procedures.

ESPs are implemented by an Open Server application called XP Server, which runs on the same machine as Adaptive Server. Adaptive Server and XP Server communicate through remote procedure calls. XP Server is automatically installed when Adaptive Server is installed.

Triggers

A **trigger** is a stored procedure that instructs 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 also 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.

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

Defaults and Rules

Transact-SQL provides keywords for maintaining 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. All **set** options except **showplan** and **char_convert** take effect immediately.

The following paragraphs list the available **set** options; for more information, refer to the *Adaptive Server Reference Manual*.

- The **parseonly, noexec, prefetch, showplan, rowcount, nocount** and **tablecount** options control the way a query is executed. The **statistics** options display performance statistics after each query. **flushmessage** determines when Adaptive Server returns messages to the user. See the *Performance and Tuning Guide* for more information.
- **arithabort** determines whether Adaptive Server aborts queries with arithmetic overflow and numeric truncation errors. **arithignore** determines whether Adaptive Server prints a warning message if a query results in an arithmetic overflow. For more information, see "Arithmetic Errors" on page 1-27.
- **offsets** and **procid** are used in DB-Library™ to interpret results from Adaptive Server.
- **datefirst**, **dateformat**, and **language** affect date functions, date order, and message display. **char_convert** controls character set conversion between Adaptive Server and a client.
- **textsize** controls the size of *text* or *image* data returned with a **select** statement. See "Text Functions Used for text and image Data" on page 10-17 for more information.
- **cursor rows** and **close on endtran** affect the way Adaptive Server handles cursors. See "Fetching Data Rows Using Cursors" on page 17-13 for more information.
- **identity_insert** allows or prohibits inserts that affect a table's IDENTITY column. See "Gaps Due to Insertions, Deletions, Identity Grab Size, and Rollbacks" on page 7-30 for more information.
- **chained** and **transaction isolation level** control how Adaptive Server handles transactions. For more information, see "Selecting the Transaction Mode and Isolation Level" on page 18-13.
- **self_recursion** allows Adaptive Server to handle triggers that cause themselves to fire. For more information, see "Trigger Self-Recursion" on page 16-25.
- **ansinull**, **ansi_permissions**, and **fipsflagger** control whether Adaptive Server flags the use of nonstandard SQL. **string_rtruncation** controls whether Adaptive Server to raise an exception error when

truncating a *char* or *nchar* string. See "Compliance to ANSI Standards" on page 1-24.

- **quoted_identifier** controls whether Adaptive Server treats character strings enclosed in double quotes as identifiers. See "Delimited Identifiers" on page 1-8 for more information.
- **role** controls the roles granted to you. For information about roles, see the *Security Features User's Guide*.

Additional Adaptive Server Extensions to SQL

Other unique or unusual features of Transact-SQL include:

- The following extensions to SQL search conditions: modulo operator (**%**), negative comparison operators (**!>**, **!<**, and **!=**), bitwise operators (**–**, **^**, **|**, and **&**), join operators (***=** and **=***), wildcard characters ([ ] and -), and the **not** operator (**^**). See Chapter 2, "Queries: Selecting Data from a Table."
- 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 Adaptive Server-supplied datatypes. See Chapter 6, "Using and Creating Datatypes," 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, "Adding, Changing, and Deleting Data."
- The ability to remove data based on data in other tables using the join in a **delete** statement. See Chapter 8, "Adding, Changing, and Deleting Data."

- 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, "Adding, Changing, and Deleting Data."
- IDENTITY columns, which provide system-generated values that uniquely identify each row within a table. See Chapter 8, "Adding, Changing, and Deleting Data."
- Updates and selections through views. Unlike most other versions of SQL, Transact-SQL places no restrictions on retrieving data through views, and 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, "Creating Indexes on Tables."
- Bitwise operators for use with *integer* and *bit* type columns. See "Bitwise Operators" on page 2-8 and Chapter 6, "Using and Creating Datatypes."
- Support for *text* and *image* datatypes. See Chapter 6, "Using and Creating Datatypes."
- The ability to gain access to both Sybase and non-Sybase databases. With the Component Integration Services feature, you can accomplish the following types of actions between tables in remote, heterogeneous servers: access remote tables as if they were local, perform joins, transfer data between tables, maintain referential integrity, provide applications such as PowerBuilder® with transparent access to heterogeneous data, and use native remote server capabilities. For information on using Component Integration Services, see the *Component Integration Services User's Guide*.

Compliance to ANSI 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 / Adaptive 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 *Adaptive Server Reference Manual.*

Table 1-7: 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, Adaptive 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. This option does not disable the extensions. Processing completes when you issue the non-ANSI SQL command.

Chained Transactions and Isolation Levels

Adaptive Server 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 **set transaction isolation level** option controls transaction isolation levels. See Chapter 18, "Transactions: Maintaining Data Consistency and Recovery," for more information.

Identifiers

To be entry-level SQL92 compliant, identifiers must not:

- Begin with a pound sign (#)
- Have more than 18 characters
- Contain lowercase letters

Delimited Identifiers

Adaptive Server 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 **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 "/*" and "*/", and can be nested. Transact-SQL also supports SQL standard-style

comments, which consist of any string beginning with two connected minus signs, a comment, and a terminating new line:

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

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

Right Truncation of Character Strings

The **string_rtruncation set** option 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

The **ansi_permissions set** option determines what permissions are required for **delete** and **update** statements. When this option is **on**, Adaptive Server uses the SQL92 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 allow compliance with the SQL92 standard as follows:

- **arithabort arith_overflow** specifies behavior following a divide-by-zero error or a loss of precision. The default setting, **arithabort arith_overflow on**, rolls back the entire transaction in which the error occurs. If the error occurs in a batch that does not contain a transaction, **arithabort arith_overflow on** does not roll back earlier commands in the batch, but Adaptive Server does not execute statements in the batch that follow the error-generating statement.

  If you set **arithabort arith_overflow off**, Adaptive Server aborts the statement that causes the error but continues to process other statements in the transaction or batch.

- **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**, Adaptive Server truncates the query results and continues processing. For compliance to the SQL92 standard, enter **set arithabort numeric_truncation on**.

- **arithignore arith_overflow** determines whether Adaptive 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, enter **set arithignore off**.

Synonymous Keywords

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

Table 1-8: ANSI-compatible keyword synonyms

Current Syntax	Additional Syntax
commit tran, **commit transaction** **rollback tran**, **rollback transaction**	**commit work** **rollback 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

The **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 Adaptive Server evaluates null values in other kinds of SQL statements such as **create table**.

Adaptive Server Login Accounts

Each Adaptive Server user has a login account identified by a unique login name and a password. The account is established by a System Security Officer. Login accounts have the following characteristics:

- A login name, unique on that server.
- A password
- A default database (optional). If one is defined, you start each Adaptive Server session in that database without having to issue the **use** command. If none is defined, you start each session in the *master* database.
- A default language (optional). This specifies the language in which prompts and messages will be displayed to you. If one is not defined, Adaptive Server's default language is used.
- A full name (optional). This is your full name, which can be useful for documentation and identification purposes. If it is omitted, no full name is recorded for your account.

Group Membership

In Adaptive Server, groups provide a convenient way to grant and revoke permissions to more than one user at a time within a database. For example, if everyone who works in the "Sales" department needs access to certain tables, all of those users can be put into a group called "sales". The Database Owner can grant specific access permissions to that group rather than having to grant permission to each user individually.

A group is created within a database, not on the server as a whole. The Database Owner is responsible for creating groups and assigning users to them. You are always a member of the "public" group, which includes all users on Adaptive Server. You can also belong to one other group. You can use the **sp_helpuser** system procedure to find out what group you are a member of. The syntax for **sp_helpuser** is:

```
sp_helpuser user_name
```

Role Membership

In Adaptive Server, a System Security Officer can define and create roles as a convenient way to grant and revoke permissions to several

users at a time, server-wide. For example, clerical staff may need to be able to insert and select from tables in several databases, but they may not need to update them. A System Security Officer could define a role called “clerical_user_role” and grant the role to everyone in the clerical staff. Database object owners could then grant “clerical_user_role” the required privileges.

Roles can be defined in a role hierarchy, where a role such as “office_manager_role” contains the “clerical_user_role”. Users who are granted roles in a hierarchy automatically have all the permissions of the roles that are lower in the hierarchy. For example, the Office Manager can perform all the actions permitted for the clerical staff. Hierarchies can include either system or user-defined roles.

You can use the following system procedures to find out more about roles assigned to you:

- **sp_displayroles**

  To find out all roles assigned to you, whether or not they are active, use **sp_displayroles**.

- **sp_activeroles**

  To find out which of your assigned roles are active, use **sp_activeroles**. If you specify the **expand_down** parameter, Adaptive Server displays any roles contained within your currently active roles.

The syntax is:

```
sp_displayroles user_name
sp_activeroles expand_down
```

For more information about roles, see the *System Administration Guide.*

Getting Information About Your Adaptive Server Account

You can get information about your own Adaptive Server login account with the **sp_displaylogin** system procedure. The syntax is:

```
sp_displaylogin
```

Adaptive Server returns the following information:

- Your server user ID
- Your login name
- Your full name

- Any roles granted to you (regardless of whether they are currently active)
- Whether your account is locked
- The date when you last changed your password

Changing Your Password

It is a good idea to change your password periodically. The System Security Officer can configure Adaptive Server to require you to change your password at preset, regular intervals. If this is the case on your server, Adaptive Server will notify you when it is time to change your password.

➤ ***Note***

If you use remote servers, you must change your password on all remote servers that you access **before** you change it on your local server. For more information, see "Changing Your Password on a Remote Server" on page 1-33.

You can change your password at any time with the system procedure **sp_password**. The syntax is:

```
sp_password old_passwd, new_passwd
```

where *old_passwd* is the old password, and *new_passwd* is your new password.

Use the following rules when creating a new password:

- It must be at least 6 bytes long.
- It can be any printable letters, numbers, or symbols.
- The maximum size for a password is 30 bytes. If your password exceeds 30 bytes, Adaptive Server uses the first 30 characters, and then informs you that it is ignoring the rest.

When you specify a password, enclose it in quotation marks if:

- It includes characters other than A–Z, a–z, 0–9, _, #, valid single-byte or multibyte alphabetic characters, or accented alphabetic characters
- It begins with a number 0–9

The following example shows how to change the password "terrible2" to "3blindmice":

```
sp_password terrible2, "3blindmice"
```

A return status of 0 means that the password was changed. For more information about **sp_password**, see the *Adaptive Server Reference Manual*.

Protecting Your Password

Your password is the first line of defense against Adaptive Server access by unauthorized people. When you create your password, choose one that cannot be guessed, following these guidelines:

- Do not use personal information, such as your birthday, street address, or any other number that has anything to do with your personal life.
- Do not use names of pets or loved ones.
- Do not use words that appear in the dictionary or words spelled backwards.

The most difficult passwords to guess are those that combine uppercase and lowercase letters or both numbers and letters. Protecting your password is your responsibility. Never give anyone your password, and never write it down where anyone can see it.

Understanding Remote Logins

You can execute stored procedures on a remote Adaptive Server if you have been granted access to the remote server and an appropriate database on that server. Remote execution of a stored procedure is a **remote procedure call** (RPC).

To get access to a remote server, the following events must occur:

- The remote server must be known to your local server. This occurs when the System Security Officer executes **sp_addserver**.
- You must acquire a login on the remote server. This is accomplished when the System Administrator executes **sp_addremotelogin**.
- You must be added as a user to the appropriate database on the remote server. This is accomplished when the Database Owner executes **sp_adduser**.

These procedures are discussed in the *System Administration Guide*.

When you can access the remote server, you can execute a **remote procedure call (RPC)** by qualifying the stored procedure name with the name of the remote server. For example, to execute **sp_help** on the GATEWAY server, enter:

```
GATEWAY...sp_help
```

Or, to fully qualify the name of the procedure, include the name of the database containing the procedure and the name of its owner:

```
GATEWAY.sybsystemprocs.dbo.sp_help
```

In addition, if a System Security Officer has set up the local and remote servers to use network-based security services, one or more of the following functions may be in effect when you execute RPCs:

- Mutual authentication – the local server authenticates the remote server by retrieving the credential of the remote server and verifying it with the security mechanism. With this service, the credentials of both servers are authenticated and verified.
- Message confidentiality via encryption – messages are encrypted when sent to the remote server, and results from the remote server are encrypted.
- Message integrity – messages between the servers are checked for tampering.

If you are using the unified login feature of network-based security, a System Security Officer can use **sp_remoteoption** on the remote server to establish you as a trusted user who does not need to supply a password to the remote server. An alternative, if you are using Open Client™ Client-Library™ / C, is to use **ct_remote_pwd** to specify a password for the remote server.

For more information about network-based security services, see the *System Administration Guide*.

➤ ***Note***

Remote logins are not included in the **evaluated configuration**.

Changing Your Password on a Remote Server

You must change your password on all remote servers that you access **before** you change it on your local server because of the way password checking is done. If you change your password on the local server first, when you issue the remote procedure call (RPC) to

execute **sp_password** on the remote server, your local and remote passwords will not match and the command will fail.

The syntax for changing your password on the remote server is:

remote_server*...sp_password *old_passwd*, *new_passwd

For example:

```
GATEWAY...sp_password terrible2, "3blindmice"
```

For information on changing your password on the local server, see "Changing Your Password" on page 1-31.

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

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

To use Transact-SQL, you must set up an account, or login, on Adaptive Server. To use **isql**, type a command similar to the following at your operating system prompt:

```
isql -Uhoratio -Pmonkeybrains -Shaze -w300
```

where "horatio" is the user, "monkeybrains" is the password, and "haze" is the name of the Adaptive Server you are connecting to. The **-w** parameter displays **isql** output at a width of 300 characters. Login names and passwords are case-sensitive.

When you specify **isql** without a login name, Adaptive Server assumes that your login name is the same as your operating system name. For details about specifying your server login name and other parameters for **isql**, see the *Utility Programs* manual for your platform.

➤ ***Note***

Do not use the **-P** option to **isql** to specify your password, because another user might see your password.

After you start **isql**, this is what you will see:

```
1>
```

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

To connect to a non-Sybase database using Component Integration Services, use the **connect to** command. For more information, see the *Component Integration Services User's Guide*. See also **connect to...disconnect** in the *Adaptive Server Reference Manual*.

Default Databases

When your Adaptive 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 Adaptive Server login, that is listed in *master..syslogins*, can be a guest. To change your default database, use the system procedure **sp_modifylogin**, described in the *Adaptive Server Reference Manual*.

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

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

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 Adaptive Server know that you have finished typing, and you are ready for your command to be executed.

With a couple of exceptions, the examples of the 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**.

Using Network-Based Security Services with *isql*

You can specify the **-V** option of **isql** to use network-based security services such as "unified login". With "unified login", you can be authenticated with a security mechanism offered by a third-party provider and then log into Adaptive Server without specifying a login name or a password. Other security services you can use, if they are supported by the third-party security mechanism, are:

- Data confidentiality
- Data integrity
- Mutual authentication
- Data origin checking
- Data replay detection
- Out-of-sequence detection

See the *System Administration Guide* for more information about the options you can specify for using network-based security.

Logging Out of *isql*

You can log out of **isql** at any time by typing:

```
quit
```

or

```
exit
```

Either of these commands returns you to the operating system prompt.

For more details on the **isql** utility, see the the *Utility Programs* manual for your platform.

Using the *pubs2* and *pubs3* Sample Database

The *pubs2* sample database is used for most of the examples in this manual, except for examples, where noted, that use the *pubs3* database. 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 have been 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 databases.

What Is in the Sample Databases?

The sample database, *pubs2*, contains these tables: *publishers*, *authors*, *titles*, *titleauthor*, *roysched*, *sales*, *salesdetail*, *stores*, *discounts*, *au_pix*, and *blurbs*. The *pubs3* sample database, which includes the *store_employees* table, has most of the same tables as *pubs2*, except for *au_pix*. *pubs3* is an updated version of *pubs2*; it is useful for referential integrity examples, and its tables are slightly different from the tables defined in *pubs2*.

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, 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. It contains each book's title ID, author ID, 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.
- *store_employees* lists employees for the stores described in the *stores* table.
- *discounts* lists three types of discounts for bookstores.
- *au_pix* contains pictures of the authors in binary form using the *image* datatype. *au_pix* is in *pubs2* only.
- *blurbs* contains long book descriptions in the *text* datatype.

The *pubs2* database is illustrated in Appendix A, "The pubs2 Database." *pubs3* is illustrated in Appendix B, "The pubs3 Database."

Part 1: Basic Concepts

2 Queries: Selecting Data from a Table

The **select** command queries 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:

This chapter focuses on basic single-table **select** statements. Advanced uses of **select** are described later in this manual.

What Are Queries?

A query requests data from the database and receives 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 columnar format, like this:

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

(5 rows affected)
```

If you are running parallel queries, the results can appear in a different order than with serial queries. For more information on running queries in parallel, see the *Performance and Tuning Guide*.

select Syntax

The **select** syntax is both simpler and more complex than the example shown in the previous section. 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. It is more complex because 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 | table_name }
           [parallel [degree_of_parallelism]]
           [prefetch size ][lru|mru])]}
       [holdlock | noholdlock] [shared]
    [,[[database.]owner.]{view_name|table_name
       [(index {index_name | table_name }
           [parallel [degree_of_parallelism]]
           [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]
```

Use the clauses in a **select** statement in the order shown above. For example, 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 explained in the section "Identifiers" on page 1-7, qualify the names of database objects if there is ambiguity about which object is being referred to. For example, if several columns are called *name*, you may have to qualify *name* with the database name, owner name, or table name, for example:

```
select au_lname from pubs2.dbo.authors
```

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 remaining sections in 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 following clauses are discussed in other chapters, as follows:

- **group by**, **having**, **order by**, and **compute** are described in Chapter 3, "Summarizing, Grouping, and Sorting Query Results."
- **into** is described in Chapter 7, "Creating Databases and Tables."
- **at isolation** is described in Chapter 18, "Transactions: Maintaining Data Consistency and Recovery."

The **holdlock**, **noholdlock**, and **shared** keywords (which deal with locking in Adaptive Server) and the **index** clause are described in the *Performance and Tuning Guide*. For information about the **for read only** and **for update** clauses, see the **declare cursor** command in the *Adaptive Server Reference Manual*.

➤ ***Note***

The **for browse** clause is used only in DB-Library applications. See the *Open Client DB-Library/C Reference Manual* for details. See also "Using Browse Mode in Place of Cursors" on page 17-27.

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, **case** expression, 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: s*elect* *

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 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 **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 Specific Columns

To select only specific columns in a table, use this syntax:

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

Separate each column name from the column name that follows it with a comma, for example:

```
select au_lname, au_fname
from authors
```

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 to 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 one 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 the following **select** queries produce the same result:

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

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

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

(3 rows affected)
```

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.

Bitwise Operators

The bitwise operators are a Transact-SQL extension for use with integer type data. These operators convert each integer operand into its binary representation, then evaluate the operands column by column. A value of 1 corresponds to true; a value of 0 corresponds to false.

Table 2-1 shows the bitwise operators.

Table 2-1: Bitwise operators

Operator	Meaning
&	Bitwise and (two operands)
|	Bitwise or (two operands)
^	Bitwise exclusive or (two operands)
~	Bitwise not (one operand)

For more information on bitwise operators, see the *Adaptive Server Reference Manual*.

Arithmetic Operators

The following table shows the available arithmetic operators.

Table 2-2: Arithmetic operators

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

You can use the arithmetic operators—addition, subtraction, division, and multiplication— 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 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 is 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 multitable 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, and then subtracts from that the author's advance divided in half.

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

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*.

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

text and *image* values can be quite large. 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 Adaptive 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, after the *@val* local variable has been declared, **readtext** displays characters 3–8 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_pointer offset size
   [holdlock | noholdlock] [readpast]
   [using {bytes | chars | characters}]
   [at isolation {
       [ read uncommitted | 0 ] |
       [ read committed | 1 ] |
       [ repeatable read | 2 ]|
       [ serializable | 3 ] } ]
```

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 18, "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** 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.

Adaptive 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, Adaptive 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 useful only when Adaptive Server is using a multibyte character set. This option ensures that **readtext** does not return partial characters.

When using bytes as the offset, Adaptive 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 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 do not specify **distinct**, you get all rows, including duplicates. Optionally, you can specify **all** before the select list to get all rows. For compatibility with other implementations of SQL, Adaptive Server syntax allows the use of **all** to explicitly ask for 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 will 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)
```

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.

When used with the **order by** clause, the **distinct** keyword can return multiple values. See "order by and group by Used with and select distinct" on page 3-28 for more information.

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, a table being created with the **into** keyword, 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 characters that follow can be digits, letters, or @, #, $, _, ¥, or £. Temporary table names must begin either with "#" (pound 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 Adaptive 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 necessary only when there might be some confusion about the name. You can give table names correlation names to save typing. Assign the **correlation name** 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 search conditions 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 more information possible search conditions, see the "**where** Clause" section in the *Adaptive Server Reference Manual*.

Comparison Operators

Transact-SQL uses the following comparison operators:

Table 2-3: Comparison operators

Operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to
!=	Not equal to (Transact-SQL extension)
!>	Not greater than (Transact-SQL extension)
!<	Not less than (Transact-SQL extension)

The operators are used in the syntax:

```
where expression comparison_operator expression
```

where an *expression* is a constant, column name, function, subquery, **case** expression, or any combination of these 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 Adaptive 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 following two queries will find all business and psychology books that do not have an advance of more than $5500. 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 $4095 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 $4095 are included in the results. Books with sales of $12,000 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 $4095 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. The expression can be a constant or a column name, and the values list can be a set of constants or, more commonly, a subquery.

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. Put single or double quotes around *char*, *varchar*, and *datatime* values. For example:

```
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 called **subqueries**. For a full discussion of subqueries, see Chapter 5, "Subqueries: Using Queries Within Other Queries." However, the following example gives an 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 tables 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
Gringlesby        Burt
Yokomoto          Akiko
MacFeather        Stearns
Ringer            Anne

(6 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
Smith             Gabriella

(18 rows affected)
```

Matching Character Strings: *like*

The **like** keyword indicates that the following character string (enclosed by single or double quotes) is a matching pattern. **like** is used with *char, varchar, nchar, nvarchar, binary, varbinary, text*, and *datetime* data.

The syntax for **like** is:

```
{where | having} [not]
   column_name [not] like  "match_string"
```

The column data is compared to a *match_string* that can include these special symbols:

Table 2-4: Special symbols for matching character strings

Symbols	Meaning
%	Matches any string of 0 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] 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.”

You can match the column data to constants, variables, or other columns that contain the **wildcard** characters shown in Table 2-4. 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 containing the letters ''en'' (Bennet, Green, McBadden).
- **like** "_heryl" searches for every 6-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'' that do not have ''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%"
```

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

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

Adaptive Server interprets wildcard characters used without **like** 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%"
```

When you use **like** with *datetime* values, Adaptive Server converts the dates to the standard *datetime* format, and then to *varchar*. Since the standard storage format does not include seconds or milliseconds, you cannot search for seconds or milliseconds with **like** and a pattern.

It is a good idea to use **like** when you search for *datetime* values, since *datetime* entries may contain a variety of date parts. For example, if you insert the value "9:20" and the current date into a column named *arrival_time*, the clause:

```
where arrival_time = "9:20"
```

would not find the value, because Adaptive Server converts the entry into "Jan 1 1900 9:20AM." However, the clause below would find the 9:20 value:

```
where arrival_time like "%9:20%"
```

Using *not like*

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

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

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

not like and ^ May Give Different Results

You cannot always duplicate **not like** patterns with **like** and the negative wildcard character [^]. This is because **not like** finds the items that do not match the entire **like** pattern, but **like** with negative wildcard characters is evaluated one character at a time.

For example, this query finds the system tables in a database whose names begin with "sys":

```
select name
from sysobjects
where name like "sys%"
```

To see all the objects that are **not** system tables, use:

```
not like "sys%"
```

If you have a total of 32 objects and **like** finds 13 names that match the pattern, **not like** will find the 19 objects that do not match the pattern.

A pattern such as the following may not produce the same results:

```
like [^s][^y][^s]%
```

Instead of 19, you might get only 14, with all the names that begin with "s" **or** have "y" as the second letter **or** have "s" as the third letter eliminated from the results, as well as the system table names. This is because match strings with negative wildcard characters are evaluated in steps, one character at a time. If the match fails at any point in the evaluation, it is eliminated.

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.

Square Brackets (Transact-SQL Extension)

Use square brackets as characters for the percent sign, the underscore, and the left bracket. 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 2-5: Using square brackets to search for wildcard characters

like Clause	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** clause:

Table 2-6: Using the escape clause

like Clause	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 Adaptive 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** clause and has no effect on other **like** clauses contained in the same statement.
- The only characters that are valid following an escape character are the wildcard characters (_ , % , [,] , and [^]), 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 last example in Table 2-6 on page 2-29. Otherwise, Adaptive Server raises a SQLSTATE error condition and returns an error message.

  For example, the following escape clauses cause a SQLSTATE 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 left bracket.

Do not use existing wildcard characters as escape characters, for these reasons:

- If you specify “_” or “%” as an escape character, it loses its special meaning within that **like** clause and acts only as an escape character.
- If you specify “[“ or “]” as an escape character, the Transact-SQL meaning of the bracket is disabled within that **like** clause.
- If you specify “-” or “^” as an escape character, it loses the special meaning that it normally has within square brackets and acts only as an escape character.

Trailing Blanks and %

Adaptive Server truncates trailing blanks following “%” in a **like** clause to a single trailing blank. **like** ‘‘% ’’ (percent sign 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

You can use wildcard characters in columns and 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 Adaptive Server will treat the data as 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 want 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

A NULL in a column 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 where NULL values are permitted, as specified in the **create table** statement, in two ways:

- If no data is entered, Adaptive 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 in 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)
```

Testing a Column for Null Values

Use **is null** in **where**, **if**, and **while** clauses to compare column values to NULL and to select them or perform a particular action based on the results of the comparison. Only columns that return a value of TRUE are selected or result in the specified action; those that return FALSE or UNKNOWN do not.

The following example selects only rows for which *advance* is $5000 or null:

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

Adaptive Server treats null values in different ways, depending on the **operators** that you use and the type of values you are comparing. In general, the result of comparing null values is UNKNOWN, since it is not possible to determine whether NULL is equal (or not equal) to a given value or to another NULL. The following cases return TRUE when *expression* is any column, variable or literal, or combination of these, which evaluates as NULL:

- *expression* **is null**
- expression = **null**
- *expression* = *@x* where *@x* is a variable or parameter containing NULL. This exception facilitates writing stored procedures with null default parameters.
- *expression* != *n* where *n* is a literal not containing NULL and *expression* evaluates to NULL.

The negative versions of these expressions return TRUE when the expression does not evaluate to NULL:

- *expression* **is not null**
- *expression* != **null**
- *expression* != *@x*

Note that the far right side of these exceptions is a literal null, or a variable or parameter containing NULL. If the far right side of the comparison is an expression (such as *@nullvar* + 1), the entire expression evaluates to NULL.

Following these rules, null column values do not join with other null column values. Comparing null column values to other null column values in a **where** clause always returns UNKNOWN for null values, regardless of the comparison operator, and the rows are not included in the results. For example, this query returns no result rows where column1 contains NULL in both tables (although it may return other rows):

```
select column1
from table1, table2
where table1.column1 = table2.column1
```

These operators 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 do not return results when used with

a NULL. Regardless of the **set ansinull** option value, the following operators never return values when used with a NULL: **<**, **<=**, **!<**, **>**, **>=**, **!>**.

Adaptive Server can determine that a column value is NULL; thus:

```
column1 =  NULL
```

can be considered to be true. However, the following comparisons can never be determined, since NULL means "having an unknown value":

```
where column1 > null
```

There is no reason to assume that two unknown values are the same.

This logic also applies when you use two column names in a **where** clause, that is, when you join two tables. A clause like "where column1 = column2" does not return rows where the columns contain null values.

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

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

For example:

```
where advance < $5000 or advance is null
```

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 have been entered. However, since the author has no contract 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 do not match anything in a comparison, a query for all the title identification numbers and advances for books with moderate advances (of less than $5000) 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 of less than $5000 **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.

Difference Between FALSE and UNKNOWN

Although neither FALSE nor UNKNOWN returns values, there is an important logical difference between FALSE and UNKNOWN, because the opposite of false ("not false") is true. For example, "1 = 2" evaluates to false and its opposite, "1 != 2", evaluates to true. But "not unknown" is still unknown. If null values are included in a comparison, you cannot negate the expression to get the opposite set of rows or the opposite truth value.

Substituting a Value for NULLs

Use the **isnull** built-in function to substitute a particular value for nulls. The substitution is made only for display purposes; actual column values are not affected. The syntax is:

isnull(*expression*, *value*)

For example, use the following statement to select all the rows from *test*, and display all the null values in column *t1* with the value unknown.

```
select isnull(t1, "unknown")
from test
```

Expressions That Evaluate to NULL

An expression with an arithmetic or bitwise operator evaluates to NULL if any of the operands are null. For example:

```
1 + column1
```

evaluates to NULL if *column1* is NULL.

Concatenating Strings and NULL

If you concatenate a string and NULL, the expression evaluates to the string. For example:

```
select "abc" + NULL + "def"

-----
abcdef
```

System-Generated NULLs

In Transact-SQL, system-generated NULLs, such as those that result from a system function like **convert**, behave differently than user-assigned NULLs. For example, in the following statement, a not equals comparison of the user-provided NULL with 1 returns TRUE:

```
if (1 != NULL) print "yes" else print "no"

yes
```

while the same comparison with a NULL generated by the convert function returns UNKNOWN:

```
if (1 != convert(integer, NULL))
print "yes" else print "no"

no
```

For more consistent behavior, set the **ansinull** option to **on**. Then both system-generated and user-provided NULLs cause the comparison to return UNKNOWN.

Connecting Conditions with Logical Operators

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

```
{where | having} [not]
   column_name join_operator column_name
```

where *join_operator* is a comparison operator and *column_name* is the name of a column used in the comparison. Qualify the name of the column if there is any ambiguity.

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"
```

You can specify up to 252 **and** and **or** conditions.

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"
```

When more than one logical operator is used in a statement, **and** operators are normally evaluated before **or** operators. You can change the order of execution with parentheses. For example:

```
select * from authors
where (city = "Oakland" or city = "Berkeley") and
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 "Bitwise Operators" on page 1-13 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 of more than $5500. 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 of more than $5500:

```
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

Aggregate functions display summaries of the values in specified columns. This chapter describes how to use the following aggregate functions: **sum**, **avg**, **count**, **count(*)**, **max**, and **min**. You can also use the **group by** clause, **having** clause, and **order by** clause to group and sort the results of queries using aggregate functions. The **compute** clause, a Transact-SQL extension, displays subtotals of grouped summaries. The **union** operator combines the results of queries.

This chapter discusses:

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

Summarizing Query Results Using Aggregate Functions

You can apply aggregate functions 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, Adaptive Server generates a single value.

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

```
select sum(total_sales)
from titles

-------------
       220591

(1 row affected)
```

To use the aggregate functions, 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(*)**. Use the optional keyword **distinct** 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. A **case** expression can be used in an expression. An expression can also be a subquery.

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

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

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

(1 row affected)
```

The syntax of the aggregate functions and the results they produce are shown in Table 3-1:

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 3-20.

You **cannot** use aggregate functions 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 in this section, 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** statement can also include a column in its select list (a Transact-SQL extension), that repeats 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 3-20.

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"

------------ ---------
      6281.25    68,396

(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, called a **scalar aggregate**. 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.

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

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

You cannot use **min** and **max** with *bit* datatypes.

You cannot use aggregate functions other than **count(*)** with *text* and *image* datatypes.

With these exceptions, you can use the aggregate functions 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. Use **count(*)** 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, you can combine **count(*)** 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 you use **distinct**, Adaptive Server eliminates duplicate values before calculating the sum, average, or count.

If you use **distinct**, you cannot include an arithmetic expression in the argument. The argument must use a column name only. **distinct** 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, omit **distinct**:

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

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

(1 row affected)
```

Null Values and the Aggregate Functions

Adaptive Server ignores any null values in the column on which the aggregate function is operating for the purposes of the function (except **count**(*), which includes them). If you have set **ansinull** to **on**, Adaptive Server returns an error message whenever a null value is ignored. For more information, see the **set** command in the *Adaptive Server Reference Manual.*

If all the values in a column are null, **count**(*column_name*) returns 0. 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 0. 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 divides 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. The maximum number of columns or expressions you can use in a **group by** clause is 16.

➤ ***Note***

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

A **group by** clause almost always appears in statements that 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    30,788
mod_cook         7,500.00    24,278
popular_comp     7,500.00    12,875
psychology       4,255.00     9,939
trad_cook        6,333.33    19,566

(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

---------   -------
5962.50     220591

(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 | table_name }
           [parallel [degree_of_parallelism]]
           [prefetch size ][lru|mru])]}
       [holdlock | noholdlock] [shared]
    [,[[database.]owner.]{view_name|table_name
       [(index {index_name | table_name }
           [parallel [degree_of_parallelism]]
           [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 above.

group by and SQL Standards

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 be in the **group by** expression or they must be arguments of aggregate functions.
- A **group by** expression can only contain column names 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. To refrain from using the extensions, set the **fipsflagger** option as follows:

```
set fipsflagger on
```

Nesting Groups with *group by*

This option displays a warning message whenever Transact-SQL extensions are used. For more information about the **fipsflagger** option, see the **set** command in the *Adaptive 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  18,722
0736    psychology    11.48   9,564
0877    UNDECIDED      NULL    NULL
0877    mod_cook      11.49  24,278
0877    psychology    21.59     375
0877    trad_cook     15.96  19,566
1389    business      17.31  12,066
1389    popular_comp  21.48  12,875

(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 the following 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**:

- The columns in the select list are **not** limited to the grouping columns and columns used with the vector aggregates.
- 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 bulleted item described above allows you to specify additional "extended" columns in the select list of the query.

For example, many versions of SQL do not allow the inclusion of the extended *title_id* column in the select list, but it is 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 Adaptive 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**, do not use the extensions frivolously. Use the **fipsflagger** option to identify queries that use these extensions. See "group by and SQL Standards" on page 3-9 for details.

For more information about Transact-SQL extensions to **group by** and how they work, see the *Adaptive 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

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

(16 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
```

The **group by** clause should be "**group by type**", not "**group by Category**".

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 form 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
           2000.00               39.98
           2275.00               21.90
           4000.00               19.94
           5000.00               34.62
           6000.00               14.00
           7000.00               43.66
           8000.00               34.99
          10125.00                5.98
          15000.00                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 should 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 of more than $5000 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 Adaptive 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. Adaptive 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 Adaptive Server handles the **where** clause and **group by**, see the *Adaptive 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 do not 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 that commanded advances of more than $1000 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 does not include any rows that meet the qualification specified in the **where** clause. Adaptive 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 to 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)
```

Adaptive 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

Use the **having** clause to display or reject rows defined by the **group by** clause. It 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 except that **where** search conditions cannot include aggregates, while **having** search conditions often do. The example below is legal:

```
having avg(price) > $20
```

But this example is not:

```
where avg(price) > $20
```

having clauses can reference any of the items that appear in the select list.

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 displays only those types that 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 who have paid more than $15,000 in total advances, whose books average less than $18 in price, and whose identification numbers (*pub_id*) are greater than 0800, 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          1,845
5023     PS3333          3,437
5023     PS7777          2,206
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 more than $10,000 in 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 three business, one mod_cook, two popular_comp, one psychology, and two trad_cook titles exceed $4000 in total sales.

As mentioned earlier, the way Adaptive 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 **group by** 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 advances of more than $4000 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, that is, rows of the single group.

```
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 *Adaptive 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 31. Each sort can be 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 you name more than one column in the **order by** clause, Adaptive Server nests the sorts. 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). Adaptive Server sorts null values 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)
```

You can use the position number of a column in a select list 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.

Adaptive Server does not allow subqueries, aggregates, variables and constant expressions in the **order by** list.

With **order by**, null values come before all others.

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

order by and *group by*

You can use an **order by** clause 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)
```

order by and *group by* Used with and *select distinct*

A **select distinct** query with **order by** or **group by** can return duplicate values if the **order by** or **group by** column is not in the **select** list. For example:

```
select distinct pub_id
from titles
order by type
```

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

(8 rows affected)
```

If a **select** has an **order by** or **group by** clause that includes columns not in the **select** list, Adaptive Server adds those columns as hidden columns in the columns being processed. The columns listed in the **order by** or **group by** clause are included in the test for distinct rows. To comply with ANSI standards, include the **order by** or **group by** column in the **select** list. For example:

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

```
pub_id type
------ ------------
0877   UNDECIDED
0736   business
1389   business
0877   mod_cook
1389   popular_comp
0736   psychology
0877   psychology
0877   trad_cook

(8 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 show subtotals of grouped summaries. 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

Compute Result:
------------------------ ------------------------
                   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

Compute Result:
------------------------ ------------------------
                  54.92                 25,125.00
type         price                    advance
------------ ------------------------ ---------
mod_cook      2.99                    15,000.00
mod_cook     19.99                         0.00
```

```
Compute Result:
----------------------- -------------------------
                  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

Compute Result:
----------------------- -------------------------
                  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

Compute Result:
----------------------- -------------------------
                  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

Compute Result:
----------------------- -------------------------
                  47.89                19,000.00

(24 rows affected)
```

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

Row Aggregates and *compute*

The row aggregates used with **compute** are listed in Table 3-2:

Table 3-2: How aggregates are used with a compute statement

Row Aggregates	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 the **by** keyword.

Rules for *compute* Clauses

- Adaptive Server does not allow the **distinct** keyword 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, suppose 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**. The **compute** keyword without **by** is discussed under "Grand Values: compute Without by" on page 3-36.

Specifying More Than One Column After *compute*

Listing more than one column after the **by** keyword breaks a group into subgroups and applies the specified row aggregate to 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

Compute Result:
---------------
          45.93
type        pub_id  price
----------- ------- -------------
psychology    0877          21.59

Compute Result:
---------------
          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 query is 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

Compute Result:
---------------
          45.93

type        pub_id  price
 ---------- ------- --------------
 psychology    0877         21.59

Compute Result:
---------------
          21.59

Compute Result:
---------------
           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

Compute Result:
--------------- ---------------
          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

Compute Result:
--------------- ---------------
          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

Compute Result:
--------------- ----
          22.98 0877

type        pub_id  price
----------- ------- --------------
trad_cook   0877            11.95
trad_cook   0877            14.99
trad_cook   0877            20.95

Compute Result:
--------------- ----
          47.89 0877

(7 rows affected)
```

Grand Values: *compute* Without *by*

You can use the **compute** keyword 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 that cost more than $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

Compute Result:
--------------- ---------
          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

Compute Result:
--------------- ---------
          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

Compute Result:
--------------- ---------
          47.89 19,000.00

Compute Result:
--------------- ---------
          70.87 34,000.00

(8 rows affected)
```

Combining Queries: The *union* Operator

The **union** operator combines the results of two or more queries into a single result set. The Transact-SQL extension to **union** lets you perform the following tasks:

- You can use **union** in the **select** clause of an **insert** statement.
- You can specify new column headings in the **order by** clause of a **select** statement when **union** is present in the **select** statement.

The syntax of the **union** operator 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

a     b
----  ---------
abc           1
def           2
ghi           3
jkl           4
mno           5

(5 rows affected)
```

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, Adaptive Server evaluates a statement containing **union** operators from left to right. Parentheses may be used to specify the order of evaluation.

For example, the following two expressions are not equivalent:

```
x union all (y union z)

(x union all y) union z
```

In the first expression, 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 expression, duplicates are included in the union between *x* and *y*, but are then eliminated in the subsequent union with *z*; **all** does not affect 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, city, state from stores
  union
  select stor_id, city 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 *Adaptive Server*

Reference Manual for more information about comparing datatypes in a **union** statement.

- You must place corresponding columns in the individual queries of a **union** statement 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

(3 rows affected)
```

The following query results in an error message, because the datatypes of corresponding columns are not compatible:

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

When you combine different (but compatible) datatypes such as *float* and *int* in a **union** statement, Adaptive Server converts them to the datatype with the most precision.

- Adaptive Server takes the column names in the table resulting from a **union** from the **first** individual query in the **union** statement. Therefore, if you want to define a new column heading for the result set, 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, refer to it 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.

- You can use **order by** and **compute** clauses only at the end of the **union** statement to define the order of the final results or to compute summary values. You cannot use them within the individual queries that make up the **union** statement.

- You can use **group by** and **having** clauses within individual queries only; you cannot use them to affect the final result set.

- You can also use the **union** operator within an **insert** statement. For example:

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

- You cannot use the **union** operator within a **create view** statement.

- You cannot use the **union** operator on *text* and *image* columns.

- You cannot use the **for browse** clause in statements involving the **union** operator.

4 Joins: Retrieving Data from Several Tables

A **join** operation compares two or more tables (or views) by specifying a column from each, comparing the values in those columns row by row, and linking the rows that have matching values. It then displays the results in a new table. The tables specified in the join can be in the same database or in different databases.

This chapter discusses:

You can state many joins as subqueries, which also involve two or more tables. See Chapter 5, “Subqueries: Using Queries Within Other Queries.”

When Component Integration Services is enabled, you can perform joins across remote servers. For more information, see the *Component Integration Services User’s Guide*.

How Joins Work

When you join two or more tables, the columns being compared must have similar values—that is, values using the same or similar datatypes.

There are several types of joins, such as equijoins, natural joins, and outer joins. The most common join, the equijoin, is based on equality.

The following join 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)
```

Because the query draws on information contained in two separate tables, *publishers* and *authors*, you need a join to retrieve the requested information. The statement:

```
where authors.city = publishers.city
```

joins the *publishers* and *authors* tables using the *city* column as the link.

Join Syntax

You can embed a join in a **select**, **update**, **insert**, **delete**, or subquery. Other join restrictions and clauses may follow the join conditions. Joins use the following syntax:

```
start of select, update, insert, delete, or subquery
   from {table_list | view_list}
   where [not]
       [table_name. | view_name.]column_name
                     join_operator
       [table_name. | view_name.]column_name
   [{and | or} [not]
       [table_name.|view_name.]column_name
                    join_operator
       [table_name.|view_name.]column_name]...
End of select, update, insert, delete, or subquery
```

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.

How Joins Are Structured

A join statement, like a select 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' names from the *authors* table, and publishers' names from the *publishers* tables:

```
select au_fname, au_lname, pub_name
from authors, publishers
```

You do not have to qualify the columns *au_fname*, *au_lname*, and *pub_name* by a table name because there is no ambiguity about the table to which they belong. But the *city* column used for the join comparison does need to be qualified, because there are columns of that name in both the *authors* and *publishers* tables:

```
where authors.city = publishers.city
```

Though neither of the *city* columns is printed in the results, Adaptive Server needs the table name to perform the comparison.

To specify that all the columns of the tables involved in the query be included in the results, use an asterisk (*****) with **select**. For example, to include all the columns in *authors* and *publishers* 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 2 rows with 13 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

Use the **from** clause to specify which tables and views to join. This is the clause that indicates to Adaptive 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:

- Base tables or views 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
- Tables being created with **into**
- Work tables created as a result of the query

If the join is part of an update or delete statement, the query can only refer to 15 tables.

The following example joins columns from the *titles* and *publishers* tables, doubling the price of all books published in California:

```
update titles
  set price = price * 2
  from titles, publishers
  where titles.pub_id = publishers.pub_id
  and publishers.state = "CA"
```

See “Joining More Than Two Tables” on page 4-16 for information on joins involving more than two tables or views.

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, for example:

```
select au_lname, au_fname
from pubs2.blue.authors, pubs2.blue.publishers
where authors.city = publishers.city
```

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

The *where* Clause

Use the **where** clause to determine which rows are included in the results. **where** specifies the connection between the tables and views named in the **from** clause. Be sure to qualify column names if there is

ambiguity about the table or view to which they belong. For example:

```
where authors.city = publishers.city
```

This **where** clause 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.

➤ ***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 69 rows instead of 2. The section "How Joins Are Processed" on page 4-7 explains why that happens.

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. See "How Joins Are Processed" on page 4-7 for an example.

Join Operators

Joins that match columns on the basis of equality are called **equijoins**. A more precise definition of an **equijoin** is given under "Equijoins and Natural Joins" on page 4-8, along with examples of joins not based on equality.

Equijoins use the following comparison 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 Transact-SQL extensions, as shown in Table 4-2:

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.

Datatypes in Join Columns

The columns being joined must have the same or compatible datatypes. Use the **convert** function when comparing columns whose datatypes cannot be implicitly converted. Columns being joined need not have the same name, although they often do.

If the datatypes used in the join are compatible, Adaptive Server automatically converts them. For example, Adaptive Server 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 *Adaptive Server Reference Manual.*

Joins and *text* and *image* Columns

You cannot use joins for columns containing *text* or *image* values. You can, however, compare the lengths of text columns from two tables with a where clause. For example:

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

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. Adaptive 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, Adaptive 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, because it implies that every author in the database has a relationship with every publisher in the database—which is not true at all.

That is why you must include a **where** clause in the join, which specifies the columns to be matched and the basis on which to match them. It may also include other restrictions. Once Adaptive Server forms the Cartesian product, it eliminates the rows that do not satisfy the join by using the conditions in the **where** clause.

For example, the **where** clause in the previous example eliminates from the results all rows in which the author's city is not the same as the publisher's city:

```
where authors.city = publishers.city
```

Equijoins and Natural Joins

Joins based on equality (=) are called **equijoins**. Equijoins compare the values in the columns being joined for equality and then include all the columns in the tables being joined in the results.

This earlier query is an example of an equijoin:

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

In the results of that statement, the *city* column appears twice. By definition, the results of an equijoin contain two identical columns. Because 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.

Another example of a natural join is:

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

You can use more than one join operator to join more than two tables or to join more than two pairs of columns. These "join expressions" are usually connected with **and**, although **or** is also legal.

Following are two examples of joins connected by **and**. The first lists information about books (type of book, author, and title), ordered by book type. Books with more than one author have multiple listings, one for each author.

```
select type, au_lname, au_fname, title
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
order by type
```

The second finds the names of authors and publishers that are located in the same city and state:

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

Joins with Additional Conditions

The **where** clause of a join query can include selection criteria as well as the join condition. For example, to retrieve the names and publishers of all the books for which advances of more than $7500 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)
```

The columns being joined (*pub_id* from *titles* and *publishers*) do not need to appear in the select list and, therefore, do not show up in the results.

You can include as many selection criteria as you want 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 does not need to be equality. You can use any of the other comparison operators: 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

Joins that compare values within the same column of one table are called **self-joins**. To distinguish the two roles in which the table appears, use aliases, or correlation names.

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 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)
```

Be sure to list the aliases in the **from** clause in the same order as you refer to them in the select list, as in this example. Depending on the query, the results may be ambiguous if you list them in a different order.

To eliminate the rows in the results where the authors match themselves, and 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. In the following example, a not-equal join and a self-join 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)
```

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)
```

For each book in *titles*, the following example finds all other books of the same type that have a different price:

```
select t1.type, t1.title_id, t1.price,
t2.title_id, t2.price
from titles t1, titles t2
where t1.type = t2.type
and t1.price != t2.price
```

Be careful when interpreting the results of a not-equal join. For example, it would be easy to think you could use a not-equal join to find the authors who live in a city where no publisher is located:

```
select distinct au_lname, authors.city
from publishers, authors
where publishers.city != authors.city
```

However, this query finds the authors who live in a city where no publishers are located, which is all of them. The correct SQL statement is a subquery:

```
select distinct au_lname, authors.city
from publishers, authors
where authors.city not in
(select city from publishers
 where authors.city = publishers.city)
```

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 want to list the names of authors who live in a city where no publisher is located. For the sake of clarity, let us 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. You must use a subquery to get the results you want. A subquery can eliminate the ineligible 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 you want:

```
au_lname             city
-------------        ------------
Blotchet-Halls       Corvallis

(1 row affected)
```

Subqueries are covered in greater detail in Chapter 5, “Subqueries: Using Queries Within Other Queries.”

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 is 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

Joins that include all rows, regardless of whether there is a matching row, are called **outer joins**. Adaptive Server supports both left and right outer joins. For example, the following query joins the *titles* and the *titleauthor* tables on their *title_id* column:

```
select *
from titles, titleauthor
where titles.title_id *= titleauthor.title_id
```

Sybase supports both Transact SQL and ANSI outer joins. Transact SQL outer joins (shown in the previous example) use the *= command to indicate a left outer join and the =* command to indicate

a right outer join. T-SQL outer joins were created by Sybase as part of the Transact SQL language. See “Transact SQL Outer Joins” on page 4-37 for more information about T-SQL outer joins.

ANSI outer joins use the keywords **left join** and **right join** to indicate a left and right join, respectively. Sybase implemented the ANSI outer join syntax to fully comply with the ANSI standard. See “ANSI Inner And Outer Joins” on page 4-20 for more information about ANSI outer joins. This is the previous example re-written as an ANSI outer join:

```
select *
from titles left join titleauthor
on titles.title_id = titleauthor.title_id
```

Inner And Outer Tables

The terms **outer table** and **inner table** describe the placement of the tables in an outer join:

- In a **left join**, the **outer table** and **inner table** are the left and right tables respectively. The outer table and inner table are also referred to as the row-preserving and null-supplying tables, respectively.
- In a **right join**, the outer table and inner table are the right and left tables respectively.

For example, in the queries below, *T1* is the outer table and *T2* is the inner table:

```
T1 left join T2
T2 right join T1
```

Or, using Transact-SQL syntax:

```
T1 *= T2
T2 =* T1
```

Outer Join Restrictions

If a table is an inner member of an outer join, it cannot participate in **both** an outer join clause and a regular join clause. The following query fails because the *salesdetail* table is part of both the outer join and a regular join clause:

```
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 'FUSSY', 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 want to know the name of the store that sold more than 500 copies of a 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 not be 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.

Views Used with Outer Joins

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 not be what you expect. The query returns all rows from the inner table. Rows that do not meet the qualification show a NULL value in the appropriate columns of those rows.

The following rules determine what types of updates you can make to columns through join views:

- **delete** statements are not allowed on join views.
- **insert** statements are not allowed on join views created **with check option.**
- **update** statements are allowed on join 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.
- If you insert or update a row through a join view, all affected columns must belong to the same base table.

ANSI Inner And Outer Joins

ANSI provides the following syntax for joining tables:

```
left_table [inner | left [outer] | right [outer]] join right_table
on left_column_name = right_column_name
```

The result of the join between the left and the right tables is called a **joined table**. Joined tables are defined in the **from** clause. For example:

```
select titles.title_id, title, ord_num, qty
from titles left join salesdetail
on titles.title_id = salesdetail.title_id
```

```
title_id title              ord_num                qty
---------------------------- ---------------------- -----
BU1032  The Busy Executive   AX-532-FED-452-2Z7   200
BU1032  The Busy Executive   NF-123-ADS-642-9G3   1000
. . .
TC7777  Sushi, Anyone?       ZD-123-DFG-752-9G8   1500
TC7777  Sushi, Anyone?       XS-135-DER-432-8J2   1090
(118 rows affected)
```

ANSI join syntax allows you to write either:

- **Inner joins**, in which the joined table includes only the rows of the inner and outer tables that meet the conditions of the **on** clause. For information, see "ANSI Inner Joins" on page 4-22. The result set of a query that includes an inner join does not include any null supplied rows for the rows of the outer table that do not meet the conditions of the **on** clause.
- **Outer joins**, in which the joined table includes all the rows from the outer table whether or not they meet the conditions of the **on** clause. If a row does not meet the conditions of the **on** clause, values from the inner table are stored in the joined table as null values. The **where** clause of an ANSI outer join restricts the rows that are included in the query result. For more information, see "ANSI Outer Joins" on page 4-25.

➤ ***Note***

You can also use ANSI syntax to join views.

Sybase ANSI join syntax do not support the **using** clause.

Correlation Name And Column Referencing Rules for ANSI Joins

The following are the correlation name and column reference rules specifically for ANSI joins. For more information about correlation names, see “Self-Joins and Correlation Names” on page 4-11.

- If a correlation name is given to a table or view, references to the column or view must always use the same correlation name, rather than the table name or view name. That is, you cannot name a table in a query with a correlation name and then use its table name later. The following example correctly uses the correlation name *t* to specify the table where its pub_id column is specified:

  ```
  select title, t.pub_id, pub_name
  from titles t left join publishers p
  on t.pub_id = p.pub_id
  ```

  However, the following example incorrectly usesthe table name instead of the correlation name for the *titles* table (*t.pub_id*) in the **on** clause of the query and produces the subsequent error message:

  ```
  select title, t.pub_id, pub_name
  from titles t left join publishers p
  on titles.pub_id = p.pub_id
  ```

  ```
  Msg 107, Level 15, State 1:
  Server ‘server_name’, Line 1:
  The column prefix ‘t’ does not match with a table
  name or alias name used in the query. Either the
  table is not specified in the FROM clause or it
  has a correlation name which must be used instead.
  ```

- The restriction specified in the **on** clause can reference:
  - Columns that are specified in the joined table’s reference
  - Columns that are specified in joined tables that are contained in the ANSI join (for example, in a nested join)
  - Correlation names in subqueries for tables specified in outer query blocks
- The condition specified in the **on** clause **cannot** reference:
  - Columns that are introduced in ANSI joins that *contain* another ANSI join (typically when the joined table produced by the second join is joined with the first join).

  The following is an example of an illegal column reference and produces an error:

```
select *
from titles left join titleauthor
on titles.title_id=roysched.title_id    /*join #1*/
left join roysched
on titleauthor.title_id=roysched.title_id    /*join #2*/
where titles.title_id != 'PS7777'
```

In this query, the first left join cannot reference the "*roysched.title_id*" column because this column is not introduced until the second join. This query can be correctly written as:

```
select *
from titles
left join (titleauthor
left join roysched
on titleauthor.title_id = roysched.title_id) /*join #1*/
on titles.title_id = roysched.title_id      /*join #2*/
where titles.title_id != 'PS7777'
```

And another example:

```
select title, price, titleauthor.au_id, titleauthor.title_id,
pub_name, publishers.city
from roysched, titles
left join titleauthor
on roysched.title_id=titleauthor.title_id
left join authors
on titleauthor.au_id=roysched.au_id, publishers
```

In this query, neither the *roysched* table or the *publishers* table are part of either left joins. Because of this, neither left join can refer to columns in either the *roysched* or *publishers* tables as part of their **on** clause condition.

ANSI Inner Joins

Joins that produce a result set that includes only the rows of the joining tables that meet the restriction are called **inner joins**. Rows that do not meet the join restriction are not included in the joined table. If you require the joined table to include all the rows from one of the tables, regardless of whether they meet the restriction, use an outer join. See "ANSI Outer Joins" on page 4-25, for more information.

Adaptive Server supports the use of both Transact-SQL inner joins and ANSI inner joins. Queries using Transact-SQL inner joins separate the tables being joined by commas and list the join comparisons and restrictions in the **where** clause. For example:

```
select au_id, titles.title_id, title, price
from titleauthor, titles
where titleauthor.title_id = titles.title_id
and price > $15
```

For information about writing T-SQL inner joins, see "How Joins Are Structured" on page 4-3.

ANSI-standard **inner joins** syntax is:

```
select select_list
   from table1 inner join table2
   on join_condition
```

For example, the following use of **inner join** is equivalent to the Transact SQL join above:

```
select au_id, titles.title_id, title, price
from titleauthor inner join titles
on titleauthor.title_id = titles.title_id
and price > 15
au_id         title_id  title                        price
----------    --------  -------------------------    -----
213-46-8915   BU1032    The Busy Executive's Datab  19.99
409-56-7008   BU1032    The Busy Executive's Datab  19.99
. . .
172-32-1176   PS3333    Prolonged Data Deprivation  19.99
807-91-6654   TC3218    Onions, Leeks, and Garlic:  20.95
(11 rows affected)
```

The two methods of writing joins, ANSI or T-SQL, are equivalent. For example, there is no difference between the result sets produced by the following queries:

```
select title_id, pub_name
from titles, publishers
where titles.pub_id = publishers.pub_id
```

and

```
select title_id, pub_name
from titles left join publishers
on titles.pub_id = publishers.pub_id
```

An inner join can be part of an **update** or **delete** statement. For example, the following query multiplies the price for all the titles published in California by 1.25:

```
update titles
  set price = price * 1.25
  from titles inner join publishers
  on titles.pub_id = publishers.pub_id
  and publishers.state = "CA"
```

The Join Table of An Inner Join

An ANSI join specifies which tables or views to join in the query. The table references specified in the ANSI join comprise the joined table. For example, the join table of the following query includes the *title*, *price*, *advance*, and *royaltyper* columns:

```
select title, price, advance, royaltyper
from titles inner join titleauthor
on titles.title_id = titleauthor.title_id
title          price     advance      royaltyper
-----------    -------   ----------   ----------
The Busy...    19.99     5,000.00     40
The Busy...    19.99     5,000.00     60
. . .
Sushi, A...    14.99     8,000.00     30
Sushi, A...    14.99     8,000.00     40
(25 rows affected)
```

If a joined table is used as a table reference in an ANSI inner join, it becomes a **nested** inner join. ANSI nested inner joins follow the same rules as ANSI outer joins.

At most, a query can reference 50 user tables (or 14 work tables) on each side of a union, including:

- Base tables or views listed in the **from** clause
- Each correlated reference to the same table (self-join)
- Tables referenced in subqueries
- Base tables referenced by the views or nested views
- Tables being created with **into**

The on Clause of an ANSI Inner Join

The **on** clause of an ANSI inner join specifies the conditions used when the tables or views are joined. Although you can join on any column of a table, your performance may be better if these columns are indexed. Often, you must use qualifiers (table or correlation names) to uniquely identify the columns and the tables to which they belong. For example:

```
from titles t left join titleauthor ta
on t.title_id = ta.title_id
```

This **on** clause eliminates rows from both tables where there is no matching *title_id*. For more information about correlation names, see "Self-Joins and Correlation Names" on page 4-11.

The **on** clause often compares the ANSI joins tables, as in the third and fourth line of the following query:

```
select title, price, pub_name
from titles inner join publishers
on titles.pub_id = publishers.pub_id
and total_sales > 300
```

The join restriction specified in this **on** clause removes all rows from the join table that do not have sales greater than 300. The **on** clause can also specify search arguements, as illustrated in the fourth line of the query.

ANSI inner joins restrict the result set the similarly whether the condition is placed in the **on** clause or the **where** clause (unless they are nested in an outer join). That is, the following queries produce the same result sets:

```
select stor_name, stor_address, ord_num, qty
from salesdetail inner join stores
on salesdetail.stor_id = stores.stor_id
where qty > 3000
```

and

```
select stor_name, stor_address, ord_num, qty
from salesdetail inner join stores
on salesdetail.stor_id = stores.stor_id
and qty > 3000
```

A query is usually more readable if the restriction is placed in the **where** clause; this explicitly tells users which rows of the join table are included in the result set.

ANSI Outer Joins

Joins that produce a joined table that includes all rows from the outer table, regardless of whether the **on** clause produces matching rows or not, are called **outer joins**. Inner joins and equijoins produce a result set that includes only the rows from the tables where there are matching values in the join clause. There are times, however, when you want to include not only the matching rows, but also the rows from one of the tables where there are no matching rows in the

second table. This type of joins is an outer join. In an outer join, rows that do not meet the **on** clause conditions are included in the joined table with null-supplied values for the inner table of the outer join. The inner table is also referred to as the null-supplying member. The roles of the inner and outer tables are described in Figure 4-1:

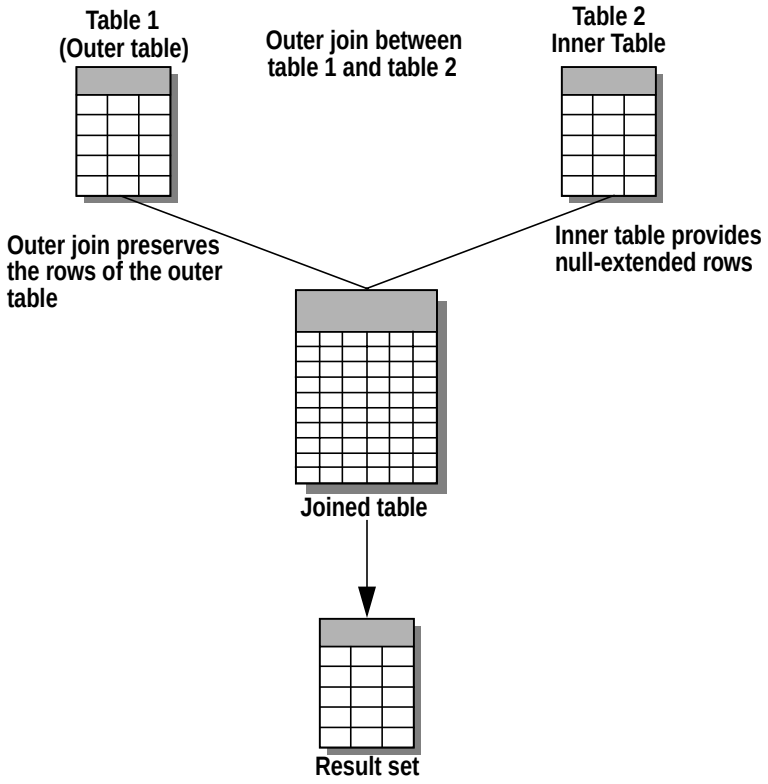


Figure 4-1: The roles of an outer and inner table in an outer join

Sybase recommends that applications use ANSI outer joins because they unambiguously specify whether the **on** or **where** clause contains the predicate. This is ambiguous when you use Transact-SQL outer joins.

This section discusses only the ANSI outer joins; for information about Transact-SQL outer joins, see “Transact SQL Outer Joins” on page 4-37.

➤ ***Note***

Queries that contain ANSI outer joins cannot contain Transact-SQL outer joins, and vice versa. However, a query with ANSI outer joins can reference a view that contains a Transact-SQL outer join, and vice-versa.

ANSI outer joins syntax is:

```
select select_list
   from table1 {left | right} [outer] join table2
   on predicate
   [join restriction]
```

Left joins retain all the rows of the table reference listed on the left of the join clause; right joins retain all the rows of the table reference on the right of the join clause. In left joins, the left table reference is referred to as the outer table or row-preserving table.

The following example determines the authors who live in the same city as their publishers:

```
select au_fname, au_lname, pub_name
from authors left join publishers
on authors.city = publishers.city
```

```
au_fname   au_lname    pub_name
---------  ----------  -------------
Johnson    White       NULL
Marjorie   Green       NULL
Cheryl     Carson      Algodata Infosystems
. . .
Abraham    Bennet      Algodata Infosystems
. . .
Anne       Ringer      NULL
Albert     Ringer      NULL
(23 rows affected)
```

The result set contains all the authors from the *authors* table. The authors who do not live in the same city as their publishers produce null values in the *pub_name* column. Only the authors who live in the same city as their publishers, Cheryl Carson and Abraham Bennet, produce a non-null value in the *pub_name* column.

You can rewrite a left outer join as a right outer join by reversing the placement of the tables in the **from** clause. Also, if the **select** statement specifies "select *", you must write an explicit list of all the column names, otherwise the columns in the result set may not be in the same order for the rewritten query.

Following is the previous example rewritten as a right outer join, which produces the same result set as the left outer join above:

```
select au_fname, au_lname, pub_name
from publishers right join authors
on authors.city = publishers.city
```

Should the Predicate Be in the *on* Or *where* Clause?

The result set of an ANSI outer join depends on whether you place the restriction, in the **on** or the **where** clause. The **on** clause defines the result set of a joined table and which rows of this joined table have null-supplied values; the **where** clause defines which rows of the joined table are included in the result set.

Whether you use an **on** or a **where** clause in your join condition depends on what you want your result set to include. The following examples illustrate the steps for making a decision to place the predicate in the **on** or the **where** clause.

Predicate Restrictions on an Outer Table

The following query places a restriction on the outer table in the **where** clause. Because the restriction is applied to the result of the outer join, it removes all the rows for which the condition is **not** true:

```
select title, titles.title_id, price, au_id
from titles left join titleauthor
on titles.title_id = titleauthor.title_id
where title.price > 20.0

title               title_id price      au_id
------------------- -------- ---------- -----------------
But Is It User F... PC1035        22.95 238-95-7766
Computer Phobic ... PS1372        21.59 724-80-9391
Computer Phobic ... PS1372        21.59 756-30-7391
Onions, Leeks, a... TC3218        20.95 807-91-6654
(4 rows affected)
```

Four rows meet the criteria and only these are included in the result set.

However, if you move this restriction on the outer table to the **on** clause, the result set includes all the rows that meet the **on** clause condition. Rows from the outer table that do not meet the condition are null-extended:

```
select title, titles.title_id, price, au_id
from titles left join titleauthor
on titles.title_id = titleauthor.title_id
and title.price > $20.00
```

```
title                  title_id   price   au_id
--------------------   ---------  ------  ---------------
The Busy Executive's   BU1032     19.99   NULL
Cooking with Compute   BU1111     11.95   NULL
You Can Combat Compu   BU2075      2.99   NULL
Straight Talk About    BU7832     19.99   NULL
Silicon Valley Gastro  MC2222     19.99   NULL
The Gourmet Microwave  MC3021      2.99   NULL
The Psychology of Com  MC3026      NULL   NULL
But Is It User Friend  PC1035     22.95   238-95-7766
Secrets of Silicon Va  PC8888     20.00   NULL
Net Etiquette          PC9999      NULL   NULL
Computer Phobic and    PS1372     21.59   724-80-9391
Computer Phobic and    PS1372     21.59   756-30-7391
Is Anger the Enemy?    PS2091     10.95   NULL
Life Without Fear      PS2106      7.00   NULL
Prolonged Data Depri   PS3333     19.99   NULL
Emotional Security:    PS7777      7.99   NULL
Onions, Leeks, and Ga  TC3218     20.95   807-91-6654
Fifty Years in Buckin  TC4203     11.95   NULL
Sushi, Anyone?         TC7777     14.99   NULL
(19 rows affected)
```

Moving the restriction to the **on** clause added 15 null-supplied rows to the result set.

Generally, if your query uses a restriction on an outer table, and you want the result set to remove only the rows for which the restriction is false, you should probably place the restriction in the **where** clause to limit the rows of the result set. Queries that place the restriction for an outer table in a **where** clause remove rows from the result set for which the restriction is not true. Outer table predicates are not used for index keys if they are in the **on** clause.

Whether you place the restriction on an outer table in the **on** or **where** clause ultimately depends on the information you need the query to return. If you only want the result set to include only the rows for which the restriction is true, you should place the restriction in the **where** clause. However if you require the result set to include all the rows of the outer table, regardless of whether they satisfy the restriction, you should place the restriction in the **on** clause.

Restrictions on an Inner Table

The following query includes a restriction on an inner table in the **where** clause:

```
select title, titles.title_id, titles.price, au_id
from titleauthor left join titles
on titles.title_id = titleauthor.title_id
where titles.price > 20.0
```

```
title            title_id   price    au_id
-------------    --------   -----    -----------
But Is It U...   PC1035     22.95    238-95-7766
Computer Ph...   PS1372     21.59    724-80-9391
Computer Ph...   PS1372     21.59    756-30-7391
Onions, Lee...   TC3218     20.95    807-91-6654
(4 rows affected)
```

Because the restriction of the **where** clause is applied to the result set **after** the join is made, it removes all the rows for which the restriction is **not** true. Put another way, the **where** clause is not true for all null-supplied valuces and removes them. A join that places its restriction in the **where** clause is effectively an inner join.

However, if you move the restriction to the **on** clause, it is applied during the join and is utilized in the production of the joined table. In this case, the result set includes all the rows of the inner table for which the restriction is true, plus all the rows of the outer table, which are null-extended if they do not meet the restriction criteria:

```
select title, titles.title_id, price, au_id
from titleauthor left join titles
on titles.title_id = titleauthor.title_id
and price > $20.00
```

```
title      title_id   price           au_id
---------  ---------  -----------     -----------
NULL       NULL       NULL            172-32-1176
NULL       NULL       NULL            213-46-8915
. . .
Onions,    TC3218     20.95           807-91-6654
. . .
NUL        NULL       NULL            998-72-3567
NULL       NULL       NULL            998-72-3567
(25 rows affected)
```

This result set includes 21 rows that the previous example did not include.

Generally, if your query requires a restriction on an inner table (for example “and price > $20.00” in query above), you probably want to

place the condition in the **on** clause; this preserves the rows of the outer table. If you include a restriction for an inner table in the **where** clause, the result set might not include the rows of the outer table.

Like the criteria for the placement of a restriction on an outer table, whether you place the restriction for an inner table in the **on** or **where** clause ultimately depends on the result set you want. If you are interested only in the rows for which the restriction is true, and not including the full set of rows for the outer table, place the restriction in the **where** clause. However, if you require the result set to include all the rows of the outer table, regardless of whether they satisfy the restriction, you should place the restriction in the **on** clause.

Restrictions Included in Both an Inner and Outer Table

The restriction in the **where** clause of the following query includes both the inner and outer tables:

```
select title, titles.title_id, price, price*qty, qty
from salesdetail left join titles
on titles.title_id = salesdetail.title_id
where price*qty > $3000.0
```

```
title              title_id   price  qty
-----------------  --------   -----  ---------  -----
Silicon Valley Ga  MC2222     19.99  40,619.68  2032
But Is It User Fr  PC1035     22.95  45,900.00  2000
But Is It User Fr  PC1035     22.95  45,900.00  2000
But Is It User Fr  PC1035     22.95  49,067.10  2138
Secrets of Silico  PC8888     20.00  40,000.00  2000
Prolonged Data De  PS3333     19.99  53,713.13  2687
Fifty Years in Bu  TC4203     11.95  32,265.00  2700
Fifty Years in Bu  TC4203     11.95  41,825.00  3500
(8 rows affected)
```

Placing the restriction in the **where** clause eliminates the following rows from the result set:

- The rows for which the restriction “price*qty>$30000.0” is false
- The rows for which the restriction “price*qty>$30000.0” is unknown because *price* is null

To keep the unmatched rows of the outer table, move the restriction into the **on** clause, as in this example:

```
select title, titles.title_id, price, price*qty, qty
from salesdetail left join titles
on titles.title_id = salesdetail.title_id
and price*qty > $30000.0
```

```
title                title_id   price  qty
-----------------    --------   -----  ---------  -----
NULL                 NULL       NULL   NULL       75
NULL                 NULL       NULL   NULL       75
. . .
Secrets of Silico    PC8888     20.00  40,000.00  2000
. . .
NULL                 NULL       NULL   NULL       300
NULL                 NULL       NULL   NULL       400
(116 rows affected)
```

This query retains all 116 rows of the *salesdetail* table in the result set, and null-extending the rows that do not meet the restriction.

Where you place the restriction that includes both the inner and outer table depends on the result set you want. If you are interested in only the rows for which the restriction is true, place the restriction in the **where** clause. However, if you want to include all the rows of the outer table, regardless of whether they satisfy the restriction, place the restriction in the **on** clause.

Nested ANSI Outer Joins

Nested outer joins use the result set of one outer join as the table reference for another. For example:

```
select t.title_id, title, ord_num, sd.stor_id, stor_name
from (salesdetail sd
left join titles t
on sd.title_id = t.title_id)   /*join #1*/
left join stores
on sd.stor_id = stores.stor_id /*join #2*/
```

```
title_id  title         ord_num  stor_id  stor_name
--------  ------------  -------  -------  ------------------------
TC3218    Onions, L...  234518   7896     Fricative Bookshop
TC7777    Sushi, An...  234518   7896     Fricative Bookshop
. . .
TC4203    Fifty Yea...  234518   6380     Eric the Read Books
MC3021  The Gourmet...  234518   6380     Eric the Read Books
(116 rows affected)
```

In this example, the joined table between the *salesdetail* and *titles* tables is logically prduced first and is then joined with the columns of the *stores* table where *salesdetail.stor_id* equals *stores.stor_id*. Semantically, each level of nesting in a join creates a joined table and is then used for the next join.

In the query above, because the first outer join becomes an operator of the second outer join, this query is a **left-nested outer join**.

This example shows a **right nested outer join**::

```
select stor_name, qty, date, sd.ord_num
from salesdetail sd left join (stores /*join #1 */
left join sales on stores.stor_id = sales.stor_id) /*join #2 */
on stores.stor_id = sd.stor_id
where date > "1/1/1990"
```

```
stor_name      qty  date                  ord_num
------------   ---- ------------------    --------------------
News & Brews   200  Jun 13 1990 12:00AM  NB-3.142
News & Brews   250  Jun 13 1990 12:00AM  NB-3.142
News & Brews   345  Jun 13 1990 12:00AM  NB-3.142
. . .
Thoreau Read   1005 Mar 21 1991 12:00AM  ZZ-999-ZZZ-999-0A0
Thoreau Read   2500 Mar 21 1991 12:00AM  AB-123-DEF-425-1Z3
Thoreau Read   4000 Mar 21 1991 12:00AM  AB-123-DEF-425-1Z3
```

In this example, the second join (between the *stores* and the *sales* tables) is logically produced first, and is joined with the *salesdetail* table. Because the second outer join is used as the table reference for the first outer join, this query is a **right-nested outer join**.

If the **on** clause for the first outer join ("from salesdetail. . .") fails, it supplies null values to both the *stores* and *sales* tables in the second outer join.

Parentheses in Nested Outer Joins

Nested outer joins produce the same result set with or without parenthesis. Large queries with many outer joins can be much more readable for users if the joins are structured using parentheses.

The on Clause in a Nested Outer Joins

The placement of the **on** clause in a nested outer join determines which join is logically processed first. Reading from left to right, the first **on** clause is the first join to be defined.

In this example, the position of the **on** clause in the first join (in parentheses indicates that it is the table reference for the second join, so it is defined **first**, producing the table reference to be joined with the *authors* table:

```
select title, price, au_fname, au_lname
from (titles left join titleauthor
on titles.title_id = titleauthor.title_id)   /*join #1*/
left join authors
on titleauthor.au_id = authors.au_id      /*join #2*/
and titles.price > $15.00
```

```
title            price      au_fname      au_lname
---------------  ---------  ------------  -------------
The Busy Exe...  19.99      Marjorie      Green
The Busy Exe...  19.99      Abrahame      Bennet
. . .
Sushi, Anyon...  14.99      Burt          Gringlesby
Sushi, Anyon...  14.99      Akiko         Yokomoto
(26 rows affected)
```

However, if the **on** clauses are in different locations, the joins are evaluated in a different sequence, but still produce the same result set (note that this is only for explanatory purposes only; realistically, if joined tables are logically produced in a different order it is highly unlikely that they will produce the same result set):

```
select title, price, au_fname, au_lname
from titles left join
(titleauthor left join authors
on titleauthor.au_id = authors.au_id)        /*join #2*/
on titles.title_id = titleauthor.title_id    /*join #1*/
and au_lname like “Yokomoto”
```

```
title                    price    au_fname     au_lname
----------------------   -------  -----------  -----------
The Busy Executive’s     19.99    Marjorie     Green
The Busy Executive’s     19.99    Abraham      Bennet
. . .
Sushi, Anyone?           14.99    Burt         Gringlesby
Sushi, Anyone?           14.99    Akiko        Yokomoto

(26 rows affected)
```

The position of the **on** clause of the first join (the last line of the query) indicates that the **second** left join is a table reference of the first join, so it is performed first. That is, the result of the second left join is joined with the *titles* table.

Converting Outer Joins with Join-Order Dependency

Almost all Transact-SQL outer joins written for previous versions of Adaptive Server that are run on a 12.0 and later Adaptive Server produce the same result set. However, there is a category of outer join queries whose result sets depend on the join order chosen during optimization. Depending on where in the query the predicate is evaulated, these queries may produce different result sets when they

are issued using the current version of Adaptive Server. The result sets they return are determined by the ANSI rules for assigning predicates to joins.

Predicates cannot be evaluated until all the tables they reference are processed. That is, in the following query the predicate "and titles.price > 20" cannot be evaluated until the *titles* table is processed:

```
select title, price, au_ord
from titles, titleauthor
where titles.title_id *= titleauthor.title_id
and titles.price > 20
```

Predicates in previous versions of Adaptive Server where evaluated according to the following semantics:

- If the predicate was evaluated on the inner table of an outer join, the predicate had **on** clause semantics.
- If the predicate was evaluated with a table that is outer to all outer joins, or is join order independent, the predicate had **where** clause semantics.

➤ ***Note***

Before you run Adaptive Server in a production environment, make sure you start it with traceflag 4413 and run any queries that you think may be join-order dependent in pre-12.0 versions of Adaptive Server. Adaptive Server started with trace flag 4413 issues a message similar to the following when you run a query that is join-order dependent in a pre-12.0 version of Adaptive Server:

```
Warning: The results of the statement on line %d are
join-order independent. Results may differ on pre-12.0
releases, where the query is potentially join-order
dependent.
```

Make sure you resolve dependencies your applications have on result sets of join-order queries produced by pre-12.0 Adaptive Server.

When Do Join-Order Dependent Outer Joins Effect Me?

Generally, you will not have any problem from join-order dependent queries because predicates typically only reference:

- An outer table, which is evaluated using **where** clause semantics
- An inner table, which is evaluated using **on** clause semantics
- The inner table and tables upon which the inner table is dependent

which do not produce join-order dependent outer joins. However, Transact-SQL queries that have either of the following characteristics may produce a different result set after they are translated to an ANSI outer join:

- Predicates that contain **or** statements and reference an inner table of an outer join and another table that this inner table is not dependent on.
- Inner table attributes that are referenced in the same predicate as a table which is not in the inner table's join order dependency
- An inner table referenced in a subquery as a correlated reference

The following examples demonstrate the issues of translating Transact-SQL queries with join-order dependency to ANSI outer join queries.

Example:

```
select title, price, authors.au_id, au_lname
from titles, titleauthor, authors
where titles.title_id =* titleauthor.title_id
and titleauthor.au_id = authors.au_id
and (titles.price is null or authors.postalcode = '94001')
```

This query is join-order independent because the outer join references both the *titleauthor* and the *titles* tables, and the *authors* table can be joined with these tables according to three join orders:

- *authors*, *titleauthors*, *titles* (as part of the **on** clause)
- *titleauthors*, *authors*, *titles* (as part of the **on** clause)
- *titleauthors*, *titles*, *authors* (as part of the **where** clause)

This query produces the following message:

```
Warning: The results of the statement on line 1 are join-order
independent. Results may differ on pre-12.0 releases, where the
query is potentially join-order dependent. Use trace flag 4413
to suppress this warning message.
```

Following is the ANSI equivalent:

```
select title, price, authors.au_id, au_lname
from titles right join
(titleauthor inner join authors
on titleauthor.au_id = authors.au_id)
on titles.title_id = titleauthor.title_id
where (titles.price is null or authors.postalcode = '94001')
```

Another example:

```
select title, au_fname, au_lname, titleauthor.au_id, price
from titles, titleauthor, authors
where authors.au_id *= titleauthor.au_id
and titleauthor.au_ord*titles.price > 40
```

The query is join-order dependent for the same reason as the previous example. Here is the ANSI equivalent:

```
select title, au_fname, au_lname, titleauthor.au_id, price
from titles,(authors left join titleauthor
on titleauthor.au_id = authors.au_id)
where titleauthor.au_ord*titles.price > 40
```

Transact SQL Outer Joins

Transact SQL includes syntax for both left and right outer joins. The **left outer join**, ***=**, selects all rows from the first table that meet the statement's restrictions. The second table generates values if there is a match on the join condition. Otherwise, the second table generates null values.

For example, the following left outer join lists all authors and finds the publishers (if any) in their city:

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

The **right outer join**, **=***, selects all rows from the second table that meet the statement's restrictions. The first table generates values if there is a match on the join condition. Otherwise, the first table generates null values.

➤ ***Note***

You cannot include a Transact-SQL outer join in a **having** clause.

A table is either an inner or an outer member of an outer join. If the join operator is ***=**, the second table is the inner table; if the join operator is **=***, the first table is the inner table. You can compare a

column from the inner table to a constant as well as using it in the outer join. For example, if you want to find out which *title* has sold more than 4000 copies:

```
select qty, title from salesdetail, titles
where qty > 4000
and titles.title_id *= salesdetail.title_id
```

However, the inner table in an outer join cannot also participate in a regular join clause.

An earlier example used a join to find the names of authors who live in the same city as a publisher returns two names: Abraham Bennet 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 is 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 Adaptive 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 values you really want to see and use the outer join to list the rows that did not make the cut. Let us look at the equijoin first and compare it to the outer join. For example, 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)
```

Also, to show 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
NULL    The Psychology of Computer Cooking
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

 (19 rows affected)
```

You can restrict an inner table with a simple clause. The following example lists the authors who live in the same city as a publisher, but excludes the author Cheryl Carson, who would normally be listed as an author living in a publisher's city:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city =* publishers.city
and authors.au_lname != "Carson"
```

```
au_fname     au_lname    pub_name
---------    ---------   ---------------
NULL         NULL        New Age Books
NULL         NULL        Binnet & Hardley
Abraham      Bennet      Algodata Infosystems

(3 rows affected)
```

How Null Values Affect Joins

Null values in tables or views being joined will never match each other. Since *bit* columns do not permit NULLs, 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 result of a join of NULL with any other value is NULL. Because 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 t1

a	b
1	one
NULL	three
4	join4

Table t2

c	d
NULL	two
4	four

Here is the 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

(3 rows affected)
```

The results make it difficult 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
first_pair
------------------------------ ------------------
title_id                       title_id
```

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 does not 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 *Adaptive 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:

You can also express subqueries as join operations. See Chapter 4, "Joins: Retrieving Data from Several Tables."

How Subqueries Work

Subqueries, also called **inner queries**, 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. a **case** expression can also include a subquery.

For example, the following subquery lists the names of all authors whose royalty split is more than $75:

```
select au_fname, au_lname
from authors
where au_id in
   (select au_id
    from titleauthor
    where royaltyper > 75)
```

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."

You can formulate many SQL statements that include a subquery 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. (Adaptive Server converts some subqueries into joins before processing them.)

The result of a subquery that returns no values is NULL. If a subquery returns NULL, the query failed.

Subquery Syntax

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 by its syntax:

```
(select [all | 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])
```

Subquery Restrictions

A subquery is subject to the following restrictions:

- The *subquery_select_list* must consist of only one column name, except in the **exists** subquery, in which case the asterisk (*) is usually used in place of the single column name. Do not specify more than one column name. Be sure to qualify column names with table or view names if there is ambiguity about the table or view to which they belong.
- 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. Alternatively, you can write many statements that contain subqueries as joins; Adaptive Server processes such statements as joins.

- In Transact-SQL, a subquery can appear almost anywhere an expression can be used, if it returns a single value.
- You cannot use subqueries in an **order by**, **group by**, or **compute by** list.
- You cannot include a **for browse** clause or a **union** in a subquery.
- 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.
- The **where** clause of a subquery can only contain an aggregate function if the subquery is in a **having** clause of an outer query and the aggregate value is a column from a table in the **from** clause of the outer query.
- The sum of the maximum lengths of all the columns specified by a subquery cannot exceed 256 bytes.

Example of Using a Subquery

Using a query to find the books that have the same price as *Straight Talk About Computers* involves two steps. First, you find the price of *Straight Talk*:

```
select price
from titles
where title = "Straight Talk About Computers"

price
-------------
       $19.99

(1 row affected)
```

Then you 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 same problem in only one step, using the 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)
```

Qualifying Column Names

Column names in a statement are implicitly qualified by the table referenced in the **from** clause at the same level. In the following example, the table name *publishers* implicitly qualifies the *pub_id* column in the **where** clause of the outer query *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")
```

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. You can nest up to 16 subqueries 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
Locksley                Chastity
Hunter                  Sheryl

(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

You can nest subqueries 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

You can use subqueries 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, you can substitute a subquery almost anyplace where you can use an expression in a **select**, **update**, **insert**, or **delete** statement. For example, a subquery can compare with a column from the inner table of an outer join.

You cannot use a subquery in an **order by** list or as an expression in the **values** list in an **insert** statement.

The following statement shows how to find 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 had more than 5000 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

(4 rows affected)
```

Types of Subqueries

There are two basic types of subqueries:

- **Expression subqueries** are introduced with an unmodified comparison operator, must return a single value, and can be used almost anywhere an expression is allowed in SQL.
- **Quantified predicate subqueries** operate on lists introduced with **in** or with a comparison operator modified by **any** or **all**. Quantified predicate subqueries return 0 or more values. This type is also used as an existence test, introduced with **exists**.

Subqueries of either type are either noncorrelated or correlated (repeating).

- 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 Adaptive Server actually processes statements with subqueries. Noncorrelated subqueries can be alternatively stated as joins and are processed as joins by Adaptive 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 include:

- Subqueries in a **select** list (introduced with **in**)
- Subqueries in a **where** or **having** clause connected by a comparison operator (**=**, **!=**, **>**, **>=**, **<**, **<=**)

Expression subqueries 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]

An expression consists of a subquery or any combination of column names, constants, and functions connected by arithmetic or bitwise operators.

The *comparison_operator* is one of the following:

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

If you use a column name in the **where** or **having** clause of the outer statement, make sure a column name in the *subquery_select_list* is join compatible with it.

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, Adaptive Server returns an error message.

Ideally, 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, because these return a single value.

For example, this statement finds the names of 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 a list of 0 and higher values, 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.

There are three types of quantified predicate subqueries:

- **any** / **all** subqueries. 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]

- **in** / **not in** subqueries. 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]

- **exists** / **not exists** subqueries. 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 Adaptive Server allows the keyword **distinct** in quantified predicate subqueries, it always processes the subquery as if **distinct** were not included.

Subqueries with *any* and *all*

The keywords **all** and **any** modify a comparison operator that introduces a subquery.

When **any** is used with **<**, **>**, or **=** with a subquery, it returns results when any value retrieved in the subquery matches the value in the **where** or **having** clause of the outer statement.

When **all** is used with **<** or **>** in a subquery, it returns results when all values retrieved in the subquery match the value in the **where** or **having** clause of the outer statement.

The syntax for **any** and **all** is:

```
{where | having} [not]
   expression comparison_operator {any |all}
   (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 you introduce a subquery 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?"

You can paraphrase this question 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

The **> all** operator means that the value in the column that introduces the subquery must be greater than each of the values returned by the subquery, in order for a row to satisfy the condition in the outer query.

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 type = "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 that 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, in order for a row to satisfy the outer query.

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 the value in the column that introduces the subquery must be greater than at least one of the values in the list returned by the subquery, in order for a row to satisfy the outer query.

The following example 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
--------------------------------------------------
The Busy Executive's Database Guide
Cooking with Computers: Surreptitious Balance
    Sheets
You Can Combat Computer Stress!
Straight Talk About Computers
The Gourmet Microwave
But Is It User Friendly?
Secrets of Silicon Valley
Computer Phobic and Non-Phobic Individuals:
```

```
    Behavior Variations
Is Anger the Enemy?
Life Without Fear
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)
```

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, this 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, to find the authors who live in a city where no publisher is located:

```
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
--------------          ------------
White                   Johnson
Green                   Marjorie
O'Leary                 Michael
Straight                Dick
Smith                   Meander
Dull                    Ann
Gringlesby              Burt
Locksley                Chastity
Greene                  Morningstar
Blotchet-Halls          Reginald
Yokomoto                Akiko
del Castillo            Innes
DeFrance                Michel
Stringer                Dirk
MacFeather              Stearns
Karsen                  Livia
Panteley                Sylvia
Hunter                  Sheryl
McBadden                Heather
Ringer                  Anne
Ringer                  Albert

(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 5-20:

```
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 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:

```
select pub_name
    from publishers
    where $100.00 not in
        (select price
         from titles
         where titles.pub_id = publishers.pub_id)
pub_name
------
New Age Books

(1 row affected)
```

New Age Books is the only publisher that 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*

Use the **exists** keyword with a subquery to test for the existence of some result from the subquery. The syntax follows:

{where | having} [not] exists (*subquery*)

That is, 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 5-28).

Although you cannot express some queries formulated with **exists** in any other way, you can express all queries that use **in** or a comparison operator modified by **any** or **all** 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
--------------------------------------------------
You Can Combat Computer Stress!
Is Anger the Enemy?
Life Without Fear
Prolonged Data Deprivation: Four Case Studies
Emotional Security: A New Algorithm
The Busy Executive's Database Guide
Cooking with Computers: Surreptitious Balance
    Sheets
Straight Talk About Computers
But Is It User Friendly?
Secrets of Silicon Valley
Net Etiquette

(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*

You can use subqueries that are introduced with **exists** and **not exists** 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

You can evaluate many of the previous queries by executing the subquery once and substituting the resulting values 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.

This example finds 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
--------------  ----------
White           Johnson
Green           Marjorie
Carson          Cheryl
Straight        Dick
Locksley        Chastity
Blotchet-Hall   Reginald
del Castillo    Innes
Panteley        Sylvia
Ringer          Albert

(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 Adaptive 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, suppose Transact-SQL first examines the row for Johnson White. Then, *authors.au_id* takes the value "172-32-1176," which Transact-SQL substitutes for the inner query:

```
select royaltyper
from titleauthor
where au_id = "172-32-1176"
```

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 Johnson White 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

You can use a correlated subquery 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

A **datatype** defines the kind of information each column in a table holds, and how that information is stored. You can use Adaptive Server system datatypes when you are defining columns, or you can create and use user-defined datatypes.

This chapter discusses:

How Transact-SQL Datatypes Work

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 *int* (integer) datatype stores whole numbers in the range of plus or minus 2^{31}, and the *tinyint* (tiny integer) datatype stores whole numbers between 0 and 255 only.

Adaptive 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 6-15.)

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 next 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 **sp_help** system procedure.

Using System-Supplied Datatypes

The following table lists the system-supplied datatypes provided for various types of information, the synonyms recognized by Adaptive Server, and the range and storage size for each. The system datatypes are printed in lowercase characters, although Adaptive Server allows you to enter them in either uppercase or lowercase. (*timestamp* and *sysname*, like all user-defined datatypes, are case-sensitive.) Most Adaptive Server-supplied datatypes are not reserved words and can be used to name other objects.

Table 6-1: Adaptive 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: Adaptive 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 fewer	*n*
varchar(n)	*character varying, char varying*	255 characters or fewer	actual entry length
nchar(n)	*national character, national char*	255 bytes or fewer (single-byte character set)	*n* * *@@ncharsize*
nvarchar(n)	*nchar varying, national char varying, national character varying*	255 bytes or fewer (single-byte character set)	*@@ncharsize* * number of characters
text		2^{31} -1 (2,147,483,647) bytes or fewer	0 or multiple of 2K
Binary			
binary(n)		255 bytes or fewer	*n*
varbinary(n)		255 bytes or fewer	actual entry length
image		2^{31} -1 (2,147,483,647) bytes or fewer	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

Adaptive 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

Use the 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])]
```

Adaptive 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 1–38 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 of 0–38 digits or use the default scale of 0 digits.

Exact numeric types with a scale of 0 are displayed without a decimal point. You cannot enter a value that exceeds either the precision or the scale for the column.

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

The numeric types *float*, *double precision*, and *real* store numeric data that can tolerate rounding during arithmetic operations.

Approximate numeric datatypes store slightly inaccurate representations of real numbers, stored as binary fractions. Anytime an approximate numeric value is displayed, printed, transferred between hosts, or used in calculations, the numbers lose precision. Note that **isql** displays only six significant digits after the decimal point and rounds the remainder. For more information on precision and approximate numeric datatypes, see the *Adaptive Server Reference Manual*.

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.

Table 6-3 shows the range, display precision, and storage size for each approximate numeric type.

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, and 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 255 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

Adaptive Server truncates entries to the specified column length without warning or error, unless you set **string_rtruncation on**. See the **set** command in the *Adaptive 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, Adaptive Server 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, it is 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.

text Datatype

The *text* datatype stores up to 2,147,483,647 bytes of printable characters on linked lists of separate data pages. Each page stores a maximum of 1800 bytes of data.

To save storage space, define *text* columns as NULL. When you initialize a *text* column with a non-null **insert** or **update**, Adaptive Server assigns a text pointer and allocates an entire 2K data page to hold the value.

You cannot use the *text* datatype:

- For parameters to stored procedures, as values passed to these parameters or for local variables
- For parameters to remote procedure calls (RPCs)
- In **order by**, **compute**, **group by**, or **union** clauses
- In an index
- In subqueries or joins
- In a **where** clause, except with the keyword **like**
- With the **+** concatenation operator
- In the **if update** clause of a trigger

If you are using databases connected with Component Integration Services, there are several differences in the way *text* datatypes are handled. See the *Component Integration Services User's Guide* for more information.

For more information about the *text* datatype, see "Changing text and image Data" on page 8-29.

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. The two digits following "0x" in *binary* and *varbinary* data indicate the type of number: "00" represents a positive number and "01" represents a negative number.

If the input value does not include the "0x", Adaptive Server assumes that the value is an ASCII value and converts it.

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

➤ ***Note***

Adaptive 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 1 byte. If you enter a value longer than *n*, Adaptive 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 they 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, Adaptive Server 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.**

image Datatype

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**, Adaptive Server assigns a text pointer and allocates an entire 2K data page to hold the value. Each page stores a maximum of 1800 bytes.

To save storage space, define *image* columns as NULL. To add *image* data without saving large blocks of binary data in your transaction log, use **writetext**. See the *Adaptive Server Reference Manual* for details on **writetext**.

You cannot use the *image* datatype:

- For parameters to stored procedures, as values passed to these parameters or for local variables
- For parameters to remote procedure calls (RPCs)
- In **order by**, **compute**, **group by**, or **union** clauses
- In an index
- In subqueries or joins
- In a **where** clause, except with the keyword **like**
- With the **+** concatenation operator
- In the **if update** clause of a trigger

If you are using databases connected with Component Integration Services, there are several differences in the way *image* datatypes are handled. See the *Component Integration Services User's Guide* for more information.

For more information about the *image* datatype, see "Changing text and image Data" on page 8-29.

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 Adaptive 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 3 digits.

Table 6-6 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 after midnight.

Enclose date and time information in single or double quotes. You can enter it in either uppercase or lowercase letters and include

spaces between data parts. Adaptive Server recognizes a wide variety of data entry formats, which are described in Chapter 8, "Adding, Changing, and Deleting Data." However, Adaptive Server rejects values such as 0 or 00/00/00, which are not recognized as dates.

The default display format for dates is "Apr 15 1987 10:23PM". You can use the **convert** function for other styles of date display. You can also do some arithmetic calculations on *datetime* values with the built-in date functions, though Adaptive Server may round or truncate millisecond values.

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

The *timestamp* user-defined datatype is necessary for columns 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 used in the system tables. Its definition is:

```
varchar(30) "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 this **user-defined datatype** to create columns. For more information about user-defined datatypes, see "Creating User-Defined Datatypes" on page 6-15.

Converting Between Datatypes

Adaptive 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, Adaptive Server automatically converts *char* expressions to *datetime* for the purposes of a comparison, if they can be interpreted as *datetime* values. However, for the purposes of display, you must use the **convert** function to convert *char* to *int*. Similarly, you must use **convert** on integer data if you want Adaptive 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])
```

In the following example, **convert** displays the *total_sales* column using the *char* datatype, in order to display all sales beginning with the digit 2:

```
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, Adaptive 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
```

```
id                                19
tid                               19
sysname                           19
varchar                           19
nvarchar                          19
char                              20
nchar                             20
timestamp                         21
varbinary                         21
binary                            22
text                              23
image                             24

(30 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 you 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 debits, interest
```

Determining Precision and Scale

For the *numeric* and *decimal* types, each combination of precision and scale is a distinct Adaptive 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*, Adaptive Server

determines the precision and scale of the results as shown in Table 6-8:

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	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, “Defining Defaults and Rules for Data.”

Use the system procedure **sp_addtype** to create user datatypes. It takes as parameters the name of the user datatype being created, the Adaptive 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 Adaptive 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 is 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, but 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 Adaptive 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, Adaptive Server assumes the default length of 1 character.
- The *float* datatype expects a precision in parentheses. If you do not supply one, Adaptive 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, Adaptive 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, Adaptive 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, 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, "Defining Defaults and Rules for Data." For complete information on system procedures, see the *Adaptive Server Reference Manual*.

Creating a User-Defined Datatype with the IDENTITY Property

To create a user-defined datatype with the IDENTITY property, use the **sp_addtype** system procedure. The new type must be based on a physical type of *numeric* with a scale of 0:

```
sp_addtype typename, "numeric (precision , 0)",
"identity"
```

The following example creates a user-defined type, *IdentType*, with the IDENTITY property:

```
sp_addtype IdentType, "numeric(4,0)", "identity"
```

When you create a column from an IDENTITY type, you can specify either **identity** or **not null**—or neither one—in the **create** or **alter table** statement. The column automatically inherits the IDENTITY property.

Following are three different ways to create an IDENTITY column from the *IdentType* user-defined type:

```
create table new_table (id_col IdentType)
create table new_table (id_col IdentType identity)
create table new_table (id_col IdentType not null)
```

➤ ***Note***

If you try to create a column that allows nulls from an IDENTITY type, the **create table** or **alter table** statement fails.

Creating IDENTITY Columns from Other User-Defined Datatypes

You can create IDENTITY columns from user-defined datatypes that do not have the IDENTITY property. The user-defined types must have a physical datatype of *numeric* with a scale of 0 and must be defined as **not null**.

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 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            NULL

(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 create databases and tables, a process called **data definition**. It discusses:

If you do not plan to create your own databases and tables, read about the basic database and table concepts, described in the section ‘‘What Are Databases and Tables?,’’ and about the **use** command, described in the sections “Choosing a Database: use” on page 7-5. Then, skip the rest of this chapter.

For information on managing databases, see the *System Administration Guide*.

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.

Adaptive 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. See the following chapters for information on creating other objects:

- Chapter 9, “Views: Limiting Access to Data”
- Chapter 12, “Defining Defaults and Rules for Data”
- Chapter 14, “Using Stored Procedures”
- Chapter 15, “Using Extended Stored Procedures”
- Chapter 16, “Triggers: Enforcing Referential Integrity”

Columns and **datatypes** define the type of data included in tables; indexes describe how that data is organized in tables. They are not considered database objects by Adaptive Server and are not listed in *sysobjects*. Columns and datatypes are covered in this chapter; indexes are discussed in Chapter 11, “Creating Indexes on Tables.”

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* and *pubs3* databases 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, because it violates the data integrity of *pubs2* or *pubs3*.

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** – 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** – limits or restricts the data values inserted into a table column. You can use triggers or rules to enforce this data integrity.
- **Uniqueness** – 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** – requires that data inserted into a table column must already have matching data in another table column or another column in the same table. A single table can have up to 192 references.

Consistency of data values in the database is another example of data integrity, which is described in Chapter 18, "Transactions: Maintaining Data Consistency and Recovery."

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 can 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 Adaptive Server login name.

To make your experiments with database objects as convenient as possible, the *pubs2* and *pubs3* databases have a "**guest**" user name in their *sysusers* system tables. The scripts that create *pubs2* and *pubs3* grant a variety of permissions to "guest".

The "guest" mechanism means that anyone who has a **login** on Adaptive Server, that is, is listed in *master..syslogins*, has access to *pubs2* and *pub3*, 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* or *pubs3* database, issue the **use** command. Adaptive Server checks whether you are listed under your own name in *pubs2..sysusers* or *pubs3..sysusers*. If not, you are admitted as a guest without any action on your part. If you are listed in the *sysusers* table for *pubs2* or *pubs3*, Adaptive Server admits you as yourself, but may give you different permissions from those of "guest". All the examples in this chapter assume you are being treated as "guest".

Most users can look at the system tables in the *master* database by means of the "guest" mechanism previously described. Users who are 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 *Adaptive Server Reference Manual*.

Using and Creating Databases

A database is a collection of related tables and other database objects—views, indexes, and so on.

When Adaptive Server is first installed, it contains these **system databases**:

- The *master* database controls the user databases and the operation of Adaptive Server as a whole.
- The *sybsystemprocs* database contains the system stored procedures.
- The *sybsystemdb* database contains information about distributed transactions.
- The temporary database, *tempdb*, stores temporary objects, including temporary tables created with the name prefix "*tempdb..*".
- The *model* database is used by Adaptive Server as a template for creating new user databases.

In addition, System Administrators can install the following optional databases:

- *pubs2* – a sample database that contains data representing a publishing operation. You can use this database to test your server connections and learn Transact-SQL. Most of the examples in the Adaptive Server documentation query the *pubs2* database.
- *pubs3* – a version of *pubs2* that uses referential integrity examples. *pubs3* has a new table, *store_employees*, which uses a self-referencing column. Other differences are that dates and area codes have been updated, an IDENTITY column has been added to its *sales* table, its the primary keys in its master tables use nonclustered unique indexes, and the *titles* table has an example of the *numeric* datatype.
- *interpubs* – similar to *pubs2*, but contains French and German data.
- *jpubs* – similar to *pubs2*, but contains Japanese data. Use it if you have installed the Japanese Language Module.

These optional databases are user databases. All of your data is stored in user databases. Adaptive 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*

The **use** command lets you access an existing database. Its syntax is:

```
use database_name
```

For example, to access the *pubs2* database, type:

```
use pubs2
```

This command allows you to access the *pubs2* database only if you are a known user in *pubs2*. Otherwise, Adaptive 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**.

It is likely that you will be automatically connected to the *master* database when you log onto Adaptive Server, so if you want to use another database, issue the **use** *database* command. 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, described under "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 Adaptive Server, and must follow the rules for identifiers described under "Identifiers" on page 1-7. Adaptive Server can manage up to 32,767 databases. You can create only 1 database at a time. The maximum number of segments for any database is 32.

Adaptive 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*.

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
```

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 example 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, 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.

Additional reasons for placing the log on a separate **physical** device from the data tables are:

- 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 1MB:

```
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". To add more space for an existing log, use **alter database** and, in some cases, the **sp_extendsegment** system procedure. See the *Adaptive Server Reference Manual* or the *System Administration Guide* for 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 **for load** option for recovery from media failure or for moving a

database from one machine to another. See the *System Administration Guide* for further details.

Altering the Sizes of Databases

If a database has filled 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 the objects 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. To change this segment mapping, 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*.

Dropping Databases

Use the **drop database** command to remove a database. **drop database** deletes the database and all of its contents from Adaptive Server, frees the storage space that had been allocated for it, and deletes references to it from the *master* database.

The syntax of **drop database** 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 instead.

Creating Tables

When you create a table, you name its columns and supply a datatype for each column. You can also specify whether a particular column can hold null values or specify integrity constraints for columns in the table. There can be as many as 2 billion tables per database.

Example of Creating a Table

Use the *newpubs* database you created in the previous section if you want to try these examples. Otherwise, these changes will affect another database, like *pubs2* or *pubs3*.

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 described under "Identifiers" on page 1-7. 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. (The last column definition does not need a comma after it.)

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*..". For more information on temporary tables, see "Using Temporary Tables" on page 7-20.

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 create tables named *info*. Users who have permissions on both tables have to qualify them as *jonah.info* and *sally.info*. Sally has to qualify references to Jonah's table as *jonah.info*, but she can refer to her own table 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 does the following:

- It 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.

- It defines 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 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 an existing segment. ***segment_name*** 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 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 not applicable." 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 does not make an entry in a column defined with the keyword **null**, Adaptive Server supplies the value "NULL". A column defined with the keyword **null** also accepts 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, Adaptive 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, Adaptive Server uses the null mode defined for the database (by default, NOT NULL). For compatibility with 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, Adaptive Server insists on an entry. If there is no entry for a NOT NULL column, Adaptive Server displays an error message.

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 that the price had not been set 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.

Constraints and Rules Used with Null Values

It is not possible to define a column to allow null values, and then to override this definition with a constraint or a rule that prohibits null values. For example, if a column definition specifies NULL and a rule specifies this:

```
@val in (1,2,3)
```

an implicit or explicit NULL does not violate the rule. The column definition overrides the rule, even a rule that specifies:

```
@val is not null
```

See "Defining Integrity Constraints for Tables" on page 7-31 for more information on constraints. Rules are covered in detail in Chapter 12, "Defining Defaults and Rules for Data."

Defaults and Null Values

You can use defaults, that is, values that are supplied automatically when no entry is made, 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 Defaults and Rules for Data."

If you specify NOT NULL when you create a column and do not create a default for it, an error message occurs when a user fails to make an entry in that column during an insert. In addition, the user cannot **insert** or **update** such a column with NULL as a value.

The following table illustrates the interaction between a column's default and its null type when a user specifies no column value or explicitly enters a NULL value. The three possible results are a null value for the column, the default value for the column, or an error message.

Table 7-1: Column definition and null defaults

Column Definition	User Entry	Result
Null and default defined	Enters no value	Default used
	Enters NULL value	NULL used
Null defined, no default defined	Enters no value	NULL used
	Enters NULL value	NULL used
Not null, default defined	Enters no value	Default used
	Enters NULL value	NULL used
Not null, no default defined	Enters no value	Error
	Enters NULL value	Error

Nulls Require Variable Length Datatypes

Only columns with variable-length datatypes can store null values. When you create a NULL column with a fixed-length datatype, Adaptive Server converts it to the corresponding variable-length datatype. Adaptive Server does not inform you of the type change.

The following chart lists the fixed-length datatypes and the variable-length datatypes to which Adaptive Server converts them. Certain

variable-length datatypes, such as *moneyn*, are reserved types; you cannot use them to create columns, variables, or parameters.

Table 7-2: Conversion of fixed-length to variable-length datatypes

Original Fixed-Length Datatype	Converted To
char	*varchar*
nchar	*nvarchar*
binary	*varbinary*
datetime	*datetimn*
float	*floatn*
int, smallint, and tinyint	*intn*
decimal	*decimaln*
numeric	*numericn*
money and smallmoney	*moneyn*

Data entered into *char*, *nchar*, and *binary* columns follows the rules for variable-length columns, rather than being padded with spaces or zeros to the full length of the column specification.

text and *image* Columns

text and *image* columns created with **insert** and NULL are not initialized and contain no value. They do not use storage space and cannot be accessed with **readtext** or **writetext.**

When a NULL value is written in a *text* or *image* column with **update**, the column is initialized, a valid text pointer to the column is inserted into the table, and a 2K data page is allocated to the column. Once the column is initialized, it can be accessed by **readtext** and **writetext**. See the *Adaptive Server Reference Manual* for more information.

Using IDENTITY Columns

An IDENTITY column contains a value for each row, generated automatically by Adaptive Server, that uniquely identifies the row within the table.

Each table can have only one IDENTITY column. You can define an IDENTITY column when you create a table with a **create table** or **select into** statement, or add it later with an **alter table** statement. IDENTITY columns cannot be updated and do not allow nulls.

You define an IDENTITY column by specifying the keyword **identity**, instead of **null** or **not null**, in the **create table** statement. IDENTITY columns must have a datatype of *numeric* and scale of 0. Define the IDENTITY column with any desired precision, from 1 to 38 digits, in a new table:

```
create table table_name
    (column_name numeric(precision,0) identity)
```

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 9999:

```
create table sales_daily
    (sale_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.

➤ ***Note***

By default, Adaptive Server begins numbering rows with the value 1, and continues numbering rows consecutively as they are added. Some activities, such as manual insertions, deletions, or transaction rollbacks, and server shutdowns or failures, can create gaps in IDENTITY column values. Adaptive Server provides several methods of controlling identity gaps. These are described in “Managing Identity Gaps in Tables” on page 7-24.

Creating IDENTITY Columns with User-Defined Datatypes

You can 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.

The following example shows a user-defined datatype with the IDENTITY property:

```
sp_addtype ident, "numeric(5)", "identity"
```

The following example shows an IDENTITY column based on the *ident* datatype:

```
create table sales_monthly
    (sale_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.

Referencing IDENTITY Columns

When you create a table column that references an IDENTITY column, as with any referenced column, make sure it has the same datatype definition as the IDENTITY column. For example, in the *pubs3* database, the *sales* table is defined using the *ord_num* column as an IDENTITY column:

```
create table sales
(stor_id char(4) not null
     references stores(stor_id),
ord_num numeric(6,0) identity,
date datetime not null,
unique nonclustered (ord_num))
```

Notice that the *ord_num* IDENTITY column is defined as a unique constraint, which it needs for referencing the *ord_num* column in *salesdetail*. *salesdetail* is defined as follows:

```
create table salesdetail
(stor_id char(4) not null
     references storesz(stor_id),
ord_num numeric(6,0)
     references salesz(ord_num),
title_id tid not null
     references titles(title_id),
qty smallint not null,
discount float not null)
```

An easy way to insert a row into *salesdetail* after inserting a row into *sales* is to use the *@@identity* global variable to insert the IDENTITY column value into *salesdetail*. The *@@identity* global variable stores the most recently generated IDENTITY column value. For example:

```
begin tran
insert sales values ("6380", "04/25/97")
insert salesdetail values ("6380", @@identity, "TC3218", 50, 50)
commit tran
```

This example is in a transaction because both inserts depend on each other in order to succeed. For example, if the *sales* insert fails, the value of *@@identity* will be different, resulting in an erroneous row being inserted into *salesdetail*. Because the two inserts are in a transaction, if one fails, the entire transaction is rejected.

For more information on IDENTITY columns, see "Retrieving IDENTITY Column Values with @@identity" on page 8-18. For information on transactions, see Chapter 18, "Transactions: Maintaining Data Consistency and Recovery."

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 a **select**, **insert**, **update**, or **delete** statement on the table.

For example, the following query selects the row for which *sale_id* equals 1:

```
select * from sales_monthly
    where syb_identity = 1
```

Creating "Hidden" IDENTITY Columns Automatically

System Administrators can use the **auto identity** database option to automatically include a 10-digit IDENTITY column in new tables. To turn this feature on in a database, use:

```
sp_dboption database_name, "auto identity", "true"
```

Each time a user creates a new table without specifying either a **primary** key, a **unique** constraint, or an IDENTITY column, Adaptive Server automatically defines an IDENTITY column. The IDENTITY column is not visible when you use **select *** to retrieve all columns from the table. You must explicitly include the column name, *SYB_IDENTITY_COL* (all uppercase letters), in the select list. If Component Integration Services is enabled, the automatic IDENTITY column for proxy tables is called *OMNI_IDENTITY_COL*.

To set the precision of the automatic IDENTITY column, use the **size of auto identity** configuration parameter. For example, the following command sets the precision of the IDENTITY column to 15:

```
sp_configure "size of auto identity", 15
```

Using Temporary Tables

Temporary tables are created in the *tempdb* database, not the current database. To create a temporary table, you must have **create table** permission in *tempdb*. **create table** permission defaults to the Database Owner.

To make a table temporary, use the pound sign (#) or "*tempdb*.." before the table name in the **create table** statement.

There are two kinds of temporary tables:

- Tables that can be shared among Adaptive Server sessions

  You create a shareable temporary table by specifying *tempdb* as part of the table name in the **create table** statement. For example, the following statement creates a temporary table that can be shared among Adaptive Server sessions:

  ```
  create table tempdb..authors
  (au_id char(11))
  ```

  Adaptive Server does not change the names of temporary tables created this way. The table exists until the current session ends or until its owner drops it using **drop table**.

- Tables that are accessible only by the current Adaptive Server session or procedure

  You create a non-shareable temporary table by specifying a pound sign (#) before the table name in the **create table** statement. For example:

  ```
  create table #authors
  (au_id char (11))
  ```

  The table exists until the current session or procedure 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 Adaptive Server so that *model* is copied to *tempdb*.

Ensuring That the Temporary Table Name Is Unique

To ensure that a temporary table name is unique for the current session, Adaptive Server:

- Truncates the table name to 13 characters, including the pound sign (#)
- Pads shorter names to 13 characters, using underscores (_)
- Appends a 17-digit numeric suffix that is unique for an Adaptive Server session

The following example shows a table created as *#temptable* and stored as *#temptable___00000050010721973*:

```
use pubs2
go
create table #temptable (task char(30))
go
use tempdb
go
select name from sysobjects where name like
    "#temptable%"
go
```

```
name
------------------------------
#temptable___00000050010721973

(1 row affected)
```

Manipulating Temporary Tables in Stored Procedures

Stored procedures can reference temporary tables that are created during the current session. Within a stored procedure, you cannot create a temporary table, drop it, and then create a new temporary table with the same name.

Temporary Tables with Names Beginning with "#"

Temporary tables with names beginning with "#" that are created within stored procedures disappear when the procedure exits. A single procedure can:

- Create a temporary table
- Insert data into the table
- Run queries on the table
- Call other procedures that reference the table

Since the temporary table must exist in order to create procedures that reference it, here are the steps to follow:

1. Use a **create table** statement to create the temporary table.
2. Create the procedures that access the temporary table, but do not create the procedure that creates the table.
3. Drop the temporary table.
4. Create the procedure that creates the table and calls the procedures created in step 2.

Tables with Names Beginning with tempdb..

You can 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 steps above to create these tables.

◆ ***WARNING!***

Create temporary tables with the "tempdb.." prefix from inside a stored procedure *only* if you intend to share the table among users and sessions. Stored procedures that create and drop a temporary table should use the # prefix to avoid inadvertent sharing.

General Rules on Temporary Tables

Temporary tables with names that begin with # are subject to the following restrictions:

- You cannot create views on these tables.
- You cannot associate triggers with these tables.
- You cannot tell which session or procedure has created these tables.

These restrictions do not apply to shareable, temporary tables created in *tempdb*.

Rules that apply to both types of temporary tables are as follows:

- You can associate rules, defaults and indexes with temporary tables. Indexes created on a temporary table disappear when the temporary table disappears.
- System procedures such as **sp_help** work on temporary tables only if you invoke them from *tempdb*.
- User-defined datatypes cannot be used in temporary tables unless the datatypes exist in *tempdb*; that is, unless they have been explicitly created in *tempdb* since the last time Adaptive Server was restarted.
- You do not have to set the **select into/bulkcopy** option **on** to **select into** a temporary table.

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* or *pubs3* and there is another database called *newpubs*, you can create a table called *newtab* in *newpubs* like this:

```
create table newpubs..newtab (col1 int)
```

You cannot create other database objects—views, rules, defaults, stored procedures, and triggers—in a database other than the current one.

Managing Identity Gaps in Tables

The *identity* column contains a unique ID number, generated by Adaptive Server, for each row in a table. Because of the way the server generates ID numbers by default, you may occasionally have large gaps in the ID numbers. The **identity_gap** parameter gives you control over ID numbers, and potential gaps in them, for a specific table.

Why Are There Gaps in Identity Numbers?

By default, Adaptive Server allocates a block of ID numbers in memory instead of writing each ID number to disk as it is needed, which requires more processing time. The server writes the highest number of each block to the table's OAM page. This number is used as the starting point for the next block after the currently allocated block of numbers is used or "burned". The other numbers of the block are held in memory, but are not saved to disk. Numbers are considered burned when they are allocated to memory, then deleted from memory either because they were assigned to a row or because they were erased from memory due to some abnormal occurrence such as a system failure.

Allocating a block of ID numbers increases performance because it reduces contention for the table. However, if the server fails or is shut down with **no wait** before all the ID numbers are assigned, the unused numbers are burned. When the server is running again, it starts numbering with the next block of numbers based on the highest number of the previous block that the server wrote to disk. Depending on how many allocated numbers were assigned to rows before the failure, you may have a large gap in the ID numbers.

Parameters for Controlling Identity Gaps

Adaptive Server provides parameters that allow you to control gaps in identity numbers as described in Table 7-3.

Table 7-3: Parameters for controlling identity gaps

Parameter Name	Scope	Used With	Description
identity_gap	table-specific	**create table** or **select into**	Creates ID number blocks of a specific size for a specific table. Overrides **identity burning set factor** for the table. Works with **identity grab size**.
identity burning set factor	server-wide	**sp_configure**	Indicates a percentage of the total available ID numbers you want to have allocated for each block. Works with **identity grab size**. If the **identity_gap** for a table is set to 1 or higher, **identity burning set factor** has no effect on that table. The burning set factor is used for all tables for which **identity_gap** is set to 0. When you set the configuration parameter **identity burning set factor**, you express the number in decimal form, and then multiply it by 10,000,000 (10 raised to the power 7) to get the correct value to use with **sp_configure**. For example, to release 15 percent (.15) of the potential IDENTITY column values at one time, you specify a value of .15 times 10^7 (or 1,500,000): **sp_configure "identity burning set factor", 1500000**
identity grab size	server-wide	**sp_configure**	Reserves a block of contiguous ID numbers for each process. Works with **identity burning set factor** and **identity_gap**.

Comparison of *identity burning set factor* and *identity_gap*

The **identity_gap** parameter gives you control over the size of identity gaps for a particular table as illustrated in the following examples. In the examples, we have created a table named *books* to list all the books in our bookstore. We want each book to have a unique ID number, and we want Adaptive Server to automatically generate the ID numbers.

Example of Using *identity burning set factor*

When defining the *identity* column for the *books* table, we used the default numeric value of (18, 0), which provides a total of 999,999,999,999,999,999 ID numbers. For the **identity burning set factor** configuration parameter, we are using the default setting of 5000 (.05 percent of 999,999,999,999,999,999), which means that Adaptive Server allocates blocks of 500,000,000,000,000 numbers.

The server allocates the first 500,000,000,000,000 numbers in memory and stores the highest number of the block (500,000,000,000,000) on the table's OAM page. When all the numbers are assigned to rows or burned, the server takes the next block of numbers (the next 500,000,000,000,000), starting with 500,000,000,000,001, and stores the number 1,000,000,000,000,000 as the highest number of the block.

Let's say, after row number 500,000,000,000,022 the server fails. Only numbers 1 through 500,000,000,000,022 were used as ID numbers for *books*. Numbers 500,000,000,000,023 through 1,000,000,000,000,000 are burned. When the server is running again, it creates ID numbers starting from the highest number stored on the table's OAM page plus one (1,000,000,000,000,001),which leaves a gap of 499,999,999,999,978 ID numbers.

To reduce this large gap in identity numbers for a specific table, you can set the identity gap as described in "Example of Using identity_gap."

Example of Using *identity_gap*

In this case, we create the *books* table with an **identity_gap** value of 1000. This overrides the server-wide **table burning set factor** setting that resulted in blocks of 500,000,000,000,000 ID numbers. Instead, ID numbers are allocated in memory in blocks of 1000.

The server allocates the first 1000 numbers and stores the highest number of the block (1000) to disk. When all the numbers are used, the server takes the next 1000 numbers, starting with 1001, and stores the number 2000 as the highest number.

Now, let's say that after row number 1002 we lose power, which causes the server to fail. Only the numbers 1000 through 1002 were used, and numbers 1003 through 2000 are lost. When the server is running again, it creates ID numbers starting from the highest number stored on the table's OAM page plus one (2000), which leaves a gap of only 998 numbers.

You can significantly reduce the gap in ID numbers by setting the **identity_gap** for a table instead of using the server-wide **table burning set factor**. However, there may be a performance cost to setting this value too low. Each time the server must write the highest number of a block to disk, performance is affected. For example, if **identity_gap** is set to 1, which means you are allocating one ID number at a time, the server must write the new number every time a row is created, which may reduce performance because of page lock contention on the table. You must find the best setting to achieve the optimal performance with the lowest gap value acceptable for your situation.

Setting the Table-Specific Identity Gap

You set the table-specific identity gap when you create a table. You can do this as part of either **create table** or **select into**.

Setting Identity Gap with *create table*

The syntax is:

```
create table table_name (column_name
   datatype(constant_expression) identity)
   with identity_gap = value
```

For example:

```
create table mytable (IdNum numeric(12,0) identity)
with identity_gap = 10
```

This statement creates a table named *mytable* with an *identity* column. The identity gap is set to 10, which means ID numbers will be allocated in memory in blocks of ten. If the server fails or is shut down with no wait, the maximum gap between the last ID number assigned to a row and the next ID number assigned to a row is ten numbers.

Setting Identity Gap with *select into*

If you are creating a table in a **select into** statement from a table that has a specific identity gap setting, the new table does not inherit the identity gap setting from the parent table. Instead, the new table uses the **identity burning set factor** setting. To give the new table a specific **identity_gap** setting, specify the identity gap in the **select into** statement. You can give the new table an identity gap that is the same as or different from the parent table.

For example, to create a new table (*newtable*) from the existing table (*mytable*) with an identity gap, you specify it in the **select into** statement:

```
select identity into newtable
with identity_gap = 20
from mytable
```

Changing the Table-Specific Identity Gap

To change the identity gap for a specific table, use **sp_chgattribute**.

The syntax is:

```
sp_chgattribute "table_name", "identity_gap",
   set_number
```

where:

- *table_name* is the name of the table for which you want to change the identity gap
- **identity_gap** indicates that you want to change the identity gap.
- *set_number* is the new size of the identity gap.

For example:

```
sp_chgattribute "mytable", "identity_gap", 20
```

To change *mytable* to use the **identity burning set factor** setting instead of the **identity_gap** setting, set **identity_gap** to **0**:

```
sp_chgattribute "mytable", "identity_gap", 0
```

Displaying Table-Specific Identity Gap Information

To see the **identity_gap** setting for a table, use **sp_help**.

For example, the zero value in the *identity_gap* column (towards the end of the output) indicates that no table-specific identity gap is set. *mytable* uses the server-wide **identity burning set factor** value.

```
sp_help mytable
Name                            Owner
       Type
------------------------------ ------------------------------
      ----------------------
mytable                         dbo
       user table

(1 row affected)
Data_located_on_segment         When_created
------------------------------ --------------------------
default                                 Jun  8 1999  5:35PM
Column_name     Type             Length Prec Scale Nulls Default_name
       Rule_name       Identity
-------------- -------------- ------ ---- ----- ----- --------------
      -------------- --------
idnum           numeric               6   12     0     0 NULL
       NULL                    1
Object does not have any indexes.
No defined keys for this object.
Object is not partitioned.
Lock scheme Allpages
The attribute 'exp_row_size' is not applicable to tables with
allpages lock scheme.
The attribute 'concurrency_opt_threshold' is not applicable to
tables with allpages lock scheme.

exp_row_size reservepagegap fillfactor max_rows_per_page identity_gap
 ------------ -------------- ---------- ----------------- ------------

            1              0          0                 0            0
concurrency_opt_threshold
-------------------------
0
```

If you change the **identity_gap** of *mytable* to 20, the **sp_help** output for the table shows 20 in the *identity_gap* column. This setting overrides the server-wide **identity burning set factor** value.

```
sp_help mytable
Name                            Owner
       Type
------------------------------ ------------------------------
      ----------------------
mytable                         dbo
       user table

(1 row affected)
Data_located_on_segment         When_created
------------------------------ --------------------------
```

```
default                                 Jun  8 1999  5:35PM
Column_name     Type            Length Prec Scale Nulls Default_name
       Rule_name       Identity
--------------- --------------- ------ ---- ----- ----- ---------------
      --------------- --------
idnum           numeric              6   12     0     0 NULL
       NULL                  1
Object does not have any indexes.
No defined keys for this object.
Object is not partitioned.
Lock scheme Allpages
The attribute 'exp_row_size' is not applicable to tables with
allpages lock scheme.
The attribute 'concurrency_opt_threshold' is not applicable to
tables with allpages lock scheme.

exp_row_size reservepagegap fillfactor max_rows_per_page identity_gap
------------ -------------- ---------- ----------------- ------------
1                         0          0                 0           20
concurrency_opt_threshold
-------------------------
0
```

Gaps Due to Insertions, Deletions, Identity Grab Size, and Rollbacks

Manual insertions into the IDENTITY column, deletion of rows, the value of the **identity grab size** configuration parameter, and transaction rollbacks can create gaps in IDENTITY column values. These gaps are not affected by the setting of the **identity burning set factor** configuration parameter.

For example, assume that you have an IDENTITY column with the following values:

```
select syb_identity from stores_cal

 id_col
 -------
      1
      2
      3
      4
      5

(5 rows affected)
```

You can delete all rows for which the IDENTITY column falls between 2 and 4, leaving gaps in the column values:

```
delete stores_cal
where syb_identity between 2 and 4
```

```
select syb_identity from stores_cal
 id_col
 ------
      1
      5

(2 rows affected)
```

After setting **identity_insert on** for the table, the table owner, Database Owner, or System Administrator can manually insert any legal value greater than 5. For example, inserting a value of 55 would create a large gap in IDENTITY column values:

```
insert stores_cal
(syb_identity, stor_id, stor_name)
values (55, "5025", "Good Reads")

select syb_identity from stores_cal

id_col
 -------
      1
      5
     55

(3 rows affected)
```

If **identity_insert** is then set to **off**, Adaptive Server assigns an IDENTITY column value of 55 + 1, or 56, for the next insertion. If the transaction that contains the **insert** statement is rolled back, Adaptive Server discards the value 56 and uses a value of 57 for the next insertion.

If Table Inserts Reach the IDENTITY Column's Maximum Value

The maximum number of rows you can insert into a table depends on the precision set for the IDENTITY column. If a table reaches that limit, you can either re-create the table with a larger precision or, if the table's IDENTITY column is not used for referential integrity, use the **bcp** utility to remove the gaps. See "Reaching the IDENTITY Column's Maximum Value" on page 8-20 for more information.

Defining Integrity Constraints for Tables

Transact-SQL provides two methods for maintaining data integrity in a database:

- Defining rules, defaults, indexes, and triggers

- Defining **create table** integrity constraints

Specifying the Appropriate Method for Data Integrity

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 not 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. Use the system procedure **sp_helpconstraint** to find a table's referenced tables.
- **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 "Allowing Null Values" on page 7-13 for information about the **null** and **not null** keywords.

For information about any constraints defined for a table, see "Using sp_helpconstraint to Find a Table's Constraint Information" on page 7-80.

◆ ***WARNING!***

Do not define constraints for system tables or alter their definition.

Specifying Table-Level or Column-Level Constraints

You can declare integrity constraints at the table or column level. The difference is syntactic. You place column-level constraints after the column name and datatype, but before the delimiting comma. You enter table-level constraints as separate comma-delimited clauses. Adaptive Server treats table-level and column-level constraints the same way; both ways are equally efficient.

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 under "Specifying Check Constraints" on page 7-41.

Creating Error Messages for Constraints

You can create error messages and bind them to constraints by creating messages with **sp_addmessage** and binding them to constraints with **sp_bindmsg**.

For example:

```
sp_addmessage 25001,
     "The publisher ID must be 1389, 0736, or 0877"

sp_bindmsg my_chk_constraint, 25001

insert my_publishers values
     ("0000", "Reject This Publisher")

Msg 25001, Level 16, State 1:
Server 'snipe', Line 1:
The publisher ID must be 1389, 0736, or 0877
Command has been aborted.
```

To change the message for a constraint, just bind a new message. The new message replaces the old message.

Unbind messages from constraints with **sp_unbindmsg** and drop user-defined messages with **sp_dropmessage**.

For example:

```
sp_unbindmsg my_chk_constraint

sp_dropmessage 25001
```

To change the text of a message but keep the same error number, unbind it, drop it with **sp_dropmessage**, add it again with **sp_addmessage**, and bind it with **sp_bindmsg**.

After Creating a Check Constraint

After you create a check constraint, the **source text** describing the check constraint is stored in the *text* column of the *syscomments* system table. In previous releases of SQL Server, users often deleted the source text from *syscomments*, in order to save disk space and remove confidential information from this public area. **Do not remove this information from *syscomments*;** doing so can cause problems for future upgrades of Adaptive Server. Instead, you can encrypt the text in *syscomments* by using the **sp_hidetext** system procedure, described in the *Adaptive Server Reference Manual*. For more information, see "Compiled Objects" on page 1-3.

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, Adaptive Server inserts the default value.

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 Adaptive 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 Adaptive 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 in 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, Adaptive 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, the following **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.”

Dropping A Key

To drop a key that you defined with **sp_primarykey**, **sp_foreignkey** or **sp_commonkey**, use **sp_dropkey**. Its syntax is:

```
sp_dropkey keytype, tabname [, deptabname]
```

How you specify *keytype* depends on how it is defined:

- If *keytype* is **foreign** or **common**, *deptabname* specifies the name of the second table in the relationship. (*tabname* specifies the first table in the relationship.)
- If *keytype* is **primary**, this parameter is not needed, since **primary** keys have no dependent tables.
- If *keytype* is **foreign**, this is the name of the primary key table.
- If *keytype* is **common**, give the two table names in the order in which they appear with **sp_helpkey**.

Specifying Referential Integrity Constraints

Referential integrity refers to the methods used to manage the relationships between tables. When you create a table, you can define constraints to ensure that the data inserted into a particular column has matching values in another table.

There are three types of references you can define in a table: references to another table, references from another table, and self-references, that is, references within the same table.

The following two tables from the *pubs3* database illustrate how declarative referential integrity works. The first table, *stores*, is a "referenced" table:

```
create table stores
(stor_id     char(4) notnull,
stor_name    varchar(40) null,
stor_address varchar(40) null,
city         varchar(20) null,
state        char(2) null,
country      varchar(12) null,
postalcode   char(10) null,
payterms     varchar(12) null,
unique nonclustered (stor_id))
```

The second table, *store_employees*, is a "referencing table" because it contains a reference to the *stores* table. It also contains a self-reference:

```
create table store_employees
(stor_id     char(4) null
       references stores(stor_id),
emp_id       id not null,
mgr_id       id null
       references store_employees(emp_id),
emp_lname    varchar(40) not null,
emp_fname    varchar(20) not null,
phone        char(12) null,
address      varchar(40) null,
city         varchar(20) null,
state        char(2) null,
country      varchar(12) null,
postalcode   varchar(10) null,
unique nonclustered (emp_id))
```

The references defined in the *store_employees* table enforce the following restrictions:

- Any store specified in the *store_employees* table must be included in the *stores* table. The **references** constraint enforces this by saying that any value inserted into the *stor_id* column in *store_employees* must already exist in the *stor_id* column in *my_stores*.
- All managers must have employee identification numbers. The **references** constraint enforces this by saying that any value inserted into the *mgr_id* column must already exist in the *emp_id* column.

Table-Level or Column-Level Referential Integrity Constraints

You can define referential integrity constraints at the column level or the table level. The referential integrity constraints in the preceding examples were defined at the column level, using the **references** keyword in the **create table** statement.

When you define table-level referential integrity constraints, include the **foreign key** clause and a list of one or more column names. The **foreign key** clause specifies that the listed columns in the current table are foreign keys whose target keys are the columns listed the following **references** clause. For example:

```
constraint sales_detail_constr
     foreign key (stor_id, ord_num)
     references my_salesdetail(stor_id, ord_num)
```

The **foreign key** syntax is permitted only for table-level constraints, and not for column-level constraints. For more information, see "Specifying Table-Level or Column-Level Constraints" on page 7-33.

After defining referential integrity constraints at the column level or the table level, you can use the **sp_primarykey**, **sp_foreignkey** and **sp_commonkey** system procedures to define the keys in the *syskeys* system table.

Maximum Number of References Allowed for a Table

The maximum number of references allowed for a table is 192. You can check a table's references by using the system procedure **sp_helpconstraint**, described under "Using sp_helpconstraint to Find a Table's Constraint Information" on page 7-80.

Using *create schema* for Cross-Referencing Constraints

If you attempt to create a table that references a table that does not yet exist, Adaptive Server does not create the table, but returns an error message saying that the referenced table does not exist. In order to create two or more tables that reference each other, use the **create schema** command.

A **schema** is a collection of objects owned by a particular user, and the permissions associated with those objects. If any of the statements within a **create schema** statement fail, the entire command is rolled back as a unit, and none of the commands take effect.

The **create schema** syntax is:

```
create schema authorization authorization_name
   create_oject_statement
       [ create_object_statement ... ]
   [ permission_statement ... ]
```

For example:

```
   create schema authorization dbo
       create table list1
           (col_a char(10) primary key,
           col_b char(10) null
           references list2(col_A))

       create table list2
           (col_A char(10) primary key,
           col_B char(10) null
           references list1(col_a))
```

General Rules for Creating Referential Integrity Constraints

When you define referential integrity constraints in a table, follow these rules:

- Make sure you have **references** permission on the referenced table to use referential integrity constraints. For information about permissions, see the *Security Features User's Guide*.
- Make sure that the referenced columns are constrained by a unique index in the referenced table. You can create that unique index using either the **unique** or **primary key** constraint or the **create index** statement. For example, the referenced column in the *stores* table was defined as follows:

  ```
  stor_id char(4) primary key
  ```

- Make sure the columns used in the references definition have matching datatypes. For example, the *stor_id* columns in both *my_stores* and *store_employees* were created using the *char(4)* datatype. The *mgr_id* and *emp_id* columns in *store_employees* were created with the *id* datatype.
- You can omit column names in the **references** clause only if the columns in the referenced table are designated as a primary key through a **primary key** constraint.
- You cannot delete rows or update column values from a referenced table that match values in a referencing table. Delete or update from the referencing table first, and then delete it from the referenced table.

  Similarly, you cannot use **truncate table** on a referenced table. Truncate the referencing table first, and then truncate the referenced table.
- You must drop the referencing table before you drop the referenced table; otherwise, a constraint violation occurs.
- Use the system procedure **sp_helpconstraint** to find a table's referenced tables.

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 more information, see Chapter 16, "Triggers: Enforcing Referential Integrity."

Specifying Check Constraints

You can declare a **check** constraint to limit the values users insert into a column in a table. Check constraints are useful for applications that check a limited, specific range of values. A **check** constraint specifies a *search_condition* that 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))
```

Column-level check constraints can reference only the column on which the constraint is defined; they 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.

Because check constraints do not override the column definitions, you cannot use a check constraint to prohibit null values if the column definition permits them. If you declare a check constraint on a column that allows null values, you can insert NULL into the column, implicitly or explicitly, even though NULL is not included in the *search_condition*. For example, suppose you define the following check constraint on a table column that allows null values:

```
check (pub_id in ("1389", "0736", "0877", "1622",
"1756"))
```

You can still insert NULL into that column. The column definition overrides the check constraint because the following expression always evaluates to true:

```
col_name != null
```

Tips on Designing Applications That Use Referential Integrity

Following are performance tips to be aware of when you design applications that use referential integrity features:

- Do not create unnecessary referential constraints. The more referential constraints a table has, the slower a statement requiring referential integrity runs on that table.
- Use as few self-referencing constraints on a table as possible.
- Use the **check** constraint rather than the **references** constraint for applications that check a limited, specific range of values. Using the **check** constraint eliminates the need for Adaptive Server to scan other tables to complete the query, since there are no references. Therefore, queries on such tables run faster than on tables using references.

  For example, the following table uses a **check** constraint to limit the authors to California:

  ```
  create table cal_authors
  (au_id id not null,
  au_lname varchar(40) not null,
  au_fname varchar(20) not null,
  phone char(12) null,
  address varchar(40) null,
  city varchar(20) null,
  state char(2) null
       check(state = "CA"),
  country varchar(12) null,
  postalcode char(10) null)
  ```

- Bind commonly scanned foreign key indexes to their own caches, to optimize performance. Unique indexes are created automatically on columns declared as primary keys. These indexes are usually selected to scan the referenced table when their corresponding foreign keys are updated or inserted.
- Keep multi-row updates of candidate keys at a minimum.
- Put referential integrity queries into procedures that use constraint checks. Constraint checks are compiled into the execution plan; when a referential constraint is altered, the procedure that has the constraint compiled is automatically recompiled when that procedure is executed.

- If you cannot embed referential integrity queries in a procedure and you have to run referential integrity queries frequently in an ad hoc batch, bind the system catalog *sysreferences* to its own cache. Doing so improves performance when Adaptive Server needs to recompile referential integrity queries.
- After you create a table that has referential constraints, test it by using **set showplan, noexec on** before running a query using the table. The **showplan** output indicates the number of auxiliary scan descriptors required to run the query; scan descriptors manage the scan of a table whenever queries are run on it. If the number of auxiliary scan descriptors is very high, redesign the table so that it uses fewer scan descriptors, or increase the value of the **number of auxiliary scan descriptors** configuration parameter.

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.

First, plan the table's design:

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 in which they are to be used.
3. Decide which column, if any, should be the IDENTITY column.
4. Decide which columns should and which should not accept null values.
5. Decide what integrity constraints or column defaults, if any, you need to add to the columns in the table. This includes deciding when to use column constraints and defaults instead of defaults, rules, indexes, and triggers to enforce data integrity.
6. Decide whether 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, "Creating Indexes on Tables."

Now, create the table and its associated objects:

1. Create the table and its indexes with the **create table** and **create index** commands.
2. Create the defaults and rules you need with the **create default** and **create rule** commands. These commands are discussed in Chapter 12, "Defining Defaults and Rules for Data."
3. 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, "Using Stored Procedures."
4. Create triggers with the **create trigger** command. Triggers are discussed in Chapter 16, "Triggers: Enforcing Referential Integrity."
5. Create views with the **create view** command. Views are discussed in Chapter 9, "Views: Limiting Access to Data."

Make a Design Sketch

The table called *friends_etc* is used in this chapter and 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 another user has already created the *friends_etc* table, check with a System Administrator or the Database Owner if you plan to follow the examples and create the objects that go with *friends_etc*. The owner of *friends_etc* will need to drop its indexes, defaults, rules, and triggers so that there will be no conflict when you create these objects.

Table 7-4 shows the proposed structure of the *friends_etc* table and the indexes, defaults, and rules that go with each column.

Table 7-4: Sample table design

Column	Datatype	Null?	Index	Default	Rule
pname	*nm*	NOT NULL	*nmind*(composite)		
sname	*nm*	NOT NULL	*nmind*(composite)		

Table 7-4: 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*	
gender	*bit*	NOT NULL		*gndrdflt*	
debt	*money*	NOT NULL		*gndrdflt*	
notes	*varchar(255)*	NULL			

Create the User-Defined Datatypes

The first two columns are for the personal (first) 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 an entry is required for that column, for example, for the two name columns in this table. The other data is meaningless without the names. In addition, the *gender*

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 Chapter 12, "Defining Defaults and Rules for Data."

Define the Table

Now, write the **create table** statement:

```
create table friends_etc
(pname       nm             not null,
sname        nm             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,
gender       bit            not null,
debt         money          not null,
notes        varchar(255)   null)
```

You have now defined columns for the personal name and surname, address, city, state, postal code, telephone number, age, birthday, gender, debt information, and notes. Later, you will create the rules, defaults, indexes, triggers, and views for this table.

Creating New Tables from Query Results: *select into*

The **select into** command lets you create a new table based on the columns specified in the **select** statement's select list and the rows chosen in the **where** clause. 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 use **select into** on a permanent table only if the **select into/bulkcopy/pllsort** database option is set to **on**. A System Administrator can turn on this option using the **sp_dboption** system procedure. Use the **sp_helpdb** system procedure to see if this option is on.

Here is what the **sp_helpdb** system procedure and its results look like when the **select into/bulkcopy/pllsort** database option is set to **on**:

```
   sp_helpdb pubs2

name       db_size owner   dbid    created      status
---------  ------- ------  -----   -----------  ------------
pubs       2 MB    sa      5       Jun  5 1997  select into
                                                /bulkcopy/pllsort

(1 row affected)

device              size         usage
-----------------   ---------    --------------
master              2 MB         data and log

(1 row affected)
```

The report from **sp_helpdb** indicates whether the option is set to **on** or **off**. Only the System Administrator or the Database Owner can set the database options.

If the **select into/bulkcopy/pllsort** 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 **select into/bulkcopy/pllsort** option is not **on**.

➤ ***Note***

Because **select into** is a minimally logged operation, use **dump database** to back up your database following a **select into**. You cannot dump the transaction log following a minimally logged operation.

Unlike a view that displays a portion of a table, a table created with **select into** is a separate, independent entity. See Chapter 9, "Views: Limiting Access to Data," for more information.

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 statement includes no **where**

clause, data from all 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)
```

Adaptive 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.

select into lets you 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)
```

Always supply a name for any column in the **select into** result table that results from an aggregate function or any other expression. Examples are:

- Arithmetic aggregates, for example, ***amount* * 2**
- Concatenation, for example, **lname + fname**
- Functions, for example, **lower(lname)**

Following is an example of 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)
```

Because functions allow null values, any column in the table that results from a function other than **convert**() or **isnull**() will allow null values.

Checking for Errors

select into is a two-step operation. The first step creates the new table and the second step inserts the specified rows into the table.

Because **select into** operations are not logged, they cannot be issued within user-defined transactions and cannot be rolled back.

If a **select into** statement fails after creating a new table, Adaptive Server does **not** automatically drop the table or deallocate its first data page. This means that any rows inserted on the first page before the error occurred remain on the page. Check the value of the *@@error* global variable after a **select into** statement to be sure that no error occurred.

If an error occurs from a **select into** operation, use the **drop table** statement to remove the new table, and then reissue the **select into** statement.

Using *select into* with IDENTITY Columns

The following section describes special rules for using the **select into** command with tables containing IDENTITY columns.

Selecting an IDENTITY Column into a New Table

To select an existing IDENTITY column into a new table, include the column name (or the **syb_identity** keyword) in the **select** statement's *column_list*:

```
select column_list
   into table_name
   from table_name
```

The following example creates a new table, *stores_cal_pay30*, based on columns from the *stores_cal* table:

```
select record_id, stor_id, stor_name
into stores_cal_pay30
from stores_cal
where payterms = "Net 30"
```

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.

Selecting the IDENTITY Column More Than Once

A table cannot have more than one IDENTITY column. If an IDENTITY column is selected more than once, it is defined as NOT NULL in the new table. It does not inherit the IDENTITY property.

In the following example, the *record_id* column, which is selected once by name and once by the **syb_identity** keyword, is defined as NOT NULL in *stores_cal_pay60:*

```
select syb_identity, record_id, stor_id, stor_name
into stores_cal_pay60
from stores_cal
where payterms = "Net 60"
```

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. 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
```

The following example creates a new table, *new_discounts*, from the *discounts* table and adds a new IDENTITY column, *id_col*:

```
select *, id_col=identity(5)
into new_discounts
from discounts
```

No table can have more than one IDENTITY column. If the *column_list* includes an existing IDENTITY column, and you add a description of a new IDENTITY column, the **select into** statement fails.

Defining a Column Whose Value Must Be Computed

IDENTITY column values are generated by Adaptive Server. New columns that are based on IDENTITY columns, but whose values must be computed rather than generated, cannot inherit the IDENTITY property.

If a table's **select** statement includes an IDENTITY column as part of an expression, the resulting column value must be computed. The new column is created as NULL if any column in the expression allows a NULL value. Otherwise, it is NOT NULL.

In the following example, the *new_id* column, which is computed by adding 1000 to the value of *record_id*, is created NOT NULL:

```
select new_id = record_id + 1000, stor_name
into new_stores
from stores_cal
```

Column values are also computed if the **select** statement contains a **group by** clause or aggregate function. If the IDENTITY column is the argument of the aggregate function, the resulting column is created NULL. Otherwise, it is NOT NULL.

IDENTITY Columns Selected into Tables with Unions or Joins

The value of the IDENTITY column uniquely identifies each row in a table. However, if a table's **select** statement contains a union or join, individual rows can appear multiple times in the result set. An IDENTITY column that is selected into a table with a union or join does not retain the IDENTITY property. If the table contains the union of the IDENTITY column and a NULL column, the new column is defined as NULL. Otherwise, it is NOT NULL.

For more information about IDENTITY columns, see "Using IDENTITY Columns" on page 7-16 and "Updating IDENTITY Columns" on page 8-29. See also **select** in the .

Altering Existing Tables

Use the **alter table** command to change the structure of an existing table. You can:

- Add columns and constraints
- Change column default values
- Add either **null** or **non-null** columns
- Drop columns and constraints
- Change locking scheme
- Partition or unpartition tables
- Convert column datatypes

- Convert the **null** default value of existing columns
- Increase or decrease column length

alter table includes the following syntax for modifying tables:

```
alter table table_name
   [add column_name datatype [identity | null |
       not null] [, column_name datatype [identity
       |null | not null]]]
   [drop column_name [, column_name]
       [modify column_name {[data_type]
       [[null] | [not null]]}
       [, column_name datatype [null | not null]]]
```

Where *table_name* is the table you are altering, and *datatype* is the datatype of the altered column. You must have the **sa_role** or be the object owner to run **alter table.** See the *Adaptive Server Reference Manual* for the complete **alter table** syntax.

For example, by default, the *au_lname* column of the *authors* table uses a *varchar(50)* datatype. To alter the *au_lname* to use a *varchar(60)*, enter:

```
alter table authors
modify au_lname varchar(60)
```

Dropping, modifying, and adding non-null columns may perform a data copy, which has implications for required space and the locking scheme. See "Data Copying" on page 7-66.

The modified table's page chains inherits the table's current configuration options (for example, if **fillfactor** is set to 50 percent, the new pages have this same **fillfactor**).

➤ ***Note***

Adaptive Server does partial logging (of page allocations) **for alter table** operations. However, because **alter table** is performed as a transaction, you cannot dump the transaction log after running **alter table**; you must dump the database to ensure it is recoverable. If the server encounters any problems during the alter table operation, Adaptive Server rolls back the transaction.

alter table acquires an exclusive table lock while it is modifying the table schema. This lock is released as soon as the command is done.

alter table does not fire any triggers.

Objects Using select * Do Not List Changes To Table

If a database has any objects (stored procedures, triggers, and so on) that perform a **select *** on a table from which you have dropped a column, an error message lists the missing column. This occurs even if you create the objects using the **with recompile** option. For example, if you dropped the *postalcode* column from the *authors* table, any stored procedure that performed a **select *** on this table issues the following error message:

```
Msg 207, Level 16, State 4:
Procedure 'columns', Line 2:
Invalid column name 'postalcode'.
(return status = -6)
```

This message does not appear if you add a new column and then run an object containing a **select ***; in this case, the new column does not appear in the output.

You must drop and re-create any objects that reference a dropped column.

Using *alter table* on Remote Tables

You can use **alter table** to modify a remote table using Component Integration Services (CIS). Before you modify a remote table, make sure that CIS is running by entering:

```
sp_companion "enable cis"
```

If CIS is enabled, the ouput of this command displays "1." By default, CIS is enabled when you install Adaptive Server.

See the *System Administration Guide* for information about enabling configuration parameters. See the *Component Integration Services User's Guide* for information about using CIS.

Adding Columns

To add a column to an existing table, use the following syntax:

```
alter table table_name
   add column_name datatype
   [default {constant_expression} | user | null}]
   {[{identity | null | not null}] | [constraint
   constraint_name
   {constraint_clause}]
   [, column_name]
```

Where:

- *table_name* is the table to which you are adding a column.
- *column_name* is the column you are adding.
- *constant_expression* is the constant you want placed in the row if a user does not specify a value (this value must be in quotes, unless the character type is *int*, *tinyint*, or *smallint*, *numeric*, or *decimal*).
- *constraint_name* is the constraint you are adding to the table.
- *constraint_clause* is the full text of the constraint you are adding. You can add any number of columns using a single **alter table** statement.

For example, the following adds a non-null column named *author_type*, which includes the constant "primary_author" as the default value, and a null column named *au_publisher* to the *authors* table:

```
alter table authors
add author_type varchar(20)
default "primary_author" not null,
au_publisher varchar(40) null
```

Adding Columns Appends Column IDs

alter table adds a column to the table with a column id that is one greater than the current maximum column ID. For example, Table 7-5 lists the default column IDs of the *salesdetail* table:

Table 7-5: Column IDs of the *salesdetail* table

Column name	*stor_id*	*ord_num*	*title_id*	*qty*	*discount*
Col ID	1	2	3	4	5

This command appends the *store_name* column to the end of the *salesdetail* table with a column ID of 6:

```
alter table salesdetail
add store_name varchar(40)
default "unknown" not null
```

If you add another column, it will have a column ID of 7.

➤ ***Note***

Because a table's column IDs change as columns are added and dropped, your applications should never rely on them.

Adding Not Null Columns

You can add a NOT NULL column to a table. This means that a constant expression, and not a null value, is placed in the column when the column is added. This also ensures that, for all existing rows, the new column is populated with the specified constant expression when the table is created.

NOT NULL columns cause Adaptive Server to issue an error message if a user fails to enter a value for a column.

For example, the following adds the column *owner* to the *stores* table with a default value of "unknown:"

```
alter table stores
add owner_lname varchar(20)
default "unknown" not null
```

The default value can be a constant expression when adding NULL columns, but it can only be a constant value when adding a NOT NULL column (as in the example above).

Adding Constraints

To add a constraint to an existing column, use:

```
alter table table_name
   add constraint constraint_name {constraint_clause}
```

Where *table_name* is the name of the table to which you are adding a constraint, *constraint_name* is the constraint you are adding, and *constraint_clause* is the rule the constrain enforces.

For example, to add a constraint to the *titles* table that makes sure an advance does not exceed 10,000:

```
alter table titles
add constraint advance_chk
check (advance < 10000)
```

If a user attempts to insert a value greater than 10,000 into the *titles* table, Adaptive Server produces an error message similar to the following:

```
Msg 548, Level 16, State 1:
Line 1:Check constraint violation occurred,
dbname = 'pubs2',table name= 'titles',
constraint name = 'advance_chk'.
Command has been aborted.
```

Adding a constraint does not affect the existing data. Also, if you add a new column with a default value and specify a constraint on that column, the default value is not validated against the constraint.

For information about dropping a constraint, see "Dropping Constraints" on page 7-59.

Dropping Columns

Drop a column from an existing table using:

```
alter table table_name
   drop column_name [, column_name ]
```

where *table_name* is the table that contains the column you are dropping, and *column_name* is the column you are dropping. You can drop any number of columns using a single **alter table** statement. However, you cannot drop the last remaining column from a table (for example, if you drop four columns from a five-column table, you cannot then drop the remaining column).

For example, the following drops the *advance* and the *contract* columns from the *titles* table:

```
alter table titles
drop advance, contract
```

alter table rebuilds all indexes on the table when it drops a column.

Dropping Columns Renumbers the Column ID

alter table re-maps column IDs when you drop a column from a table. Columns with IDs above the number of the dropped column move up one column ID to fill the gap that the dropped column leaves behind. For example, the *titleauthor* table contains the following column names and column IDs:

Table 7-6: titleauthor column IDs

Column Name	au_id	title_id	au_ord	royaltyper
Column ID	1	2	3	4

If you drop the *au_ord* column from the table with the following command:

```
alter table titleauthor drop au_ord
```

titleauthor now has the following column names and column IDs:

Table 7-7: Column IDs after dropping au_ord

Column Name	au_id	title_id	royaltyper
Column ID	1	2	3

The *royaltyper* column has moved up and taken the column id of 3. The non-clustered index on both *title_id* and *royaltyper* are also rebuilt when *au_ord* is dropped. Also, all instances of column IDs in different system catalogs are renumbered.

You will not generally notice the renumbering of column IDs. If, however, you have stored procedures or applications that depend on column IDs, you must rewrite these so they access the correct column ID's.

➤ ***Note***

Because a table's column IDs change as columns are added and dropped, your applications should never rely on them.

Dropping Constraints

To drop a constraint, use:

```
alter table table_name
   drop constraint constraint_name
```

Where *table_name* and *constraint_name* have the same values as described in ''Adding Constraints,'' above.

For example, to drop the constraint created above:

```
alter table titles
drop constaint advance_chk
```

To find detailed information about a table's constraints, use **sp_helpconstraint**, described under "Using sp_helpconstraint to Find a Table's Constraint Information" on page 7-80.

Modifying Columns

To modify an existing column, use:

```
alter table table_name
   modify column_name datatype [null | not null]
   [null | not null]
   [, column_name ...]
```

Where *table_name* is the table that contains the column you are modifying, *column_name* is the column you are modifying, and *data_type* is the datatype to which you are modifying the column. You can modify any number of columns in a single **alter table** statement.

For example, the following changes the datatype of the type column in the titles table from *char*(12) to *varchar*(20) and makes it nullable:

```
alter table titles
modify type varchar(20) null
```

◆ ***WARNING!***

You may have objects (stored procedures, triggers, and so on) that depend on a column having a particular datatype. Before you modify a column, make sure that any objects that reference it will be able to run successfully after the modification. Use sp_depends to determine a table's dependent objects.

Which Datatypes Can I Convert?

You can only convert datatypes that are either implicitly or explicitly convertible to the new datatype, or if there is an explicit conversion function in Transact-SQL. See Table 10-1 on page 10-32 for a list of the supported datatype conversions. If you attempt an illegal datatype modification, Adaptive Server raises an error message and the operation is aborted. **alter table** does not require that the table is empty to modify the table. If **alter table** encounters an illegal conversion during the data copy phase, the **alter table** is aborted.

➤ ***Note***

You cannot convert an existing column datatype to the timestamp datatype, nor can you modify a column that uses the timestamp datatype to any other datatype.

If you issue the same **alter table** command more than once, Adaptive Server does not modify the column, but issues a message similar to the following:

```
Warning: ALTER TABLE operation did not affect column 'au_lname'.
Msg 13905, Level 16, State 1:
Server 'SYBASE1', Line 1:
Warning: no columns to drop, add or modify. ALTER TABLE
'authors' was aborted.
```

Modifying Tables May Prevent Successful Bulk Copy of Previous Dump

Modifying either the length or datatype of a column may prevent you from successfully using bulk copy to copy in older dumps of the table. The older table schema from the dump may not be compatible with the new table schema. Before you modify a column's length or datatype, make sure this will not prevent you from copying in a previous dump of the table.

Decreasing Column Length May Truncate Data

If you decrease the length of a column, you must make sure the reduced column length does not result in truncated data. For example, you could use **alter table** to reduce the length of the *title* column of the *titles* table from a *varchar(80)* to a *varchar(2)*, but the data is meaningless:

```
select title from titles

title
-----
Bu
Co
Co
Em
Fi
Is
Li
Ne
```

```
On
Pr
Se
Si
St
Su
Th
Th
Th
Yo
```

Adaptive Server issues error messages about truncating the column data only if the **set string_rtruncation** option is turned on. If you need to truncate character data, set the appropriate string-truncation option and modify the column to decrease its length.

Modifying datetime Columns

If you modify the column from a *char* datatype to *datetime* or *smalldatetime*, you cannot specify the order that the month, day, and year appear in the output. Nor can you specify the language used in the output. Instead, both of these settings are given a default value. However, you can use **set dateformat** or **set language** to alter the output to match the setting of the information stored in the column. Also, Adaptive Server does not support modifying a column from *smalldatetime* to *char* datatype. See *Adaptive Server Reference Manual* for more information.

Modifying The Null Defalt Value of A Column

If you are changing only the NULL default value of a column (say, from NOT NULL to NULL), you do not need to specify a column's datatype. For example, the following modifies the *address* column in the *authors* table from NULL to NOT NULL:

```
alter table authors
modify address not null
```

If you modify a column and specify the datatype as NOT NULL, the operation succeeds as long as none of the rows have NULL values. If, however, any of the rows have a NULL value, the operation fails and any incomplete transactions are rolled back. For example, the following statement fails because the *titles* table contains NULL values for the *The Psychology of Computer Cooking*:

```
alter table titles
modify advance numeric(15,5) not null
```

```
Attempt to insert NULL value into column
'advance', table 'pubs2.dbo.titles';
column does not allow nulls. Update fails.
Command has been aborted.
```

Instead, first update the table to change all NULL values of the modified column to NOT NULL, and then re-issue the command.

Modifying Columns That Have Precision Or Scale

Check the length of your data before you modify the column's scale.

If an **alter table** command causes a column value to lose precision (say from *numeric*(10,5) to *numeric*(5,5)), Adaptive Server aborts the statement. If this statement is part of a batch, the batch is aborted if the option is turned on.

If an **alter table** command causes a column value to lose scale (say from *numeric(10, 5)* to *numeric(10,3)*, the rows are truncated without warning. This occurs whether or not **arithabort numeric_truncation** is on or off.

If **arithignore arith_overflow** is on and **alter table** causes a numeric overflow, Adaptive Server issues a warning. However, if **arithignore arith_overflow** is off, Adaptive Server does not issue a warning if **alter table** causes an numeric overflow. By default, **arithignore arith_overflow** is off when you first install Adaptive Server.

Make sure you review the data truncation rules and fully understand their implications before issuing commands that may truncate the length of the columns in your production environment. You should first perform the commands on a set of test columns.

Modifying *text* And *image* Columns

Text columns can only be converted to:

- *nchar*
- *varchar*
- *varchar*

Image columns can only be converted to:

- *varbinary*
- *binary*

You cannot modify *char* or *varchar* datatype columns to *text* columns. If you are converting from *text* to *char* or *varchar,* the maximum length

of the column is 255 bytes. If you do not specify a column length, **alter table** uses the default length of one byte. If you are modifying a multibyte character *text* or *image* column, and you do not specify a column length that is sufficient to contain the data, Adaptive Server truncates the data to fit the column length.

Adding, Dropping, and Modifying IDENTITY Columns

This section describes adding, dropping, and modifying IDENTITY columns using **alter table**. For a general discussion of IDENTITY columns, see "Using IDENTITY Columns" on page 7-8.

Adding IDENTITY Columns

You can add IDENTITY columns only with a default value of NOT NULL. If you specify a default value of NULL, Adaptive Server issues the following error message:

```
Msg 2764, Level 16, State 2:
Line 1:
Identity field 'record_id' must be a numeric with a scale of 0
and not null allowed.
```

You cannot specify a default clause for a new IDENTITY column.

To add an IDENTITY column to a table, specify the **identity** keyword in the **alter table** statement:

```
alter table table_name add column_name
      numeric(precision,0) identity not null
```

The following example adds an IDENTITY column, *record_id*, to the *stores* table:

```
alter table stores
    add record_id numeric(5,0) identity not null
```

When you add an IDENTITY column to a table, Adaptive Server assigns a unique sequential value, beginning with the value 1, to each row. If the table contains a large number of rows, this process can be time consuming. If the number of rows exceeds the maximum value allowed for the column (in this case, 10^5 - 1, or 99,999), the **alter table** statement fails.

Dropping IDENTITY Columns

You can drop IDENTITY columns just like any other column. For example the following drops the IDENTITY column created in the previous section:

```
alter table stores
drop record_id
```

However the following restrictions apply to dropping an identity column:

- You cannot drop IDENTITY columns if the **sp_dboption "identity in nonunique index"** is turned on in the database because the IDENTITY column is part of the index, and there are some indexes on the table. The IDENTITY column may be part of one or more indexes.

  To drop an IDENTITY column that may be part of other indexes, first drop the all the indexes, then drop the IDENTITY column, and then recreate the indexes.

  If the IDENTITY column is "hidden," you must first identify it using the **syb_identity** keyword. For more information, see "Referring to IDENTITY Columns with syb_identity" on page 7-19.

- You cannot drop an IDENTITY column from a table that has **set identity_insert** turned on (which enables explicit inserts into an IDENTITY column). This restriction exists because, if you dropped an IDENTITY column with **set identity_insert** turned on and added a new IDENTITY column, it could inherit the **set identity_insert** property of the original IDENTITY column, which might not have been your intent.

  Issue **sp_helpdb** to determine if **set identity insert** is turned on.

  To drop an IDENTITY column from a table that has **set identity_insert** turned on:

  1. Turn off the **set identity_insert** option

  ```
  set identity_insert table_name off
  ```

  2. Drop the IDENTITY table.

  3. Add the new IDENTITY table

  4. Turn on the **set identity_insert** option

  ```
  set identity_insert table_name on
  ```

You can modify the size of an IDENTITY column to increase their range. This might be necessary if either, your current range is too small, or the range was used up because of a server shutdown.

In the previous example, you can increase the range of *record_id* by:

```
alter table stores
modify record_id numberic(9,0)
```

Also, you can decrease the range by specifying a smaller precision for the target datatype. Note that, if the IDENTITY value in the table is too large for the range of the target IDENTITY column, an arithmetic conversion is raised and **alter table** aborts the statement.

- You cannot add a non-null IDENTITY column to a partitioned table using a set of **alter table** commands requiring a data copy. This is because data copy is done in parallel for a partitioned tables, and cannot guarantee unique IDENTITY values when inserting in parallel.

Data Copying

Adaptive Server performs a data copy only if it must temporarily copy data out of a table before it changes the table's schema. If the table has any indexes, Adaptive Server rebuilds the indexes with the **create index sorted_data** option after the data copy is finished.

➤ ***Note***

If **alter table** command is performing a data copy, the database that contains the table must have **select into/bulkcopy/pllsort** turned on. See the *Adaptive Server Reference Manual* for information about using **sp_dboption** to turn this option on.

Adaptive Server performs a data copy when:

- You drop a column.
- You modify the following properties of a column:
  - The column's datatype (except when you increase the length of *varchar*, *varbinary*, or NULL *char* or NULL *binary* columns)
  - Change the column from NULL to NOT NULL, or vice-versa.

- Decrease a column's length. If you decrease a column's length, you may not know beforehand if all the data will fit in the reduced column length. For example, if you decrease the *au_lname* column's length to a *varchar*(30), it may contain a name that requires a *varchar*(35). When you decrease a column's data length, Adaptive Server first performs a data copy to ensure that the change in the column length is successful.

- You increase the length of a number column (for example from *tinyint* to *int*), Adaptive Server performs data copying in case one row has a NOT NULL value for this column.
- You add a NOT NULL column.

alter table does not perform a data copy when:

- You change the length of either a *varchar* or a *varbinary* column.
- You change the user-defined datatype ID but the physical datatype doesn't change. For example, if your site has two datatypes *mychar1* and *mychar2* that have different user-defined datatypes but the same physical datatype, data copy does not happen if you change *mychar1* to *mychar2*.
- You change the NULL default value of a variable length column from NOT NULL to NULL.

To identify if **alter table** performs a data copy:

1. Set **showplan** on.
2. Set **noexec** on.
3. Perform the **alter table** command.

   Setting **showplan** on reports if Adaptive Server will perform a data copy. and setting **noexec** on ensures that Adaptive Serverdoes not perform any work. If no data copy is required, only catalog updates are performed to reflect the changes made by the **alter table** command.

Changing *exp_row_size*

If you perform a data copy, you can also change the **exp_row_size**, which allows you to specify how much space to allow per row. You can change the **exp_row_size** only if the modified table schema contains variable length columns, and only to within the range specified by the *maxlen* and *minlen* values in *sysindexes* for the modified table schema.

If the column has fixed-length columns, you can change the **exp_row_size** to only 0 or 1. If you drop all the variable length columns from a table, you must specify an **exp_row_size** of 0 or 1. Also, if you do not supply an **exp_row_size** with the **alter table** command, the old **exp_row_size** is used. Adaptive Server raises an error if the table contains only fixed length columns and the old **exp_row_size** is not compatible with the modified schema.

You cannot use the **exp_row_size** clause with any of the other **alter table** subclauses (for example, defining a constraint, changing the locking scheme, and so on). You can also use the **sp_chgattribute** to change the **exp_row_size**. For more information about changing the **exp_row_size** for both **alter table** and **sp_chgattribute**, see the *Adaptive Server Reference Manual*.

Modifing Locking Schemes and Table Schema

If **alter table** performs a data copy, you can also include a command to change the locking scheme of a table. For example, the following modifies the *au_lname* column of the *authors* table and changes the locking scheme of the table from allpages locking to datarows locking:

```
alter table authors
modify au_lname varchar(10)
lock datarows
```

However, you cannot use **alter table** to change table schema and the locking scheme of a table that has a clustered index. If a table has a clustered index you can:

1. Drop the index.
2. Modify the table schema and change the locking scheme in the same statement (if the change in the table schema also includes a data copy).
3. Rebuild the clustered index.

Alternately, you can issue an **alter table** command to change the locking scheme, then issue another **alter table** command to change the table's schema.

Altering Columns With User Defined Datatypes

You can use **alter table** to add, drop, or modify columns using user-defined datatypes.

Adding A Column With User-Defined Datatypes

Use the same syntax to add a column with a user-defined datatype as you would to add a column with a system-defined datatype. For example, the following adds a column to the *authors* table of *pubs2* using the *usertype* datatype:

```
alter table titles
add newcolumn usertype not null
```

If there is a default bound to the user defined datatype that is NULL, and the new column is NOT NULL, the default is not used.

The NULL or NOT NULL you specify as the default when you add a column takes precedence over the default specified by the user-defined datatype. That is, if you add a column and specify NOT NULL as the default, the new column has a default of NOT NULL even if the user defined datatype specifies NULL. If you do not specify NULL or NOT NULL, the default specified by the user-defined datatype is used.

You must supply a default clause when you add columns that are **not null**, even if there is a default bound to the user-defined datatype.

If the user-defined datatype specifies IDENTITY column properties (precision and scale), the column is added as an IDENTITY column.

Dropping A Column With User-Defined Datatypes

Drop a column that includes user-defined datatypes the same as you would drop a column that uses system-defined datatypes.

Modifying A Column With User-Defined Datatypes

The syntax to modify a column to include user defined datatypes is the same as modifying a column to include system-defined datatypes. For example, the following modifies the *au_lname* of the *authors* table to use the user-defined *newtype* datatype:

```
alter table authors
modify au_lname newtype(60) not null
```

If you do not specify either NULL or NOT NULL as the default, columns use the default specified by the user-defined datatype.

Modifying the table does not affect any current rules or defaults bound to the column. However, if you specify new rules or defaults, any old rules or defaults bound to the user-defined datatype are

dropped. If there are no previous rules or defaults bound to the column, any user-defined rules and defaults are applied.

You cannot modify an existing column to an IDENTITY column. You can only modify an existing IDENTITY column with user-defined datatypes that have IDENTITY column properties (precision and scale).

Errors And Warnings from *alter table*:

Most errors you encounter when running **alter table** inform you of schema constructs that prevent the requested command (for example, if you try to drop a column that is part of an index). You must fix the errors or warnings that refer to schema objects that depend on the affected column before you re-issue the command. To work with these errors:

1. Set **showplan** on.
2. Set **noexec** on.
3. Perform the **alter table** command.

alter table reports all error conditions it found. It does not perform any work because **noexec** is turned on. After you have changed the command to address any errors reported, make sure you set **showplan** and **noexec** to off so that Adaptive Server actually performs the work.

alter table detects and reports certain errors when actually running the command (for example, if you are dropping a column, the presence of a referential constraint). All run-time data dependent errors (for example, errors of numeric overflow, character truncation, and so on) can only be identified when the statement is executed. You must changing the command to fit the data available, or fix the data value(s) to work with the required target datatypes the statement specifies. To identify these errors, you will have to run the command without **noexec** turned on.

Errors And Warning Generated By alter table modify

➤ ***Note***

Make sure you understand the implications of modifying a datatype before you issue the command. Generally, use **alter table modify** only to implicitly convert between convertible datatypes. This ensures that any hidden conversions required during processing of **insert** and **update** statements do not fail because of datatype incompatibility.

Certain errors are only generated by the **alter table modify** command. Although the **alter table modify** command is used to convert columns to compatible datatypes, **alter table** may issue error if the columns you are converting have certain restrictions.

For example, if you add a column *second_advance* to the *titles* table with a datatype of *int* and create a clustered index on the *second_advance* column, you could not then modify this column to an *char* datatype. This would cause the *int* values to be converted from integers (1, 2, 3) to strings (‘1’, ‘2’, ‘3’). When the index is rebuilt with sorted data, the data values are expected to be in sorted order. But in this example, the datatype has changed from *int* to *char* and is no longer in sorted order for the *char* datatype’s ordering sequence. So, the **alter table** command fails during the index rebuild phase with the following error:

```
Msg 1530, Level 16, State 1: Line 1: Create index with
sorted_data was aborted because of row out of order. Primary key
of first out of order row is ‘”9    “’
```

You should be very cautious when choosing a new datatype for columns that are part of index key columns of clustered indexes. The **alter table modify** command must specify a target datatype which will not violate the ordering sequence of the modified data values after its data copy phase. Although this requirement is not enforced by Adaptive Server, it will be caught during the index rebuild phase.

alter table modify also issues an warning message if you modify the datatype to an incompatible datatype in a column that contains a constraint. For example if you try to modify from datatype *char* to datatype *int*, and the column includes a constraint, **alter table modify** issues the following warning:

```
Warning: a rule or constraint is defined on column ‘new_col’
being modified. Verify the validity of rules and constraints
after this ALTER TABLE operation.
```

```
Warning: column 'new_col' is referenced by one or more rules or
constraints. Verify the validity of the rules/constraints after
this ALTER TABLE operation.
```

This warning indicates that there might be some datatype inconsistency in the value that the constraint expects and the modified datatype of the column *new_col*.

Also, if you now attempt to insert data that is a datatype *char* into this table, it will fail with the following message:

```
Msg 257, Level 16, State 1: Line 1: Implicit conversion from
datatype 'CHAR' to 'INT' is not allowed.  Use the CONVERT
function to run this query.
```

This fails because the check constraint was defined to expect the column to be a datatype *int*, but the data being inserted is a datatype *char*.

Furthermore, you cannot insert the data as type *int* because column now uses a datatype of *char*. Instead, the **insert** returns the following message:

```
Msg 257, Level 16, State 1: Line 1: Implicit conversion from
datatype 'INT' to 'CHAR' is not allowed.  Use the CONVERT
function to run this query.
```

The **modify** operation is very flexible, but must be used with caution. In general, modifying to an implicitly convertible datatype works without errors. Modifying to an explicitly convertible datatype may lead to inconsistencies in the tables schema. Use **sp_depends** to indentify all column-level dependencies before modifying a column's datatype.

Scripts Generated by *if exists()...alter table*

Scripts that include constructs like the following may produce errors if the table described in the script does not include the column specified:

```
if exists (select 1 from syscolumns
    where id = object_id("some_table")
        and name = "some_column")
    begin
        alter table some_table drop some_column
    end
```

In this example, *some_column* **must** exist in the table *some_table* for the batch to succeed.

If *some_column* exists in *some_table*, the first time you run the batch, **alter table** drops the column. On subsequent executions, the batch fails to compile.

Adaptive Server raises these errors while preprocessing this batch, and are similar to those that are raised when a normal **select** tries to access a non-existent column. These errors are raised when you modify a table's schema using clauses that require a data copy. If you add a null column, and use the above construct, Adaptive Server does not raise these errors.

To avoid such errors when you modify a table's schema, include **alter table** in an execute immediate command:

```
if exists (select 1 from syscolumns
       where id = object_id("some_table")
         and name = "some_column")
begin
    exec ("alter table some_table drop
        some_column")
end
```

Because the execute immediate statement is only run if the **if exists()** function succeeds, Adaptive Server does not raise any errors when it compiles this script.

You do need to use the execute immediate construct for other uses of **alter table**, for example to change the locking scheme, and for any other cases when the command does not require data copy.

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**. For more information, see **sp_renamedb** in the *Adaptive Server Reference Manual*.

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
```

You must leave off the table name prefix from the new column name, or the new name will not be accepted. To change the name of an index, use this syntax:

```
sp_rename "table.index", newindexname
```

Do not include the table name in the new name.

To change the name of the user datatype *tid* to *t_id*, use this syntax:

```
exec sp_rename tid, "t_id"
```

You cannot change the name of system objects or system datatypes. You can change the names only of the objects you own. However, the Database Owner can change the name of any user's objects. Also, the object whose name you are changing must be in the current database.

Renaming Dependent Objects

Procedures, triggers, and views that depend on an object whose name has been changed work fine until they are recompiled. Recompiling takes place for many reasons and without notification to the user; for example, when a user loads a database, drops and re-creates a table, or drops an index.

The procedure, trigger, or view no longer works after it is recompiled by Adaptive Server. You must change the text of the affected procedure, trigger, or view to reflect the new object name. Also, the old object name appears 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.

You can use the **defncopy** utility program to copy the definitions of procedures, triggers, rules, defaults, and views into an operating system file. Edit this file to correct the object names, and use **defncopy** to copy the definition back into Adaptive Server. For more information on **defncopy**, refer to the *Utility Programs* manual for your platform.

Dropping Tables

Use the **drop table** command to remove a table from a database. Its syntax is:

```
drop table [[database.]owner.] table_name
    [, [[database.]owner.] table_name]...
```

When you issue this command, Adaptive Server removes the specified tables from the database, together with their contents and all 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 to 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.

Assigning Permissions to Users

The **grant** and **revoke** commands control the Adaptive 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** 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, 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 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 acquire any permission by using the **setuser** command to assume the identity of the object owner and then writing the appropriate **grant** or **revoke** statement.

You can assign two kinds of permissions 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 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*.

A System Security Officer can also use roles to simplify the task of granting and revoking access permissions.

For example, instead of having object owners grant privileges on each object individually to each employee, the System Security Officer can create roles, request object owners to grant privileges to each role, and grant these user-defined roles to individual employees, based on the functions they perform in the organization. The System Security Officer can also revoke user-defined roles granted to the employee.

For complete information about user-defined roles, see the *System Administration Guide*.

Getting Information About Databases and Tables

Adaptive Server provides several system procedures and built-in functions to get information about databases, tables, and other database objects. This section describes some of them.

For complete information about system procedures, see the *Adaptive Server Reference Manual*.

Getting Help on Databases

The system procedure **sp_helpdb** can report information about a specified database or about all databases in Adaptive Server. It reports 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    Jun 18 1997       no options set

(1 row affected)
device             size         usage
-----------------  -----------  --------------
pubsdev            2 MB         data + log

(1 row affected)
```

The **sp_databases** catalog stored procedure lists all the databases on a server. For example:

```
sp_databases
database_name     database_size  remarks
----------------- -------------  ------------
master                     5120  NULL
model                      2048  NULL
pubs2                      2048  NULL
pubs3                      2048  NULL
sybsecurity                5120  NULL
sybsystemprocs            30720  NULL
tempdb                     2048  NULL

(7 rows affected, return status = 0)
```

To find out who owns a database, use the **sp_helpuser** system procedure:

```
sp_helpuser dbo
Users_name    ID_in_db Group_name   Login_name
------------- -------- ------------ ------------
dbo                  1 public       sa

(return status = 0)
```

The system functions **db_id**() and **db_name**() identify the current database. For example:

```
select db_name(), db_id()

------------------------------ ------
master                              1
```

Getting Help on Database Objects

Adaptive Server provides system procedures, catalog stored procedures and built-in functions that return helpful information about database objects such as tables, columns, and constraints.

Using *sp_help* on Database Objects

The system procedure **sp_help** can report information about a specified database object (that is, any object listed in *sysobjects*), a specified datatype (listed in *systypes*), or all objects and datatypes in the current database.

The syntax is:

sp_help [*objname*]

Here is the output for the *publishers* table:

```
Name                           Owner       Type
-------------------------   -----------  ----------
publisher                   dbo          user table

(1 row affected)
Data_located_on_segment            When_created
-------------------------------    --------------------
default                              Jul 7 1997 1:43PM
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           pub_idrule           0
    1       NULL           NULL                 0
    1       NULL           NULL                 0
    1       NULL           NULL                 0

index_name index_description   index_keys index_max_rows_per_page
---------- ------------------  ---------- ----------------------
pubind     clustered, unique   pub_id     0
           located on default

(1 row affected)
```

```
keytype  object      related_object  related_keys
-------  ----------  --------------  ------------
primary  publishers  -- none --      pub_id, *, *, *, *, *
foreign  titles      publishers      pub_id, *, *, *, *, *

(1 row affected)
Object is not partitioned.

(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_helpconstraint* to Find a Table's Constraint Information

The system procedure **sp_helpconstraint** reports information about the declarative referential 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. **sp_helpconstraint** also reports the number of references associated with the specified tables.

Its syntax is:

```
sp_helpconstraint [objname] [, detail]
```

where *objname* is the name of the table being queried. Used by itself, **sp_helpconstraint** displays the number of references associated with each table in the current database. When specified with a table name, **sp_helpconstraint** reports the name, definition, and number of integrity constraints associated with the table. If you specify the **detail** option with this system procedure, it also returns information about the constraint's user or error messages.

For example, suppose you run **sp_helpconstraint** on the *store_employees* table in *pubs3*. The *store_employees* table was created as follows:

```
create table store_employees
(stor_id     char(4) null
       references stores(stor_id),
emp_id       id not null,
mgr_id       id null
       references store_employees(emp_id),
emp_lname    varchar(40) not null,
emp_fname    varchar(20) not null,
phone        char(12) null,
address      varchar(40) null,
city         varchar(20) null,
state        char(2) null,
country      varchar(12) null,
postalcode   varchar(10) null,
unique nonclustered (emp_id))
```

You can execute **sp_helpconstraint** to report its constraints:

```
name                          defn
----------------------------  --------------------------------
store_empl_stor_i_272004000   store_employees FOREIGN KEY
                              (stor_id) REFERENCES stores(stor_id)
store_empl_mgr_id_288004057   store_employees FOREIGN KEY
                              (mgr_id) SELF REFERENCES
                              store_employees(emp_id)
store_empl_2560039432         UNIQUE INDEX( emp_id) :
                              NONCLUSTERED, FOREIGN REFERENCE

(3 rows affected)

Total Number of Referential Constraints: 2
Details:
-- Number of references made by this table: 2
-- Number of references to this table: 1
-- Number of self references to this table: 1

Formula for Calculation:
Total Number of Referential Constraints
= Number of references made by this table
+ Number of references made to this table
- Number of self references within this table
```

A quick way to find the largest number of referential constraints associated with any table in the current database is to run **sp_helpconstraint** without specifying a table name, for example:

```
sp_helpconstraint
```

```
id          name                        Num_referential_constraints
----------- ------------------------- ----------------------------
   80003316 titles                                               4
   16003088 authors                                              3
  176003658 stores                                               3
  256003943 salesdetail                                          3
  208003772 sales                                                2
  336004228 titleauthor                                          2
  896006223 store_employees                                      2
   48003202 publishers                                           1
  128003487 roysched                                             1
  400004456 discounts                                            1
  448004627 au_pix                                               1
  496004798 blurbs                                               1

(11 rows affected)
```

In this report, the *titles* table has the largest number of referential constraints in the *pubs3* database.

Finding Out How Much Space a Table Uses

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.

Listing Tables, Columns, and Datatypes

Catalog stored procedures retrieve information from the system tables in tabular form. Also, you can supply wildcard characters for some parameters.

Listing Tables

The catalog stored procedure **sp_tables** lists all user tables in a database when used in the following format:

```
sp_tables @table_type = "'TABLE'"
```

Listing Columns and Datatypes

The catalog stored procedure **sp_columns** returns the datatype of any or all columns in one or more tables in a database. You can use wildcard characters to get information about more than one table or column.

For example, the following command returns information about all columns that includes the string “id” in all the tables with “sales” in their name:

```
   sp_columns "%sales%", null, null, "%id%"
table_qualifier table_owner
       table_name     column_name
       data_type type_name  precision  length  scale radix  nullable
       remarks

ss_data_type colid
---------------  -----------
      ----------       -----------
      ---------  --------- ---------  ------  ----- -----  --------
      -------

------------ -----
pubs2              dbo
       sales           stor_id
       1          char       4           4       NULL  NULL    0
NULL

47            1
pubs2            dbo
       salesdetail     stor_id
       1         char        4           4       NULL  NULL    0
NULL
```

```
4             1
pubs          dbo
        salesdetail     title_id
        12      varchar      6           6       NULL  NULL 0
      NULL

39            3

(3 rows affected, return status = 0)
```

Finding an Object Name and ID

The system functions **object_id**() and **object_name**() identify the ID and name of an object. For example:

```
select object_id("titles")

----------
  208003772
```

Object names and IDs are stored in the *sysobjects* system table.

8 Adding, Changing, and Deleting Data

After you create a database, tables, and indexes, you can put data into the tables and work with it—adding, changing, and deleting data as necessary.

This chapter discusses:

What Choices Are Available to Modify Data?

The commands you use to add, change, or delete data are called **data modification statements**. These commands are as follows:

- **insert** – adds new rows to a table.
- **update** – changes existing rows in a table.
- **writetext** – adds or changes *text* and *image* data without writing lengthy changes in the system's transaction log.
- **delete** – removes specific rows from a table.
- **truncate table** – removes all rows from a table.

For information about these commands, see the *Adaptive Server Reference Manual*.

Another method of adding data to a table is to transfer it from a file using the bulk copy utility program **bcp**. See the *Utility Programs* manual for your platform for more information.

You can modify data, using the **insert**, **update**, or **delete** statements, in only one table per statement. A Transact-SQL enhancement to these commands is that the modifications you make can be based on data in other tables, even those in other databases.

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** allow you to change data 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 you change it in the *authors* table, you must also change it in the *titleauthor* table and in any other table in the database that has a column containing that value. If you do not, you will not 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.

Keeping data modifications consistent throughout all tables in a database is called **referential integrity**. One way to deal with it is to define referential integrity constraints for the table. Another way is to create special procedures called triggers that take effect when you give **insert**, **update**, and **delete** commands for particular tables or columns (the **truncate table** command is not caught by triggers or referential integrity constraints). Triggers are discussed in Chapter 16, “Triggers: Enforcing Referential Integrity”; integrity constraints are discussed in Chapter 7, “Creating Databases and Tables.”

To delete data from referential integrity tables, change the referenced tables first and then the referencing table.

Transactions

A copy of the old and new state of each row affected by each data modification statement is written to the transaction log. (An exception is **writetext**, when the **select/into bulkcopy** database option is set to **false**.) 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 Adaptive 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 18, "Transactions: Maintaining Data Consistency and Recovery."

Using the Sample Databases

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* or *pubs3* 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* or *pubs3* database.

If you are starting with a clean *pubs2* or *pubs3* 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. For example, 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 Adaptive Server-supplied datatypes have special rules for entering and searching for data. For more information on datatypes, see Chapter 7, "Creating Databases and Tables."

char, *nchar*, *varchar*, *nvarchar*, and *text*

Remember that all character, *text*, and *datetime* data must be enclosed in single or double quotes when you first enter it and when you are searching for it. Use single quotes if the **quoted_identifier** option of the **set** command is set **on**. If you use double quotes, Adaptive Server treats the text as an identifier.

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:

- Use two quotes. For example, if you begin a character entry with a single quote and you want to include a single quote as part of the entry, use two single quotes: 'I don' 't understand.' For double quotes: "He said, ""It's not really confusing."""
- Enclose the quoted material 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 that is longer than the width of your screen, enter a backslash (\) before going to the next line.

Use the **like** keyword and wildcard characters described in Chapter 2, "Queries: Selecting Data from a Table," to search for *character*, *text*, and *datetime* data.

See the *Adaptive Server Reference Manual* for details on inserting *text* data and 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 variety of input formats. The display and entry formats are controlled separately. The default display format provides output that looks like "Apr 15 1997

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.

Adaptive 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 **on**; if you use double quotes, Adaptive Server treats the entry as an identifier.)

Adaptive 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, in which case, Adaptive Server uses the default. The default date and time 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. Adaptive 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. First, enter the hours; then minutes; then seconds; then milliseconds; then AM (or am) or PM (pm). 12AM is midnight. 12PM is noon. To be recognized as time, a value must contain either a colon or an AM or PM signifier. Note that *smalldatetime* is accurate only to the minute.

Milliseconds can be preceded by either 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 specifies 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 *Adaptive Server Reference Manual* for more information.

➤ ***Note***

dateformat affects only the 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”.

Date Formats

Adaptive Server recognizes three basic date formats, as described below. Each format must be enclosed in quotes and can be preceded or followed by a time specification, as described under “Entering Times” on page 8-5.

- The month is entered in alphabetic format.
  - Valid formats for specifying the date alphabetically are:

```
Apr[il] [15][,] 1997
Apr[il] 15[,] [19]97
Apr[il] 1997 [15]

[15] Apr[il][,] 1997
15 Apr[il][,] [19]97
15 [19]97 apr[il]
[15] 1997 apr[il]

1997 APR[IL] [15]
1997 [15] APR[IL]
```

  - 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 of less than 50 are interpreted as "20yy", and values of 50 or more are interpreted as "19yy".
  - Type the century only when the day is omitted or when you need a century other than the default (19).
  - If the day is missing, Adaptive Server defaults to the first day of the month.
  - When you specify the month in alphabetic format, the **dateformat** setting is ignored (see the **set** command in the *Adaptive Server Reference Manual*).

- The month is entered in numeric format, in a string with a slash (/), hyphen (-), or period (.) separator.
  - 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 the 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 in the *Adaptive Server Reference Manual* for more information.
  - To enter "April 15, 1997" in *mdy* **dateformat**, you can use these formats:

    ```
    [0]4/15/[19]97
    [0]4-15-[19]97
    [0]4.15.[19]97
    ```

  - The other entry orders are shown below with "/" as a separator; hyphens or periods can also be used:

```
15/[0]4/[19]97 (dmy)
1997/[0]4/15 (ymd)
1997/15/[0]4 (ydm)
[0]4/[19]97/15 (myd)
15/[19]97/[0]4 (dym)
```

- The date is given as an unseparated 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]960415.
  - 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’’.

The **set datefirst** command specifies the day of the week (Sunday, Monday, and so on) when **weekday** or **dw** is used with **datename**, and a corresponding number when used with **datepart**. Changing the language with **set language** can also affect the format for dates, depending on the default first day of the week value for the language. For the default language of *us_english*, the default **datefirst** setting is Sunday=1, Monday=2, and so on; others produce Monday=1, Tuesday=2, and so on. The default behavior can be changed on a per-session basis with **set datefirst**. See the **set** command in the *Adaptive Server Reference Manual* for more information.

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, Adaptive Server first converts the dates to the standard *datetime* format and then converts them to *varchar*. Since the standard display format does not 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, because *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 following clause would not find it because Adaptive Server converts the entry to "Jan 1, 1900 9:20AM":

```
where arrival_time = "9:20"
```

However, this 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 you enter *binary*, *varbinary*, or *image* data, the data must be preceded by "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 the *Adaptive Server Reference Manual* for information on trailing zeros in hexadecimal values.

money and *smallmoney*

Monetary values entered with the 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 3 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, Adaptive Server multiplies the mantissa by 10 raised to the given exponent. Table 8-1 shows 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	$-2 * 10^5$
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, Adaptive 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])]
```

Adaptive 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 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. The scale must be less than or equal to the precision. You can specify a scale ranging from 0 to 38 digits or use the default scale of 0 digits.

If a value exceeds the column's precision or scale, Adaptive 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, as 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. Adaptive Server updates the *timestamp* value after each

insert or update. See “Inserting Data into Specific Columns” on page 8-13 for more 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 specifies 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 (up to a limit of 16 tables, including the table into which you are inserting). 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 you add *text* or *image* values with **insert**, all the data is written to the transaction log. You can use the **writetext** command 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 8-13 and “Changing text and image Data” on page 8-29.

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, then 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 columns in a row by specifying only those columns and their data. All other columns that are not 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 is used.

You may especially want to use this form of the **insert** 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 are not 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, for example, *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
```

Adaptive 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.

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 4-digit number the first 2 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 name as a parameter 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.

Using the NULL Character String

Only columns for which NULL was specified in the **create table** statement and into which you have explicitly entered NULL (no quotes), or into which no data has been entered, contain null values. Avoid entering the character string "NULL" (with quotes) as data for a character column. It can only lead to confusion. Use "N/A" or "none" or a similar value instead. When you want to enter the value NULL explicitly, do **not** use single or double quotes.

To explicitly insert NULL into a column:

```
values({expression | null}
   [, {expression | null}]...)
```

The following example shows two equivalent **insert** statements. In the first statement, the user explicitly inserts a NULL into column *t1*. In the second, Adaptive Server provides a NULL value for *t1* because the user has not specified an explicit column value:

```
create table test
(t1 char(10) null, t2 char(10) not null)

insert test
values (null, "stuff")

insert test (t2)
values ("stuff")
```

NULL Is Not an Empty String

The empty string (" "or ' ') is always stored as a single space in variables and column data. This concatenation statement:

```
"abc" + "" + "def"
```

is equivalent to "abc def", not to "abcdef". The empty string is never evaluated as NULL.

Inserting Nulls into Columns That Do Not Allow Them

To insert data with **select** from a table that has null values in some fields into a table that does not allow null values, you must provide a substitute value for any NULL entries in the original table. For example, to insert data into an *advances* table that does not allow null values, this example substitutes "0" for the NULL fields:

```
insert advances
select pub_id, isnull(advance, 0) from titles
```

Without the **isnull** function, this command inserts all the rows with non-null values into *advances* and produces error messages for all the rows where the *advance* column in *titles* contains NULL.

If this kind of substitution cannot be made for your data, you cannot insert data containing null values into columns with a NOT NULL specification.

Adding Rows Without Values in All Columns

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 of the column. See Chapter 12, "Defining Defaults and Rules for Data," or **create default** in the *Adaptive 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 **create table** in the *Adaptive Server Reference Manual*.
- A unique, sequential value is entered if the column has the IDENTITY property.
- Adaptive Server rejects the row and displays an error message if NULL was not specified for the column when the table was created and no default exists.

Table 8-4 shows what you would see under these circumstances:

Table 8-4: Columns with no values

Default Exists for Column or Datatype	Column Defined NOT NULL	Column Defined to Allow NULL	Column Is 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**.

Changing a Column's Value to NULL

Use the **update** statement to set a column value to NULL. Its syntax is:

```
set column_name = {expression | null}
   [, column_name = {expression | null}]...
```

The following example finds all rows in which the *title_id* is TC3218 and replaces the *advance* with NULL:

```
update titles
set advance = null
where title_id = "TC3218"
```

Adaptive Server-Generated Values for IDENTITY Columns

When you insert a row into a table with an IDENTITY column, Adaptive 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*. Adaptive Server automatically generated the next sequential value, 2, for *row_id*:

```
select * from sales_daily
where stor_id = "7896"

sale_id      stor_id
-------      -------
      1      7896

(1 row affected)
```

Explicitly Inserting Data into an IDENTITY Column

At times, you may want to insert a specific value into an IDENTITY column, rather than accept a server-generated value. For example, you may want the first row inserted 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.

The table owner can explicitly insert a value into an IDENTITY column. The Database Owner and System Administrator can

explicitly insert a value into an IDENTITY column if they have been granted explicit permission by the table owner or if they are acting as the table owner through the **setuser** command.

Before inserting the data, set the **identity_insert** option **on** for the table. You can **set identity_insert on** for only one table at a time in a database within a session.

This example specifies a "seed" value of 101 for the IDENTITY column:

```
set identity_insert sales_daily on

insert sales_daily (syb_identity, stor_id)
values (101, "1349")
```

The **insert** statement lists each column, including the IDENTITY column, for which a value is specified. When the **identity_insert** option is set to **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 null values.

After you set **identity_insert** to **off**, you can insert IDENTITY column values automatically, without specifying the IDENTITY column, as before. Subsequent insertions use IDENTITY values based on the value explicitly specified after you set **identity_insert on**. For example, if you specify 101 for the IDENTITY column, subsequent insertions would be 102, 103, and so on.

➤ ***Note***

Adaptive 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, create a unique index on the IDENTITY column before inserting any rows.

Retrieving IDENTITY Column Values with @@*identity*

Use the @@*identity* global variable to retrieve the last value inserted into an IDENTITY column. The value of @@*identity* changes each time an **insert**, **select into**, or **bcp** statement attempts to insert a row into a table.

- If the statement affects a table without an IDENTITY column, @@*identity* is set to 0.
- If the statement inserts multiple rows, @@*identity* reflects the last value inserted into the IDENTITY column.

This change is permanent. *@@identity* does not revert to its previous value if the **insert**, **select into**, or **bcp** statement fails or if the transaction that contains it is rolled back. Also, the value for *@@identity* within a stored procedure or trigger does not affect the value outside the stored procedure or trigger.

For example:

```
select @@identity

---------------------------------------
                                    101

create procedure reset_id as
    set identity_insert sales_daily on
    insert into sales_daily (syb_identity, stor_id)
         values (102, "1349")
    select @@identity
select @@identity

execute reset_id

---------------------------------------
                                    102

select @@identity

---------------------------------------
                                    101
```

Reserving a Block of IDENTITY Column Values

The **identity grab size** configuration parameter allows each Adaptive Server process to reserve a block of IDENTITY column values for inserts into tables that have an IDENTITY column. This configuration parameter is a performance enhancement for multiprocessor environments. It reduces the number of times an Adaptive Server engine must hold an internal synchronization structure when inserting implicit identity values. For example, the following command sets the number of reserved values to 20:

```
sp_configure "identity grab size", 20
```

Afterward, when a user performs an insert into a table containing an IDENTITY column, Adaptive Server reserves a block of 20 IDENTITY column values for that user. Therefore, during the current session, the next 20 rows the user inserts into the table will have sequential IDENTITY column values. If a second user inserts rows into the same table while the first user is performing inserts, Adaptive Server will reserve the next block of 20 IDENTITY column values for the second user.

For example, suppose the following table containing an IDENTITY column has been created and the **identity grab size** is set to 10:

```
create table my_titles
(title_id   numeric(5,0)    identity,
title       varchar(30)     not null)
```

The first user, user 1, inserts the following rows into the *my_titles* table:

```
insert my_titles (title)
values ("The Trauma of the Inner Child")

insert my_titles (title)
values ("A Farewell to Angst")

insert my_titles (title)
values ("Life Without Anger")
```

Adaptive Server allows user 1 a block of 10 sequential IDENTITY values, for example, *title_id* numbers 1–10.

While user 1 is inserting rows to *my_titles*, the second user, user 2, begins inserting rows into *my_titles*. Adaptive Server grants user 2 the next available block of reserved IDENTITY values, that is, values 11–20.

If user 1 enters only three titles and then logs off Adaptive Server, the remaining seven reserved IDENTITY values are lost. The result is a gap in the table's IDENTITY values. Avoid setting the **identity grab size** too high, because this can cause gaps in the IDENTITY column numbering.

Reaching the IDENTITY Column's Maximum Value

The maximum value that you can insert into an IDENTITY column is 10 PRECISION - 1. If you do not specify a precision for the IDENTITY column, Adaptive Server uses the default precision (18 digits) for *numeric* columns.

Once an IDENTITY column reaches its maximum value, all further insert statements return an error that aborts the current transaction. When this happens, use **one** of the following methods to remedy the problem.

Create a New Table with a Larger Precision

If the table contains IDENTITY columns that are used for referential integrity, you need to retain the current numbers for the IDENTITY column values.

1. Use the **create table** command to create a new table that is identical to the old one except that it has a larger precision value for the IDENTITY column.
2. Use the **insert into** command to copy the data from the old table into to the new one.

Renumber the Table's IDENTITY Columns with bcp

If the table does not contain IDENTITY columns used for referential integrity, and if there are gaps in the numbering sequence, you can renumber the IDENTITY column to eliminate gaps, which allows more room for insertions. Manually inserting values into the IDENTITY column, deleting rows, rolling back transactions, restarting Adaptive Server, and setting the **identity grab size** too high can create gaps in IDENTITY column values.

To sequentially renumber IDENTITY column values (and thus remove the gaps), use the **bcp** utility as follows:

1. From the operating system command line, use **bcp** to copy out the data. For example:

   ```
   bcp pubs2..mytitles out mytitles_file -N -c
   ```

   The **-N** instructs **bcp** not to copy the IDENTITY column values from the table to the host file. The **-c** instructs **bcp** to use character mode.
2. In Adaptive Server, create a new table that is identical to the old table.
3. From the operating system command line, use **bcp** to copy the data into the new table:

   ```
   bcp pubs2..mynewtitles in mytitles_file -N -c
   ```

   The **-N** instructs **bcp** to have Adaptive Server assign the IDENTITY column values when loading data from the host file. The **-c** instructs **bcp** to use character mode.
4. In Adaptive Server, drop the old table, and use **sp_rename** to change the new table name to the old table name.

If the IDENTITY column is a primary key for joins, you may need to update the foreign keys in other tables.

By default, when you bulk copy data into a table with an IDENTITY column, **bcp** assigns each row a temporary IDENTITY column value of 0. As it inserts each row into the table, the server assigns it a unique, sequential IDENTITY column value, beginning with the next available value. To enter an explicit IDENTITY column value for

each row, specify the **-E** (UNIX) or **/identity** (OpenVMS) flag. Refer to the *Utility Programs* manual for your platform for more information on **bcp** options that affect IDENTITY columns.

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, columns in a table, make sure that a default exists or that NULL has been specified for the columns for which you are not inserting values. Otherwise, Adaptive Server returns 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 Adaptive 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 do not need to specify column names in either table. Suppose you have a table named *newauthors* that contains some rows of author information in the same format as in *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, suppose the **create table** statement for the *authors* table contained the columns *au_id*, *au_fname*, *au_lname*, and *address*, in that

order, and *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 two ways:

```
insert authors (au_id, address, au_lname, au_fname)
select * from newauthors
```

or

```
insert authors
select au_id, au_fname, au_lname, address
    from newauthors
```

If the column sequence in the two tables fails to match, Adaptive 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, which contains 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, 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, some authors in the *authors* table do not have titles and, therefore, 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, try this statement:

```
insert titleauthor (au_id)
select au_id
    from authors
    where au_id not in
    (select au_id from titleauthor)
```

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 enter 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 existing row in the same table. Make sure you follow the rule on the *pub_id* column. This is 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

Use the **update** command to change single rows, groups of rows, or all rows in a table. The **update** command is followed by the name of the table or view. As in all 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 an 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 canceled, for example, if it affects the table's IDENTITY column, or if 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.

Adaptive 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, try using 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 8-29.

update Syntax

A simplified version of the **update** syntax for updating specified rows with an expression is:

```
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 is how to change his row in the *authors* table:

```
update authors
set au_lname = "Health", au_fname = "Goodbody"
where au_lname = "Blotchet-Halls"
```

The following 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
```

You can set variables in an **update** statement with the following simplified syntax statement:

```
update table_name
   set variable_name = expression
      where search_conditions
```

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)} |
           variable_name1 =
           {expression1 | null | (select_statement)}
           [, column_name2 = {expression2 | null |
            (select_statement)}]... |
              variable_name2 = {expression1 | 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 are to be updated. If you do not have a **where** clause, the specified columns of **all** the rows are 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. See the installation and configuration guide for your platform for instructions on installing 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.

Assigning Variables in the *set* Clause

You can assign variables in the **set** clause of an **update** statement, similar to how you can assign them in a **select** statement. Using variables with **update** reduces lock contention and CPU consumption that can occur when extra **select** statements are used in conjunction with **update**.

The following example uses a declared variable to update the *titles* table:

```
declare @price money
select @price = 0
update titles
    set total_sales = total_sales + 1,
    @price = price
    where title_id = "BU1032"
select @price, total_sales
    from titles
    where title_id = "BU1032"
```

```
                              total_sales
 ------------------------ -----------
                    19.99        4096

(1 row affected)
```

For details on assigning variables in an **update** statement, see the *Adaptive Server Reference Manual*. For more information on declared variables, see “Local Variables” on page 13-31.

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 cities in Northern California.

Using the *from* Clause with *update*

Use the **from** clause to pull data from one or more tables into the table you are 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 using 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 NULL cannot be updated by selecting from a field that does, if any of the data is NULL.

As an alternative to the **from** clause in the **update** statement, you can use a subquery, which is ANSI-compliant.

Updating IDENTITY Columns

You can use the **syb_identity** keyword, qualified by the table name, where necessary, to update an IDENTITY column. For example, the following **update** statement finds the row in which the IDENTITY column equals 1 and changes the name of the store to "Barney's":

```
update stores_cal
set stor_name = "Barney's"
where syb_identity = 1
```

Changing *text* and *image* Data

Use the **writetext** command to change *text* or *image* values when you do not want to store long text values in the database transaction log. In these cases, do not use the **update** command, which can also be used for *text* or *image* columns, because **update** commands 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/pllsort on**. This permits the insertion of non-logged data. After using **writetext**, it is necessary to dump the database. You cannot use **dump transaction** after making unlogged changes to the database.

The **writetext** command completely overwrites any data in the column it affects. For **writetext** to work, the column must already contain a valid text pointer.

You can use the **textvalid()** function to check for a valid pointer, as follows:

```
select textvalid("blurbs.copy", textptr(copy))
from blurbs
```

There are two ways to create a text pointer:

- **insert** actual data into the *text* or *image* column
- **update** the column with data or a NULL

An "initialized" *text* column uses 2K of storage, even to store a couple of words. Adaptive Server saves space by not initializing text columns when explicit or implicit null values are placed in *text* columns with **insert.** The following code fragment inserts a value with a null text pointer, checks for the existence of a text pointer, and then updates the *blurbs* table. Explanatory comments are embedded in the text:

```
/* Insert a value with a text pointer. This could
** be done in a separate batch session from the
** of this example. */

insert blurbs (au_id) values ("267-41-2394")

/* Check for a valid pointer in an existing row.
** Use textvalid in a conditional clause; if no
** valid text pointer exists, update 'copy' to null
** to initialize the pointer. */

if (select textvalid("blurbs.copy", textptr(copy))
    from blurbs
    where au_id = "267-41-2394") = 0
begin
    update blurbs
       set copy = NULL
       where au_id = "267-41-2394"
end

/* Now that we know we have a valid pointer, we can
** use writetext to insert the text into the
** column. The next statements put the text
** into the local variable @val, then writetext
** places the new text string into the row
** pointed to by @val. */

declare @val varbinary(16)
select @val = textptr(copy)
    from blurbs
    where au_id = "267-41-2394"
writetext blurbs.copy @val
    "This book is a must for true data junkies."
```

For more information on batch files and the control-of-flow language used in this example, see Chapter 13, "Using Batches and Control-of-Flow Language."

Deleting Data

Like **insert** and **update**, **delete** works for both single-row and multiple-row operations, but it 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.]{view_name|table_name}
   [where search_conditions]

delete [[database.]owner.]{table_name | view_name}
   [from [[database.]owner.]{view_name|table_name
    [(index {index_name | table_name }
       [ prefetch size ][lru|mru])]}
    [, [[database.]owner.]{view_name|table_name
    (index {index_name | table_name }
      [ prefetch size ][lru|mru])]} ]...]
   [where search_conditions]

delete [from]
   [[database.]owner.]{table_name|view_name}
   where current of cursor_name
```

The optional **from** immediately after the **delete** keyword is included for compatibility with other versions of SQL. The **from** on the second line is an Adaptive 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 do not 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, because it does not allow you to delete any titles that have sales recorded in the *sales* table.

Deleting from IDENTITY Columns

You can use the **syb_identity** keyword in a **delete** statement on tables containing an IDENTITY column. For example, the following **delete** statement removes the row for which *row_id* equals 1:

```
delete sales_monthly
where syb_identity = 1
```

After you delete IDENTITY column rows, you may want to eliminate gaps in the table's IDENTITY column numbering sequence. See "Renumber the Table's IDENTITY Columns with bcp" on page 8-21.

Deleting All Rows from a Table

Use **truncate table** as a fast method of deleting all the rows in a table. It is 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.

You cannot use **truncate table** if another table has rows that reference it through a referential integrity constraint. Delete the rows from the foreign table, or truncate the foreign table and then truncate the primary table. See "General Rules for Creating Referential Integrity Constraints" on page 7-40.

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 16, "Triggers: Enforcing Referential Integrity," for details on triggers.

9 Views: Limiting Access to Data

A **view** is a named **select** statement that is stored in a database as an object. It allows you to view a subset of rows or columns in one or more tables. You use the view by invoking its name in other Transact-SQL statements. You can use views to focus, simplify, and customize each user's perception of the tables in a particular database. Views also provide a security mechanism by allowing users access only to the data they require.

This chapter discusses:

How Views Work

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 see only the data in which you are interested. That is why one speaks of looking at data or changing data "through" a view.

For example, suppose you are working on a project that is specific to the state of Utah. You can create a view that lists only the authors who live in Utah, as follows:

```
create view authors_ut
as select * from authors
where state = "UT"
```

To see the *authors_ut* view, enter:

```
select * from authors_ut
```

When the authors who live in Utah are added to or removed from the *authors* table, the *authors_ut* view reflects the updated *authors* table.

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 that are derived from them.

Advantages of Views

The examples in this chapter demonstrate how 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 on the data that interests them and on the tasks for which they are responsible. Data that is not of interest to a user 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 do not have to specify all the conditions and qualifications each time an operation is performed on that data.

Customization

Views allow different users to see the same data in different ways, even when they are 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 is derived are owned by the same user, that user 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, 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*.

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, suppose you restructure the database by using **select into** to split the *titles* table into these two new base tables and then dropping the *titles* table:

```
titletext (title_id, title, type, notes)

titlenumbers (title_id, pub_id, price, advance,
royalty, total_sales, pub_date)
```

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 a 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 retrieve only 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 $5000 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 only the records of interest are visible:

```
create view hiprice
as select *
from titles
where price > $15
  and advance > $5000
```

When Adaptive 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 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*, Adaptive 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
```

Adaptive Server 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, Adaptive 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 the 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, suppose 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 that is defined with a **select** * clause unless the view is dropped and redefined. This is because the asterisk in the original view definition considers only the original columns.

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 "Identifiers" on page 1-7.

You can build views on other views and procedures that reference views. You can define primary, foreign, and common keys on views. However, 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 the **create view** command:

```
create view [owner.]view_name
   [(column_name [, column_name]...)]
   as select [distinct] select_statement
   [with check option]
```

As illustrated in the **create view** example given under "View Examples" on page 9-5, you need not specify any column names in the **create** clause of a view definition statement. Adaptive Server gives the columns of the view the same names and datatypes as the columns referred to in the select list of the **select** statement. The select list can be designated by the asterisk (*****), as in the example, or it can be a full or partial list of the column names in the base tables.

To build views that do not contain duplicate rows, use the **distinct** keyword of the **select** statement to ensure that each row in the view is unique. However, **distinct** views cannot be updated.

It is always legal to specify column names. However, you **must** specify column names in the **create** clause for **every** column in the view if any of the following conditions are true:

- One or more 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 want to give a 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_view1 (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 = city, State = state
from publishers
```

The examples of view definition statements given in "Using the select Statement with create view" illustrate the rest of the rules for including column names in the **create** clause.

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. You can create a view using more than one table and other views by using 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 categories1 (category, average_price)
as select type, avg(price)
from titles
group by type
```

If you create a view for security reasons, 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 categories2 (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 “Organizing Query Results into Groups: The group by Clause” on page 3-7.

View Definition with a Join

You can create a view derived from more than one base table. Here is 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 column values match the same column values contained in another row. Two null values are considered to be identical.

Adaptive Server applies the **distinct** requirement to the view's definition when it accesses the view for the first time, 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 only some of the view's columns, but all of its rows), you can get results that 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*, that contains these 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

(3 rows affected)
```

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 by listing the column name, or the **syb_identity** keyword, in the view's **select** statement. For example:

```
create view sales_view
as select syb_identity, stor_id
from sales_daily
```

However, you cannot add a new IDENTITY column to a view by using the *identity_column_name* = **identity**(*precision*) syntax.

You can select the IDENTITY column from the view using the **syb_identity** keyword, unless the view:

- Selects the IDENTITY column more than once
- Computes a new column from the IDENTITY column
- Includes an aggregate function
- Joins columns from multiple tables
- Includes the IDENTITY column as part of an expression

If any of these conditions is true, Adaptive 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.

In the following example, the *row_id* column is not recognized as an IDENTITY column with respect to the *store_discount*s view because *store_discounts* joins columns from two tables:

```
create view store_discounts
as
select stor_name, discount
from stores, new_discounts
where stores.stor_id = new_discounts.stor_id
```

When you define the view, the underlying column retains the IDENTITY property. When you update a row through the view, you cannot specify a new value for the IDENTITY column. When you insert a row through the view, Adaptive Server generates a new, sequential value for the IDENTITY column. Only the table owner, Database Owner, or System Administrator can explicitly insert a value into the IDENTITY column after setting **identity_insert on** for the column's base table.

After Creating a View

After you create a view, the **source text** describing the view is stored in the *text* column of the *syscomments* system table. In previous releases of SQL Server, users often deleted the source text from *syscomments*, in order to save disk space and remove confidential information from this public area. **Do not remove this information from *syscomments***; doing so can cause problems for future upgrades of Adaptive Server. Instead, encrypt the text in *syscomments* by using the **sp_hidetext** system procedure, described in the *Adaptive Server Reference Manual*. For more information, see "Compiled Objects" on page 1-3.

Validating a View's Selection Criteria Using *with check option*

Normally, Adaptive 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 using the **with check option** clause, 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 is an example of a view, *stores_ca*, created using **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_ca
as select * from stores
where state = "CA"
with check option
```

When you try to insert a row through *stores_ca*, Adaptive 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_ca
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*, Adaptive 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_ca
set state = "CA"
where stor_id = "7066"
```

Views Derived from Other Views

When a view is created using **with check option**, all views 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_ca
where payterms = "Net 30"
```

Because *stores_cal* was created using **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"
```

Retrieving Data Through Views

When you retrieve data through a view, Adaptive 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, Adaptive 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 following view definition statement 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, Adaptive 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 "Views Derived from Other Views" on page 9-10.

➤ ***Note***

You can use **select** on *text* and *image* columns in views. However, you cannot use the **readtext** and **writetext** commands in views.

View Resolution

When you define a view, Adaptive Server checks to make sure that all the tables or views listed in the **from** clause exist. 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.

To fully resolve a view, Adaptive 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 under "Modifying Data Through Views" on page 9-17.

If any of these checks fails, Adaptive Server issues an error message.

Redefining Views

Unlike many other database management systems, Adaptive 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 Adaptive Server to translate the dependent view.

As an example, the *authors* table and three possible views are shown below. 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 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 these columns: *au_id*, *au_lname*, *au_fname*, *phone*, *address*, *city*, *state*, and *postalcode*.

You can drop *view2* and replace it with another view, also named *view2*, that contains slightly different selection criteria, such as:

```
create view view2
as select au_lname, phone
from view3
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 because it cannot derive the *phone* column from the object on which it depends.

However, *view3* still exists and you can use it 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 as 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**. Its syntax is:

```
sp_rename objname, newname
```

For example, to rename *titleview* to *bookview*, enter:

```
sp_rename titleview, bookview
```

Follow these conventions when renaming views:

- Make sure the new name follows the rules used for identifiers discussed under "Identifiers" on page 1-7. (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.
- Make sure the view is in the current database.

Altering or Dropping Underlying Objects

You can change the name of a view's underlying objects. For example, if a view references a table entitled *new_sales*, and you rename that table to *old_sales*, the view will work on the renamed table.

However, if a table referenced by a view has been dropped, and someone tries to use the view, Adaptive Server produces an error message. 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 Adaptive Server places no restrictions on retrieving data through views, and although Transact-SQL places fewer restrictions on modifying data through views than other versions of SQL, the following rules apply to various data modification operations:

- **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. Adaptive 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 the **with check option** clause.
- **update** statements are allowed on multitable views there **with check option** is used. 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, Adaptive Server checks to make sure that none of the above restrictions is violated and that no data integrity rules are violated.

Restrictions on Updating Views

Why can some views be updated and some cannot? You can best understand the restrictions by seeing examples of views that cannot be updated.

Restrictions on updated views apply to the following areas:

- Computed columns in a view definition
- **group by** or **compute** in a view definition
- Null values in underlying objects
- Views created using **with check option**
- Multitable views
- Views with IDENTITY columns

Computed Columns in a View Definition

This 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*, 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 do not make sense because there is no underlying value to delete.

group by or *compute* in a View Definition

This 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 are not 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 Adaptive Server does not support them.

Null Values in Underlying Objects

This restriction applies to **insert** statements when some NOT NULL columns are contained 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 business_titles
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 *business_view*. Although the *title* column does not 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 does not duplicate any value in the view.

Views Created Using *with check option*

This 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

This restriction determines what types of modifications you can make through views that join columns from multiple tables. Adaptive Server prohibits **delete** statements on multitable views, but allows **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 with IDENTITY Columns

This 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 following people:

- The table owner
- The Database Owner or the System Administrator if the table owner has granted them permission
- The Database Owner or the System Administrator if he or she impersonates the table owner by using the **setuser** command.

To enable such inserts through a view, use **set identity_insert on** for the column's base table. It is not sufficient to use **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 [owner.]view_name [, [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 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, Adaptive Server returns 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, and if it has the same name as the dropped table or view, 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 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

System procedures, catalog stored procedures, and Adaptive Server built-in functions provide information from the system tables about views.

For complete information about system procedures, see the *Adaptive Server Reference Manual*.

Getting Help on Views with *sp_help*

You can get a report on a view with the system procedure **sp_help.** For example:

```
sp_help hiprice
Name           Owner   type
---------      ------  -----
hiprice        dbo     view

(1 row affected)
Data_located_on_segment        When_created
------------------------------ --------------------
not applicable                 July 7 1997 11:57AM

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
total_sales     int         4         NULL    NULL
notes           varchar   200         NULL    NULL
pubdate         datetime    8         NULL    NULL
contract        bit         1         NULL    NULL

Nulls      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
0          NULL            NULL            0
0          NULL            NULL            0

No defined keys for this object.

(return status = 0)
```

Using *sp_helptext* to Display View Information

The System Security Officer must reset the **allow select on syscomments.text column** configuration parameter, in the evaluated configuration. (See **evaluated configuration** in the Glossary for more information.) When this happens, you must be the creator of the view or a System Administrator to view the text of a view through **sp_helptext.**

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)
```

If the source text of a view was encrypted using **sp_hidetext**, Adaptive Server displays a message advising you that the text is hidden. For information about hiding source text, see **sp_hidetext** in the *Adaptive Server Reference Manual*.

Using *sp_depends* to List Dependent Objects

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 is an example:

```
sp_depends titles

Things inside the current database that reference
the object.
```

```
object               type
-------------        ---------------------------
dbo.history_proc     stored procedure
dbo.title_proc       stored procedure
dbo.titleid_proc     stored procedure
dbo.deltitle         trigger
dbo.totalsales_trig  trigger
dbo.accounts         view
dbo.bookview         view
dbo.categories       view
dbo.hiprice          view
dbo.multitable_view  view
dbo.titleview        view

(return status = 0)
```

Listing All Views in a Database

The catalog stored procedure **sp_tables** lists all views in a database when used in the following format:

```
sp_tables @table_type = "'VIEW'"
```

Finding an Object Name and ID

The system functions **object_id**() and **object_name**() identify the ID and name of a view. For example:

```
select object_id("titleview")

----------
 480004741
```

Object names and IDs are stored in the *sysobjects* table.

Part 2: Advanced Topics

10 Using the Built-In Functions in Queries

Built-in functions, a Transact-SQL extension to SQL, return information from the database. You can use built-in functions in the **select** list, in the **where** clause, or anywhere an expression is allowed. You can also use them as part of a stored procedure or program.

This chapter discusses:

For reference material on the built-in functions, see the *Adaptive Server Reference Manual*.

System Functions That Return Database Information

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])
```

You can use the system functions 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)
```

Adaptive Server handles userids as follows:

- The System Administrator becomes the Database Owner in any database he or she is 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.

Table 10-1 lists 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.
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*.

Table 10-1: System functions, arguments, and results (continued)

Function	Argument	Result
data_pgs	([*dbid*,] *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. Use the function in a query run against the *sysindexes* table.
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 Adaptive Server process).
host_name	( )	Returns the current host computer name of the client process (not the Adaptive 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.
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.

Table 10-1: System functions, arguments, and results (continued)

Function	Argument	Result
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 or 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.
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).
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.
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.
used_pgs	(*object_id*, *doampg*, *ioampg*)	Returns the total number of pages used by a table and its clustered index.
user		Returns the user's name.

Table 10-1: System functions, arguments, and results (continued)

Function	Argument	Result
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.
valid_name	(*character_expression*)	Returns 0 if the *character expression* is not a valid identifier (illegal characters or more than 30 bytes long), or a number other than 0 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 Adaptive 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

The examples in this section use the following system functions:

- **col_length**
- **datalength**
- **isnull**
- **user_name**

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 and do not store trailing blanks. **datalength** of any NULL data returns NULL. When a *char* value is declared to allow NULLS, Adaptive Server stores it internally as a *varchar*. All other datatypes report their defined length. Here is 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 Used for Character Strings or Expressions

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 that 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.

You can concatenate *binary* and *varbinary* columns and *char*, *nchar*, *varchar*, and *nvarchar* columns. However, you **cannot** concatenate *text* or *image* columns.

You can nest string functions and use them anywhere an expression is allowed. When you use constants with a string function, enclose them in single or double quotes.

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. Adaptive Server automatically converts the argument to the desired type.

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 *char_expr1*, *char_expr2*, and so on.

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 that 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-matching wildcards supported by Adaptive Server.
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*.

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.

Table 10-3: String functions, arguments and results (continued)

Function	Argument	Result
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 in a table column, **char_length** returns the number of characters. For fixed-length data, it returns the defined length of the column. 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.
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, returns 0 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 "Character Strings in Query Results" on page 2-8 for a description of the wildcard characters that can be used in *pattern*. **patindex** 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	(*expression*)	Returns the reverse of the character or binary expression; if *expression* is "abcd", it returns "dcba"; if *expression* is 0x12345000, returns 0x00503412.
right	(*expression*, *integer_expr*)	Returns the part of the character or binary 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.

Table 10-3: String functions, arguments and results (continued)

Function	Argument	Result
soundex	(*char_expr*)	Returns a 4-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 nonnegative. 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*.
stuff	(*char_expr1*, *start*, *length*, *char_expr2*)	Delete *length* characters from *char_expr1* at *start*, and 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

The examples in this section use the following system functions:

- **charindex, patindex**
- **str**
- **stuff**
- **soundex, difference**
- **substring**

charindex, patindex

The **charindex** and **patindex** functions return the starting position of a pattern you specify. Both take two arguments, but they work slightly differently, since **patindex** can use wildcard characters, but **charindex** cannot. **charindex** can be used only on *char, nchar, varchar, nvarchar,*

binary, and *varbinary* columns; **patindex** works on *char*, *nchar*, *varchar*, *nvarchar*, and *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 are 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 Adaptive 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 that do not contain the pattern. In the following example, **patindex** finds all the rows in *sysobjects* that 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
syscolumns
syscomments
sysconstraints
sysdepends

(6 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.

Set length and decimal arguments to **str** 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_numeric* 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, *difference*

The **soundex** function converts a character string to a 4-digit code for use in a comparison. It ignores vowels 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 right(0x12345000, 3)	0x345000
select right(0x12345000, 6)	0x12345000
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

(1 row affected)
```

The following example shows how to extract the lower 4 digits from a binary field, where each position represents 2 binary digits:

```
select substring(xactid,5,2) from syslogs
```

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.

When you concatenate character strings, enclose each character expression in single or double quotes.

The concatenation syntax is:

select (*expression* + *expression* [+ *expression*]...)

Here is how to combine two character strings:

```
select ("abc" + "def")

-------
abcdef

(1 row affected)
```

This query displays California authors' 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 1986 12:00AM

(1 row affected)
```

Concatenation and the Empty String

Adaptive Server evaluates the empty string (“” or ‘’) as a single space. This statement:

```
select "abc" + "" + "def"
```

produces:

```
abc def
```

Nested String Functions

You can nest the string functions. 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, 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 2 characters of each *title_id* for books priced more than $20, type:

```
select substring(pub_id + title_id, 1, 6)
from titles
where price > $20

--------------
1389PC
0877PS
0877TC

(3 rows affected)
```

Text Functions Used for *text* and *image* Data

Text built-in functions are used for operations on *text* and *image* data. Table 10-5 lists text function names, arguments, and results:

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; returns 0 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 you are searching for first or last characters. See "Character Strings in Query Results" on page 2-8 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 that 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.

In addition to the functions in Table 10-5, **datalength** works on *text* columns. See "System Functions That Return Database Information" on page 10-1 for information about **datalength**.

The **set textsize** command specifies the limit, in bytes, of the *text* or *image* data to be returned with a **select** statement. For example, the following command sets the limit on *text* or *image* data returned with a **select** statement to 100 bytes:

```
set textsize 100
```

The current setting is stored in the *@@textsize* global variable. The default setting is controlled by the client program. To reset the default, issue the following command:

```
set textsize 0
```

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 **declare** example is the *@@textptr* global variable:

```
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 Adaptive 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.

Aggregate Functions

The aggregate functions generate summary values that appear as new columns in the query results. Table 10-6 lists the aggregate functions, their arguments, and the results they return.

Table 10-6: Aggregate functions

Function	Argument	Result
avg	(**all** | **distinct**) *expression*	Returns the numeric average of all (distinct) values
count	(**all** | **distinct**) *expression*	Returns the number of (distinct) non-null values or the number of rows
max	(*expression*)	Returns the highest value in an expression
min	(*expression*)	Returns the lowest value in a column
sum	(**all** | **distinct**) *expression*	Returns the total of the values

Examples are as follows:

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

----------------------- -----------
                6281.25      30,788

(1 row affected)
select count(distinct city) from authors

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

(1 row affected)
select discount from salesdetail
compute max(discount)

discount
--------------------
           40.000000
           ...
           46.700000

Compute Result:
--------------------
           62.200000

(117 rows affect)
select min(au_lname) from authors

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

(1 row affected)
```

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 10-7: 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.
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. Adaptive Server 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*).

Table 10-8 lists the mathematical functions, their arguments, and the results they return:

Table 10-8: 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.

Table 10-8: Mathematical functions (continued)

Function	Argument	Result
atn2	(*approx_numeric1*, *approx_numeric2*)	Returns the angle (in radians) whose tangent is (*approx_numeric1* / *approx_numeric2*).
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 that is 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 10-8: 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 equal to the precision of the first argument plus 1 and a 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. Value must be positive or 0.
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 10-37.

Some simple examples of mathematical functions follow:

Table 10-9: 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*, Adaptive Server rounds only the most significant digit. For example:

```
select round(55.55, -3)
```

returns a value of 100.00. (The number of zeros to the right of the decimal point is equal to the scale of the first argument's precision plus 1.)

Date Functions

The date built-in functions display information about dates and times. They manipulate *datetime* and *smalldatetime* values, performing arithmetic operations on them.

Adaptive Server stores values with the *datetime* datatype internally 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 1997 10:23PM
```

See "Using the General Purpose Conversion Function: convert" on page 10-32 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. Adaptive Server may round or truncate millisecond values.

Adaptive 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 10-10: 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 **datename** function produces ASCII values where appropriate, such as for the day of the week.

datepart returns a number that follows ISO standard 8601, which defines the first day of the week and the first week of the year. Depending on whether the **datepart** function includes a value for **calweekofyear**, **calyearofweek**, or **caldayorweek**, the date returned may be different for the same unit of time. For example, if Adaptive Server is configured to use US English as the default language:

```
datepart(cyr, "1/1/1989")
```

returns 1988, but:

```
datepart(yy, "1/1/1989)
```

returns 1989.

This disparity occurs because the ISO standard defines the first week of the year as the first week that includes a Thursday **and** begins with Monday.

For servers using US English as their default language, the first day of the week as Sunday, and the first week of the year is the week that contains January 4th.

Table 10-11 lists each date part, its abbreviation (if there is one), and the possible integer values for that date part

Table 10-11: Date parts

Date Part	Abbreviation	Values
year	yy	1753–9999
quarter	qq	1–4
month	mm	1–12
week	wk	1–54
day	dd	1–31
dayofyear	dy	1–366
weekday	dw	1– 7 (Sunday–Saturday)
hour	hh	0–23
minute	mi	0–59
second	ss	0–59
millisecond	ms	0–999

For example:

```
select datename (mm, "1997/06/16")
-----------
June

(1 row affected)
select datediff (yy, "1984", "1997")
-----------
        13

(1 row affected)
select dateadd (dd, 16, "1997/06/16")
------------------------------
           Jul  2 1997 12:00AM

(1 row affected)
```

Note that the values of the weekday date part are affected by the language setting.

Some examples of the week number date parts:

```
select datepart(cwk,"1997/01/31")
-----------
          5

(1 row affected)
select datepart(cyr,"1997/01/15")
-----------
       1997

(1 row affected)
select datepart(cdw,"1997/01/24")
-----------
          5

(1 row affected)
```

Table 10-12 lists the week number date parts, their abbreviations and values .

Table 10-12: Week number date parts

Date Part	Abbreviation	Values
calweekofyear	**cwk**	1–52

Table 10-12: Week number date parts

Date Part	Abbreviation	Values
calyearofweek	**cyr**	1753–9999
caldayofweek	**cdw**	1–7 (1 is Monday in us_english)

Table 2-5 of the *Adaptive Server Reference Manual* combines the values of Table 10-11 and Table 10-12 shows their interdependency.

Get Current Date: *getdate*

The **getdate** function produces the current date and time in Adaptive Server internal format for *datetime* values. **getdate** takes the NULL argument, ().

To find the current system date and time, type:

```
select getdate()

--------------------------
Aug 19 1997 12:45PM

(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.

To display the date using milliseconds, use the **convert** function, for example:

```
select convert(char(26), getdate(), 109)

--------------------------
Aug 19 1997 12:45:59:650PM

(1 row affected)
```

See “Changing the Display Format for Dates” on page 10-40 for more information.

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 0.

The following examples assume an April 12 date:

```
select datepart(month, getdate())
--------------
            8

(1 row affected)
select datename(month, getdate())
-------------
August

(1 row affected)
```

Calculate Intervals or Increment Dates

The **datediff** function calculates the amount of time in date parts between the first and second of the two dates you specify—in other words, it finds the interval between the two dates. The result is a signed integer value equal to *date2* - *date1* in date parts.

This query uses the date November 30, 1990 and finds the number of days that elapsed between *pubdate* and that date:

```
select pubdate, newdate = datediff(day, pubdate,
    "Nov 30 1990")
from titles
```

For the rows in the *titles* table having a *pubdate* of October 21, 1990, 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 pubdate, interval = datediff(month, pubdate,
    "Nov 30 1990")
from titles
```

This query produces a value of 1 for the rows with a *pubdate* in October 1990 and a value of 5 for the rows with a *pubdate* in June 1990. When the first date in the **datediff** function is later than the second date, 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 (-65) 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 calculating the difference.

Add Date Interval: *dateadd*

The **dateadd** function adds an interval (specified as a integer) 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 1986 12:00AM
Jun 12 1988 12:00AM
Jul  3 1985 12:00AM
Jun 25 1987 12:00AM
Jun 12 1989 12:00AM
Jun 21 1985 12:00AM
Jul  6 1997  1:43PM
Jul  3 1986 12:00AM
Jun 15 1987 12:00AM
Jul  6 1997  1:43PM
Oct 24 1990 12:00AM
Jun 18 1989 12:00AM
Oct  8 1990 12:00AM
Jun 15 1988 12:00AM
Jun 15 1988 12:00AM
Oct 24 1990 12:00AM
Jun 15 1985 12:00AM
Jun 15 1987 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. Adaptive Server

provides three datatype conversion functions, **convert**, **inttohex**, and **hextoint.**

Adaptive 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, Adaptive 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.

Adaptive 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 summarizes the datatype conversions supported by Adaptive 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, Adaptive Server generates an error message.
- Conversions of a type to itself are marked “-”. In general, Adaptive 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	E
varbinary	I	-	I	I	I	I	I	I	I	I	I	I	I	I	I	I	U	E
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	E	E
varchar, nvarchar	E	E	E	E	E	E	E	E	E	I	I	E	E	E	I	I	E	E
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	E	E	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**, converts between a variety of datatypes and specifies 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          375
Fifty Years in Buckingham Palace
    Kitchens                            15096
Sushi, Anyone?                           4095

(3 rows affected)
```

In the following 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, Adaptive Server uses the default length of 30 for character and binary data. If you do not specify a precision or scale, Adaptive Server uses the defaults of 18 and 0, respectively.

Conversion Rules

The following sections describe the rules Adaptive 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. However, if a *char* expression that

consists of a blank or blanks is converted to a *datetime* expression, Adaptive Server converts the blanks into the Sybase default *datetime* value of "Jan 1, 1900".

Adaptive Server generates syntax errors if the data includes unacceptable characters. The following types of characters 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 you convert 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

You can convert exact and approximate numeric data 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)
```

It fails because the *char* datatype is limited to 1 character, and the numeric 12.34 requires 5 characters for the conversion to be successful.

Rounding During Conversion to or from Money Types

The *money* and *smallmoney* types store 4 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 4 decimal 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 3 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 5:

```
select convert(int, $4.50)
-----------
          5

(1 row affected)
```

Adaptive Server assumes that data converted to *money* or *smallmoney* is in full currency units, such as dollars, rather than in fractional units, such as cents. For example, the integer value of 5 would be converted to the money equivalent of 5 dollars, not 5 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 were redefined in 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

Adaptive 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 a string of characters, rather than as 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, Adaptive 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** adds leading zeros to make it even, and then pads it with zeros on the right.

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.

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–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 8 digits, **hextoint** truncates it. If the data is less than 8 digits, **hextoint** right-justifies and pads it with zeros. 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*

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 Between Binary and Numeric or Decimal Types

In *binary* and *varbinary* data strings, the first two digits after "0x" represent the *binary* type: "00" represents a positive number and "01" represents a negative number. When you convert a *binary* or *varbinary* type to *numeric* or *decimal*, be sure to specify the "00" or "01" values after the "0x" digit; otherwise, the conversion will fail.

For example, here is how to convert the following *binary* data to *numeric*:

```
select convert(numeric
(38, 18),0x00000000000000006b14bd1e6eea0000000000000000000000000000000000)

----------------------
123.456000000000000000
```

This example converts the same *numeric* data back to *binary*:

```
select convert(binary,convert(numeric(38, 18), 123.456))

------------------------------------------------------------------
0x00000000000000006b14bd1e6eea0000000000000000000000000000000000
```

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 Adaptive 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 Adaptive Server handles these errors. The default setting, **arithabort arith_overflow on**, rolls back the entire transaction in which the error occurs. If you set **arithabort arith_overflow off**, Adaptive 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 Adaptive 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*, Adaptive 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**, Adaptive 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 65,536 on machines that consider byte 0 most significant and 256 on machines that consider byte 0 least significant.

The convert Function and Implicit Conversions

Binary types can be converted to integer types either explicitly, with the **convert** function, or implicitly. The data is stripped of its "0x" prefix and 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. Therefore, the 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 Types

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 variety of date display formats for 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 2 digits or 4 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*, the styles that include seconds or milliseconds will show zeros in those positions.

Table 10-13: 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*

Table 10-13: Converting date formats with the style parameter (continued)

Without Century (yy)	With Century (yyyy)	Standard	Output
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*.

Security Functions

The security functions return information about security services and user-defined roles. Table 10-14 lists the name of each security function, the argument it takes, and the result it returns.

Table 10-14: Security functions

Function	Argument	Result
is_sec_service_on	(*security_service_nm*)	Determines whether a particular security service is enabled. Returns 1 if the service is enabled. Otherwise, returns 0.
mut_excl_roles	("*role_1*", "*role_2*"[, "**membership**" | "**activation**"]*)*	Returns information about the level of mutual exclusivity between two roles.
proc_role	("*role_name*")	Returns 0 if the invoking user does not possess or has not activated the specified role; 1 if the invoking user has activated the specified role; and 2 if the user possesses the specified role directly or indirectly, but has not activated the role.
role_contain	(["*role1*", "*role2*"])	Returns 1 if the first role specified is contained by the second.
role_id	("*role_name*")	Returns the role id of the specified role name.
role_name	(*role_id*)	Returns the role name of the specified role id.

Table 10-14: Security functions (continued)

Function	Argument	Result
show_role	( )	Returns the login's current active roles, if any (**sa_role**, **sso_role**, **oper_role**, **replication_role**, or ***role_name***). If the login has no roles, returns NULL.
show_sec_services	( )	Returns a list of the available security services that are enabled for the current session

For more information about security features and user-defined roles, see the *Security Administration Guide* and the *Security Features User's Guide*.

11 Creating Indexes on Tables

An **index** provides quick access to data in a table, based on the values in specified columns. A table can have more than one index. Indexes are transparent to users accessing data from that table; Adaptive Server automatically decides when to use the indexes created for tables.

This chapter discusses:

For information on how you can design indexes to improve performance, see the *Performance and Tuning Guide*.

How Indexes Work

Indexes help Adaptive Server locate data. They speed up data retrieval by pointing to the location of a table column's data on disk. For example, suppose you need to run frequent queries using the identification numbers of stores in the *stores* table. To prevent Adaptive Server from having to search through each row in the table—which can be time-consuming if the *stores* table contains millions of rows—you could create the following index, entitled *stor_id_ind*:

```
create index stor_id_ind
on stores (stor_id)
```

The *stor_id_ind* index goes into effect automatically the next time you query the *stor_id* column in *stores*. In other words, 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; Adaptive Server decides whether to use the indexes for each query submitted for that table. As the data in a table changes over time, Adaptive Server may change the table's indexes to reflect those

changes. Again, these changes are transparent to users; Adaptive Server handles this task on its own.

Adaptive 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. Adaptive 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 Adaptive 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 quick 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, nonclustered indexes are automatically re-created when a clustered index is rebuilt.

Another reason is that inserting, deleting, or updating data in indexed columns takes longer than in unindexed columns. However, this cost is usually outweighed by the extent to which indexes improve retrieval performance.

When to Index

Use the following general guidelines:

- 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 Adaptive 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 can be 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.

When Not to Index

In some cases, indexes are not useful:

- Columns that are seldom or never referenced in queries do not benefit from indexes, since the system seldom 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

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] [with consumers = x]
```

The following sections 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 that 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.

Use composite indexes when two or more columns are best searched as a unit, for example, the first and last names of authors in the *authors* table. List the columns to be included in the composite index in sort-priority order, inside the parentheses after the table name, like this:

```
create index auth_name_ind
on authors(au_fname, au_lname)
```

The columns in a composite index do not have to be in the same order as the columns in the **create table** statement. For example, the order of *au_lname* and *au_fname* could be reversed in the preceding index creation statement.

You can specify up to 16 columns in a single composite index. All the columns in a composite index must be in the same table. The

maximum allowable size of the combined index values is 600 bytes. That is, the sum of the lengths of the columns that make up the composite index cannot exceed 600.

Using the *unique* Option

A unique index permits no two rows 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 because 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 Adaptive 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 "Using the ignore_dup_key Option" on page 11-12 for more information.

You can use the **unique** keyword on composite indexes. This was not done for the *auth_name_ind* index we just created.

Including IDENTITY Columns in Nonunique Indexes

The **identity in nonunique index** database 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 process updatable cursors and isolation level 0 reads.

To enable **identity in nonunique indexes**, enter:

```
sp_dboption pubs2, "identity in nonunique index", true
```

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 is positioned at the correct row the next time a **fetch** is performed on that cursor.

For example, after setting **identity in nonunique index** and **auto identity** to **true**, suppose you create the following table, which has no indexes:

```
create table title_prices
(title varchar(80)  not null,
price  money        null)
```

sp_help shows that the table contains an IDENTITY column, *SYB_IDENTITY_COL*, which is automatically created by the **auto identity** database option. If you create an index on the *title* column, use **sp_helpindex** to verify that the index automatically includes the IDENTITY column.

Ascending and Descending Index-Column Values

You can use the **asc** (ascending) and **desc** (descending) keywords to assign a sort order to each column in an index. By default, sort order is ascending.

Creating indexes so that columns are in the same order specified in the **order by** clauses of queries eliminates sorting the columns during query processing. The following example creates an index on the Orders table. The index has two columns, the first is *customer_ID*, in ascending order, the second is *date*, in descending order, so that the most recent orders are listed first:

```
create index nonclustered cust_order_date
   on Orders
   (customer_ID asc,
   date desc)
```

Using *fillfactor*, *max_rows_per_page*, and *reservepagegap*

fillfactor, **max_rows_per_page**, and **reservepagegap** are space-management properties that apply to tables and indexes and affect the way physical pages are filled with data. For a detailed discussion of setting space-management properties for indexes, see **create index** in the *Reference Manual*. The following table summarizes information about the space-management properties for indexes.

Table 11-1: Summary of space-management properties for indexes.

Property	Description	Use	Comments
fillfactor	For a clustered index, specifies the percent of space on a page that can be filled when the index is created. A **fillfactor** under 100% leaves space for inserts into a page without immediately causing page splits. Benefits: • Initially, fewer page splits. • Reduced contention for pages, because there are more pages and fewer rows on a page.	Applies only to a clustered index on a data-only-locked table	The **fillfactor** percentage is used only when an index is created on a table with existing data. It does not apply to pages and inserts after a table is created. If no **fillfactor** is specified, the system-wide default **fillfactor** is used. Initially, this is set to 100%, but can be changed using **sp_configure**.
max_rows_per_page	Specifies the maximum number of rows allowed per page. Benefit: • Can reduce contention for pages by limiting the number of rows per page and increasing the number of pages.	Applies only to allpages-locked tables. The maximum value that you can set **max_rows_per_page** to is 256.	**max_rows_per_page** applies at all times, from the creation of an index, onward. If **max_rows_per_page** is not specified, the default is as many rows as will fit on a page.

Table 11-1: Summary of space-management properties for indexes.

Property	Description	Use	Comments
reservepagegap	Determines the number of pages left empty when extents are allocated. For example, a reservepagegap of "16" means that 1 page of the 16 pages in 2 extents is left empty when the extents are allocated. Benefits: • Can reduce row forwarding and lessen the frequency of maintenance activities such as running **reorg rebuild** and recreating indexes.	Applies to pages in all locking schemes.	If **reservepagegap** is not specified, no pages are left empty when extents are allocated.

The following **create index** statement sets the **fillfactor** for an index to 65% and sets the **reservepagegap** to one empty page for each extent allocated:

```
create index postalcode_ind2
   on authors (postalcode)
   with fillfactor = 10, reservepagegap = 8
```

Using Clustered or Nonclustered Indexes

With a clustered index, Adaptive Server sorts rows on an ongoing basis so that their physical order is the same as their logical (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, the database's design determines a primary key. 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**.

Also, 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**. However, these procedures do not enforce the key relationships.

You can display information about defined keys with **sp_helpkey** and about columns that are likely join candidates with **sp_helpjoins**.

For a definition of primary and foreign keys, see Chapter 16, "Triggers: Enforcing Referential Integrity." For complete information on system procedures, see the *Adaptive 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, Adaptive Server creates a nonclustered index.

Here is how the *titleidind* 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** as follows:

```
drop index titles.titleidind
```

Then, create the clustered index:

```
create clustered index titleidind
on titles(title_id)
```

If you think you will often want to sort the people in the *friends_etc* table, which you created in Chapter 7, 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, for example:

```
create clustered index nmind
on friends_etc(pname, sname)
```

Creating Clustered Indexes on Segments

The **create index** command allows you to create the index on a specified segment. 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 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.

Creating Clustered Indexes on Partitioned Tables

You can create a clustered index on a partitioned table if all of the following conditions are true:

- The server is configured to run in parallel,
- The **select into/bulkcopy/pllsort** database option is set to **true**, and
- As many worker threads are available as the number of partitions.

However, to speed recovery, dump the database after creating the clustered index.

For more information about partitioned tables and creating clustered indexes on them, see the *System Administration Guide*. See a System Administrator or the Database Owner before creating a clustered index on a partitioned table.

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.** Table 11-2 shows which option to use, based on the type of index:

Table 11-2: 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 canceled. You can avoid having a large transaction canceled 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 canceled with an error message. After the cancellation, any transaction that was active at the time may continue as though the **update** or **insert** had never taken place. Nonduplicate keys are inserted normally.

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, Adaptive 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 an Adaptive 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.

A non-unique clustered index will allow duplicate keys, but will not allow duplicate rows unless you specify **allow_dup_row.**

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 **insert** or **update** duplicate rows.

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 option if a unique clustered index exists on any column in the table.

The **ignore_dup_row** option eliminates duplicates from a batch of data. When you enter a duplicate row, Adaptive Server ignores that row and cancels that particular **insert** or **update** with an informational error message. After the cancellation, any transaction that may have been active at the time continues as though the insert or update had never taken place. 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.

Table 11-3 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 11-3: Duplicate row options in indexes

Option	Has Duplicates	Enter Duplicates
Neither option set	**create index** command fails.	Command fails.
allow_dup_row set	Command completes.	Command completes.
ignore_dup_row set	Index created but duplicate rows deleted; error message.	Duplicates not inserted / updated; error message; transaction completes.

Using the *sorted_data* Option

The **sorted_data** option of **create index** 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 1GB.

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.

Certain other **create index** options require a sort even if **sorted_data** is specified. See the **create index** description in the *Adaptive Server Reference Manual*.

Using the *on segment_name* Option

The **on *segment_name*** clause specifies 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
```

If you use **segment_name** when creating a clustered index, the table containing the index moves to the segment you specify. See a System Administrator or the Database Owner before creating tables or indexes on segments; certain segments may be reserved for performance reasons.

Dropping Indexes

The **drop index** command removes an index from the database. Its syntax is:

```
drop index table_name.index_name
   [, table_name.index_name]...
```

When you use this command, Adaptive 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
                                                index_max_rows_per_page
------------ ----------------------------    -------------
                                                -----------------------
nmind       clustered located on default    pname, sname
                                                0
postalcodeind nonclustered located on default postalcode
                                               0

(2 rows affected, return status = 0)
```

sp_help runs **sp_helpindex** at the end of its report.

The catalog stored procedure **sp_statistics** returns a list of indexes on a table. For example:

```
sp_statistics friends_etc
table_qualifier               table_owner
         table_name                        non_unique
         index_qualifier                   index_name
         type   seq_in_index column_name                      collation
         cardinality pages
------------------------------- -------------------------------
        ------------------------------- ----------
        ------------------------------- ------------------------------
        ------ ------------ -------------------------------- ---------
        ----------- -----------
pubs2                           dbo
        friends_etc                             NULL
        NULL                                NULL
             0          NULL NULL                               NULL
                   0           1
pubs2                             dbo
```

```
         friends_etc                              1
         friends_etc                        nmind
            1           1 pname                        A
                   0           1
pubs2                              dbo
         friends_etc                               1
         friends_etc                         nmind
               1            2 sname                           A
                   0            1
pubs2                              dbo
         friends_etc                                1
         friends_etc                        postalcodeind
             3           1  postalcode                        A
                NULL       NULL

(4 rows affected, return status = 0)
```

In addition, if you follow the table name with "1", the **sp_spaceused** system procedure reports the amount of space used by a table and its indexes. For example:

```
sp_spaceused friends_etc, 1
```

```
index_name           size       reserved   unused
-------------------- ---------- ---------- ----------
nmind                2 KB       32 KB      28 KB
postalcodeind        2 KB       16 KB      14 KB

name         rowtotal   reserved     data    index_size      unused
------------ ---------- ------------ ------- --------------- --------
friends_etc  1          48 KB        2 KB    4 KB            42 KB

(return status = 0)
```

Updating Statistics About Indexes

The **update statistics** command helps Adaptive 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. Use **update statistics** when a large amount of data in an indexed column has been added, changed, or deleted.

When Component Integration Services is enabled, **update statistics** can generate accurate distribution statistics for remote tables. For more information, see the *Component Integration Services User's Guide.*

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  index_max_rows_per_page
---------- -----------------  ----------  -----------------------
auidind    clustered, unique  au_id       0
aunmind    nonclustered       au_lname,   0
                              au_fname

(2 rows affected, return status = 0)
```

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
```

Because 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. Adaptive Server runs **update statistics** automatically when you create an index on existing data.

Updating Partition Statistics

Like the **update statistics** command for unpartitioned tables, the **update partition statistics** command helps Adaptive Server make the best decisions when it processes a query, by keeping it up to date about the number of pages within the partitions. Use **update partition statistics** when a large amount of data in a partitioned table has been added, changed, or deleted.

Permission to issue the **update partiton statistics** command defaults to the table owner and is not transferable. Its syntax is:

```
update partition statistics table_name
```

For example, suppose the *authors* table was partitioned as follows:

```
alter table authors partition 3
```

Then you run **sp_helpartition** to see how the partitions were distributed:

```
sp_helpartition authors
```

```
partitionid firstpage   controlpage ptn_data_pages
----------- ----------- ----------- --------------
          1         553         554              1
          2         817         816              1
          3        1009        1008              1

(3 rows affected, return status = 0)
```

Afterwards, you update the statistics for *authors* as follows:

```
update partition statistics authors
```

Using **sp_helpartition** on *authors* shows the following update:

```
partitionid firstpage   controlpage ptn_data_pages
----------- ----------- ----------- --------------
          1         553         554             10
          2         817         816              1
          3        1009        1008              1

(3 rows affected, return status = 0)
```

Dropping and re-creating a clustered index automatically redistributes the data within partitions and updates the partition statistics.

12 Defining Defaults and Rules for Data

A **default** is a value that Adaptive Server inserts into a column if the user does not explicitly enter a value for that column. 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.

This chapter discusses:

How Defaults and Rules Work

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.

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 Adaptive Server.

Defaults allow you to specify a value that Adaptive 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, Adaptive Server checks it against the most recent rule that has been bound to the specified column. Data entered before 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.

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.

When you use the **default** clause of the **create table** or **alter table** commands, you can specify the keyword **user** to insert the name of the user who is inserting the data, **null** to insert the null value, or a constant expression compatible with the datatype of the column. The constant expression cannot include the name of any columns or other database objects, but can include built-in functions. To replace the user name or constant expression with a default you create, use the **alter table** command to replace the default with **null** and then issue the **sp_bindefault** command.

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. In general, to create defaults you:

1. Use the **create default** command to define the default itself.
2. Use the **sp_bindefault** system procedure to bind the default to the appropriate table column or user-defined datatype.
3. Test the bound default by inserting data. Many errors in creating and binding defaults can be caught only by testing with an **insert** or **update** command.

You can drop defaults with the **drop default** command and remove their association with **sp_unbinddefault**.

Be sure to check the following when you create and bind defaults:

- Make sure the column is large enough for the default. For example, *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 under "Binding Defaults" on page 12-4.
- 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 reject the default entry. Either change the default or change the rule.

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. For example, 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.

Another example: Suppose you want to create a default value of "Oakland" that can be used with the *city* column of *friends_etc* 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, type:

```
create default citydflt
as "Oakland"
```

After **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 is appropriate.

If you specify NOT NULL when you create a column and do not associate a default with it, Adaptive Server produces 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. For example, suppose you create the following default:

```
create default advancedflt as "UNKNOWN"
```

Now, bind the default to the appropriate column or user-defined datatype with the **sp_bindefault** system procedure.

```
sp_bindefault advancedflt, "titles.advance"
```

The default takes effect only if the user does not add an entry to the *advance* column of the *titles* table. No entry is different from entering a null value. A default can connect to a particular column, to a number of columns, or to all columns in the database having a given user-defined datatype.

➤ ***Note***

To get the default, you must issue an **insert** or **update** command with a column list that does not include the column that has the default.

The following restrictions apply to defaults:

- The default applies to new rows only. It does not retroactively change existing rows. Of course, it takes effect only when no

entry is made. If you supply any value for the column, including NULL, the default has no effect.

- You cannot bind a default to a system datatype, because the target would be too broad.
- You cannot bind a default to a *timestamp* column, because Adaptive Server generates values for *timestamp* columns.
- You cannot bind defaults to system tables. If you try to bind a default to a system table, Adaptive Server displays an error message and does not bind the default.
- You can bind a default to an IDENTITY column or to a user-defined datatype with the IDENTITY property, but Adaptive Server ignores such defaults. Each time you insert a row into a table without specifying a value for the IDENTITY column, Adaptive Server assigns a value that is 1 greater than the last value assigned.
- If a default already exists on a column, you must remove it before binding a new default. Use **sp_unbindefault** to remove defaults created with **sp_bindefault**. Use **alter table** to remove defaults created with **create table**.

To bind *citydflt* to the *city* column in *friends_etc*, type:

```
sp_bindefault citydflt, "friends_etc.city"
```

Notice that the table and column name are enclosed in quotes, because of the embedded punctuation (the period).

If you create a special datatype for all city columns in every table in your database, and bind *citydflt* to that datatype, "Oakland" will appear 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
```

To prevent existing columns or a specific user datatype from inheriting the new default, use the **futureonly** parameter when binding a default to a user datatype. However, do not use **future only** when binding a default to a column. Here is how you create and bind the new default "Berkeley" to the datatype *citytype* for use by new table columns only:

```
create default newcitydflt as "Berkeley"

sp_bindefault newcitydflt, citytype, futureonly
```

"Oakland" will continue to appear as the default for any existing table columns using *citytype*.

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, along with its binding, that is appropriate for a section of Oakland:

```
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 you use the optional third parameter, **futureonly**, which prevents existing columns of that user datatype from inheriting the default.

➤ ***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 is not connected to that column.

To unbind a default from the user-defined datatype *citytype*, use this command:

```
sp_unbindefault citytype
```

The complete syntax of **sp_unbindefault** system is:

```
sp_unbindefault objname [, futureonly]
```

If the *objname* parameter you give is not of the form *table.column*, Adaptive 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.

How Defaults Affect Null Values

If you specify NOT NULL when you create a column and do not create a default for it, Adaptive 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, Adaptive Server inserts NULL in that position each time you add rows without entering any value for that column. When you drop a default for a NOT NULL column, you get an error message when rows are added, but no value for that column is explicitly entered.

Table 12-1 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 12-1: Column definition and null defaults

Column Definition	User Entry	Result
Null and default defined	No value	Default used
	NULL value	NULL used
Null defined, no default defined	No value	NULL used
	NULL value	NULL used
Not null, default defined	No value	Default used
	NULL value	NULL used
Not null, no default defined	No value	Error
	NULL value	Error

After Creating a Default

After you create a default, the **source text** describing the default is stored in the *text* column of the *syscomments* system table. In previous releases of SQL Server, users often deleted the source text from *syscomments*, in order to save disk space and remove confidential information from this public area. **Do not remove this information from *syscomments***; doing so can cause problems for future upgrades of Adaptive Server. Instead, encrypt the text in *syscomments* by using the **sp_hidetext** system procedure, described in the *Adaptive Server Reference Manual*. For more information, see "Compiled Objects" on page 1-3.

Dropping Defaults

To remove a default from the database entirely, use the **drop default** command. Be sure to unbind the default from all columns and user datatypes before you drop it. (See "Unbinding Defaults" on page 12-6.) If you try to drop a default that is still bound, Adaptive Server displays an error message and the **drop default** command fails.

Here is how to remove *citydflt*. First, you unbind it, for example:

```
sp_unbindefault citydft
```

Then you can drop *citydft*:

```
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.

Creating Rules

A rule lets you specify what users can or cannot enter into a particular column or any column with a user-defined datatype. In general, to create a rule you:

1. Use the **create rule** command to create the rule.
2. Use the system procedure **sp_bindrule** to bind the rule to a column or user-defined datatype.

3. Test the bound rule by inserting data. Many errors in creating and binding rules can be caught only by testing with an **insert** or **update** command.

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. For example, 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 "415" and be followed by 7 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.

Use the optional third parameter, **futureonly**, 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, Adaptive Server maintains the changed rule for existing columns of that user-defined datatype.

The following restrictions apply to rules:

- You cannot bind a rule to a *text*, *image*, or *timestamp* datatype column.
- Adaptive Server does not allow rules on system tables. If you try to bind a rule to a system table, Adaptive Server returns an error message and rejects the rule.

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 is a rule to ensure that all the postal codes you enter will have 946 as the first 3 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 that invoke the rule.

Rules Bound to User-Defined Datatypes

You cannot 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 replaces a rule bound to the user datatype of that column, but binding a rule to a datatype does 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 12-2: 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. To avoid accidentally typing an amount outside those limits, create a rule like this one:

```
create rule debtrule
as @debt = $0.00 or @debt between $5.00 and $200.00
```

The *@debt* rule definition allows for an entry of $0.00 in order to maintain the default previously defined for this column.

Bind *debtrule* to the *debt* column like this:

```
sp_bindrule debtrule, "friends_etc.debt"
```

Rules and Null Values

You cannot define a column to allow nulls, and then override this definition with a rule that prohibits null values. For example, if a column definition specifies NULL and the rule specifies the following, an implicit or explicit NULL does not violate the rule:

```
@val in (1,2,3)
```

The column definition overrides the rule, even a rule that specifies:

```
@val is not null
```

After Defining a Rule

After you define a rule, the **source text** describing the rule is stored in the *text* column of the *syscomments* system table. In previous releases of SQL Server, users often deleted the source text from *syscomments*, in order to save disk space and remove confidential information from this public area. **Do not remove this information from syscomments**; doing so can cause problems for future upgrades of Adaptive Server. Instead, encrypt the text in *syscomments* by using the **sp_hidetext** system procedure, described in the *Adaptive Server Reference Manual*. For more information, see “Compiled Objects” on page 1-3.

Unbinding Rules

Unbinding a rule disconnects 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#*, use this command:

```
sp_unbindrule "p#"
```

The complete syntax of **sp_unbindrule** is:

```
sp_unbindrule objname [, futureonly]
```

If the *objname* parameter you use is not of the form *"table.column"*, Adaptive 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

To remove a rule from the database entirely, use the **drop rule** command. Be sure to unbind the rule from all columns and user datatypes before you drop it. If you try to drop a rule that is still bound, Adaptive 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.

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. 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 *pubs2* 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#*, use this command:

```
sp_help "p#"
```

The **sp_helptext** procedure reports the definition (the **create** statement) of a rule or default.

If the source text of a default or rule was encrypted using **sp_hidetext**, Adaptive Server displays a message advising you that the text is hidden. For information about hiding source text, see **sp_hidetext** in the *Adaptive Server Reference Manual*.

If the System Security Officer has reset the **allow select on syscomments.text column** parameter with the system procedure **sp_configure** (as required to run Adaptive Server in the evaluated configuration), you must be the creator of the default or rule or a System Administrator to view the text of a default or rule through **sp_helptext**. For more information, see **evaluated configuration** in the *Adaptive Server Glossary*.

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.

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. Adaptive 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.

This chapter discusses:

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 statements to Adaptive Server one at a time, entering the statement and receiving results interactively.

Adaptive Server can also process multiple statements submitted as a batch, either interactively or from a file. A batch or batch file is a set of Transact-SQL statements that are submitted together and executed as a group, one after the other. A batch is terminated by an end-of-batch signal. With the **isql** utility, this is the word "go" on a line by itself. For details on **isql**, see the the *Utility Programs* manual for your platform.

Here is an example of a batch that contains two Transact-SQL statements:

```
select count(*) from titles
select count(*) from authors
go
```

Technically, a single Transact-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 Transact-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 Transact-SQL statements. The Transact-SQL control-of-flow language transforms standard SQL into a very high-level programming language.

Rules Associated with Batches

There are rules governing which Transact-SQL statements can be combined into a single batch. These batch rules are as follows:

- Before referencing objects in a database, issue a **use** statement for that database. For example:

  ```
  use master
  go
  select count(*)
  from sysdatabases
  go
  ```

- You **cannot** combine the following database commands with other statements in a batch:
  - **create procedure**
  - **create rule**
  - **create default**
  - **create trigger**
- You **can** combine the following database commands with other Transact-SQL statements in a batch:
  - **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**
- **create view**

- You cannot bind rules and defaults to columns and use them in the same batch. **sp_bindrule** and **sp_bindefault** cannot be in the same batch as **insert** statements that invoke the rule or default.
- You cannot **drop** an object and then reference or re-create it in the same batch.
- If a table already exists, you cannot re-create it in a batch, even if you include a test in the batch for the table's existence.

  Adaptive Server compiles a batch before executing it. During compilation, Adaptive Server makes no permission checks on objects, such as tables and views, that are referenced by the batch. Permission checks occur when Adaptive Server executes the batch. An exception to this is when Adaptive Server accesses a database other than the current one. In this case, Adaptive Server displays an error message at compilation time without executing any statements in the batch.
- Assume that your batch contains these statements:

  ```
  select * from taba
  select * from tabb
  select * from tabc
  select * from tabd
  ```

  If you have the necessary permissions for all statements except the third one (**select * from tabc**), Adaptive Server returns an error message for that statement and returns results for all the others.

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 is 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)
```

You can combine a **use** statement with other statements, as long as the 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. The batch begins by making the *master* database current; afterwards, *pubs2* is the current database.

```
use master
go
select count(*) from sysdatabases
use pubs2
go

-------------
           6

(1 row affected)
```

You can combine a **drop** statement with other statements as long as you do not 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:
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 procedure testproc as
    select column1 from test
go

Msg 111, Level 15, State 7:
Line 6:
CREATE PROCEDURE must be the first command in a
query batch.

create default phonedflt as "UNKNOWN"
sp_bindefault phonedflt, "authors.phone"
go

Msg 102, Level 15, State 1:
Procedure 'phonedflt', Line 2:
Incorrect syntax near 'sp_bindefault'.
```

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:
titles not found. Specify owner.objectname or use
sp_help to check whether the object exists
(sp_help may produce lots of output)

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 Transact-SQL statements to **isql** from an operating system file. A file can include more than one batch, that is, 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 hello
   (column1 char(10), column2 int)
insert hello
   values ("hello", 598)
select * from hello
go
```

Here are the results of submitting this file to the **isql** utility:

```
-------------
           18

(1 row affected)
-------------
           23

(1 row affected)
column1      column2
---------    ---------
hello              598

(1 row affected)
```

See **isql** in the *Utility Programs* manual for your platform for environment-specific information on running batches stored in files.

Using Control-of-Flow Language

Use control-of-flow language 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.
case	Defines conditional expressions using **when...then** statements instead of **if...else**.
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* ***/** or **--***comment*	Insert a comment anywhere in a Transact-SQL statement.

if...else

The keyword **if**, with or without its companion **else**, introduces a condition that determines whether the next statement is executed. The Transact-SQL statement executes if the condition is satisfied, that is, if it returns TRUE.

The **else** keyword introduces an alternate Transact-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 is 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 is an example, using both **if** and **else**, that tests for the presence of user-created objects, all of which have ID numbers that are greater 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       16003088
 publishers    U       48003202
 roysched      U       80003316
 sales         U      112003430
 salesdetail   U      144003544
 titleauthor   U      176003658
 titles        U      208003772
 stores        U      240003886
 discounts     U      272004000
 au_pix        U      304004114
 blurbs        U      336004228
 friends_etc   U      704005539
 test          U      912006280
 hello         U     1056006793

(14 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, depending on the complexity of the **select** statements (or other language constructs) you include with each **if...else** construct.

case Expression

case expression simplifies many conditional Transact-SQL constructs. Instead of using a series of **if** statements, **case** expression allows you to use a series of conditions that return the appropriate values when the conditions are met. **case** expression is ANSI SQL92 compliant.

With **case** expression, you can:

- Simplify queries and write more efficient code
- Convert data between the formats used in the database (such as *int*) and the format used in an application (such as *char*)
- Return the first non-null value in a list of columns
- Compare two values and return the first value if the values do not match, or a NULL value if the values do match
- Write queries that avoid division by 0

case expression includes the keywords **case**, **when**, **then**, **coalesce**, and **nullif**. **coalesce** and **nullif** are an abbreviated form of **case** expression. For details on **case** expression syntax, see the *Adaptive Server Reference Manual*.

Using *case* Expression for Alternative Representation

Using **case** expression you can represent data in a manner that is more meaningful to the user. For example, the *pubs2* database stores a 1 or a 0 in the *contract* column of the *titles* table to indicate the status of the book's contract. However, in your application code or for user interaction, you may prefer to use the words "Contract" or "No Contract" to indicate the status of the book. To select the type from the *titles* table using the alternative representation:

```
select title, "Contract Status" =
    case
        when type = 1 then "Contract"
        when type = 0 then "No Contract"
    end
from titles
```

```
title                                    Contract Status
-----                                    ---------------
The Busy Executive's Database Guide      Contract
Cooking with Computers: Surreptitio      Contract
You Can Combat Computer Stress!          Contract
. . .
The Psychology of Computer Cooking       No Contract
. . .
Fifty Years in Buckingham Palace         Contract
Sushi, Anyone?                           Contract

(18 rows affected)
```

case and Division by Zero

case expression allows you to write queries that avoid division by zero (called *exception avoidance*). For example, if you are dividing the *total_sales* column for each book by the *advance* column, the query would result in division by zero:

```
select title_id, total_sales, advance,
total_sales/advance from titles
```

```
title_id   total_sales          advance
-------    -----------          ---------     ------
BU1032     4095                  5,000.00     0.82
BU1111     3876                  5,000.00     0.78
BU2075     18722                10,125.00     1.85
BU7832     4095                  5,000.00     0.82

Divide by zero occurred.
```

Division by zero results when the query attempts to divide the *total_sales* (2032) of *title_id* MC2222 by the *advance* (0.00).

case expression avoids this by not allowing the zero to figure in the equation. Instead, when the query comes across the zero, it returns a predefined value, rather than performing the division. For example:

```
select title_id, total_sales, advance, "Cost Per Book" =
     case
           when advance != 0
           then convert(char, total_sales/advance)
           else "No Books Sold"
     end
from titles
```

```
title_id       total_sales     advance          Cost Per Book
--------       -----------     ----------       -------------
BU1032                4095       5,000.00                0.82
BU1111                3876       5,000.00                0.78
BU2075               18722      10,125.00                1.85
BU7832                4095       5,000.00                0.82
MC2222                2032           0.00       No Books Sold
MC3021               22246      15,000.00                1.48
MC3026                NULL           NULL       No Books Sold
. . .
TC3218                 375       7,000.00                0.05
TC4203               15096       4,000.00                3.77
TC7777                4095       8,000.00                0.51

(18 rows affected)
```

The division by zero for *title_id* MC2222 no longer prevents the query from running. Also, the null values for MC3021 do not prevent the query from running.

case Expression Results

The rules for determining the datatype of a **case** expression are based on the same rules that determine the datatype of a column in a **union** operation. A **case** expression has a series of alternative result expressions (*R1*, *R2*, ..., *Rn* in the example below) which are specified by the **then** and **else** clauses. For example:

```
case
      when search_condition1 then R1
      when search_condition2 then R2
      ...
      else Rn
   end
```

The datatypes of the result expressions *R1*, *R2*, ..., *Rn* are used to determine the overall datatype of **case**. The same rules that determine the datatype of a column of a **union** that specifies *n* tables, and has the expressions *R1*, *R2*, …, *Rn* as the *i*th column, also determine the datatype of a **case** expression. The datatype of **case** is determined in the same manner as would be determined by the following query:

```
select...R1...from ...
union
select...R2...from...
union...
...
select...Rn...from...
```

Not all datatypes are compatible, and if you specify two datatypes that are incompatible (for example, *char* and *int*), your Transact-SQL query will fail. For more information about the union-datatype rules, see the *Adaptive Server Reference Manual*.

case Expression Requires At Least One Non-NULL Result

At least one result from the **case** expression must return a value other than the keyword NULL. The following query:

```
select price,
    case
        when title_id like "%" then NULL
        when pub_id like "%" then NULL
    end
from titles
```

returns the error message:

```
All result expressions in a CASE expression must
not be NULL
```

case

Using **case** expression, you can test for conditions that determine the result set.

The syntax is:

```
case
      when search_condition1 then result1
      when search_condition2 then result2
      . . .
      when search_conditionn then resultn
      else resultx
end
```

where *search_condition* is a logical expression, and *result* is an expression.

If search_*condition1* is true, the value of **case** is *result1*; if *search_condition1* is not true, *boolean_condition2* is checked. If *search_condition2* is true, the value of **case** is *result2*, and so on. If none of the search conditions are true, the value of **case** is *resultx*. The **else** clause is optional. If it is not used, the default is **else NULL**. **end** indicates the end of the **case** expression.

At least one result must be non-null. All the result expressions must be compatible. For information about determining the datatype of **case**, see "case Expression Results" on page 13-12.

The total sales of each book for each store are kept in the *salesdetail* table. If you want to show a series of ranges for the book sales , you could track how each book sold at each store, using the following ranges:

- Books that sold less than 1000 (low-selling books)
- Books that sold between 1000 and 3000 (medium-selling books)
- Books that sold more than 3000 (high-selling books)

You would write the following query:

```
select stor_id, title_id, qty, "Book Sales Catagory" =
     case
           when qty < 1000
              then "Low Sales Book"
           when qty >= 1000 and qty <= 3000
              then "Medium Sales Book"
           when qty > 3000
           then "High Sales Book"
     end
from salesdetail
group by title_id
```

```
stor_id     title_id      qty      Book Sales Catagory
-------     --------      ----     ------------------
5023        BU1032         200     Low Sales Book
5023        BU1032        1000     Low Sales Book
7131        BU1032         200     Low Sales Book
. . .
7896        TC7777          75     Low Sales Book
7131        TC7777          80     Low Sales Book
5023        TC7777        1000     Low Sales Book
7066        TC7777         350     Low Sales Book
5023        TC7777        1500     Medium Sales Book
5023        TC7777        1090     Medium Sales Book

(116 rows affected)
```

The following example selects the *titles* from the *titleauthor* table according to the author's royalty percentage (*royaltyer*) and then assigns each title with a value of High, Medium, or Low royalty:

```
select title, royaltyper, "Royalty Category" =
   case
      when (select avg(royaltyper) from titleauthor tta
        where t.title_id = tta.title_id) > 60 then "High Royalty"
      when (select avg(royaltyper) from titleauthor tta
        where t.title_id = tta.title_id) between 41 and 59
      then "Medium Royalty"
      else "Low Royalty"
   end
from titles t, titleauthor ta
where ta.title_id = t.title_id
order by title
```

```
title                                  royaltyper   royalty Category
-------                                ----------   ----------------
But Is It User Friendly?               100          High Royalty
Computer Phobic and Non-Phobic Ind     25           Medium Royalty
Computer Phobic and Non-Phobic Ind     75           Medium Royalty
Cooking with Computers: Surreptiti     40           Medium Royalty
Cooking with Computers: Surreptiti     60           Medium Royalty
Emotional Security: A New Algorith     100          High Royalty
. . .
Sushi, Anyone?                         40           Low Royalty
The Busy Executive's Database Guide    40           Medium Royalty
The Busy Executive's Database Guide    60           Medium Royalty
The Gourmet Microwave                  75           Medium Royalty
You Can Combat Computer Stress!        100          High Royalty

(25 rows affected)
```

case and Value Comparisons

This form of **case** is used for value comparisons. It allows only an equality check between two values; no other comparisons are allowed (for example, you cannot use "<", ">", "!=" or any other comparison operator).

The syntax is:

```
case valueT
      when value1 then result1
      when value2 then result2
      . . .
      when valuen then resultn
      else resultx
   end
```

where *value* and *result* are expressions.

If *valueT* equals *value1*, the value of the **case** is *result1*. If *valuetT* does not equal value1, *valueT* is compared to *value2*. If *valueT* equals

value2, then the value of the **case** is *result2*, and so on. If *valueT* does not equal the value of *value1* through *valuen*, the value of the **case** is *resultx*.

At least one result must be non-null. All the result expressions must be compatible. Also, all *values* must be compatible. For information about determining the datatype of **case**, see "case Expression Results" on page 13-12.

The syntax described above is equivalent to:

```
case
       when valueT = value1 then result1
       when valueT = value2 then result2
       . . .
       when valueT = valuen then resultn
       else resultx
   end
```

which is the same format used for **case** and search conditions (see "case" on page 13-13 for more information about this syntax).

The following example selects the *title* and *pub_id* from the *titles* table and specifies the publisher for each book based on the *pub_id*:

```
select title, pub_id, "Publisher" =
   case pub_id
     when "0736" then "New Age Books"
     when "0877" then "Binnet & Hardley"
     when "1389" then "Algodata Infosystems"
     else "Other Publisher"
   end
from titles
order by pub_id
```

```
title                          pub_id      Publisher
-----                          ------      -------------
Life Without Fear                0736      New Age Books
Is Anger the Enemy?              0736      New Age Books
You Can Combat Computer          0736      New Age Books
. . .
Straight Talk About Computers    1389      Algodata Infosystems
The Busy Executive's Database    1389      Algodata Infosystems
Cooking with Computers: Surre    1389      Algodata Infosystems

(18 rows affected)
```

Which is equivalent to the following query, which uses a **case** and search condition syntax:

```
select title, pub_id, "Publisher" =
   case
     when pub_id = "0736" then "New Age Books"
     when pub_id = "0877" then "Binnet & Hardley"
     when pub_id = "1389" then "Algodata Infosystems"
     else "Other Publisher"
   end
from titles
order by pub_id
```

coalesce

coalesce examines a series of values (*value1*, *value2*, ..., *valuen*) and returns the first non-null value. The syntax of **coalesce** is:

coalesce(*value1*, *value2*, ..., *valuen*)

Where *value1*, *value2*, ..., *valuen* are expressions. If *value1* is non-null, the value of **coalesce** is *value1*; if *value1* is null, *value*2 is examined, and so on. The examination continues until a non-null value is found. The first non-null value becomes the value of **coalesce**.

At least one *value* must be non-null. All the *value* expressions must be compatible. For information about determining the datatype of **coalesce**, see "case Expression Results" on page 13-12.

When you use **coalesce**, Adaptive Server translates it internally to the following format:

```
case
    when value1 is not NULL then value1
    when value2 is not NULL then value2
    . . .
    when valuen-1 is not NULL then valuen-1
    else valuen
end
```

where *valuen-1* refers to the next to last value, before the final value, *valuen*.

The example below uses coalesce to determine whether a store orders a low quantity (more than 100 but less than 1000) or a high quantity of books (more than 1000):

```
select stor_id, discount, "Quantity" =
    coalesce(lowqty, highqty)
from discounts
```

```
stor_id discount    Quantity
------- --------   ---------   ----
NULL               10.500000   NULL
NULL                6.700000    100
NULL               10.000000   1001
8042                5.000000   NULL

(4 rows affected)
```

nullif

nullif compares two values; if the values are equal, **nullif** returns a null value. If the two values are not equal, **nullif** returns the value of the first value. This form is useful for finding any missing, unknown, or inapplicable information that is stored in an encoded form. For example, values that are unknown are sometimes historically stored as -1. Using **nullif**, you can replace the -1 values with null and get the null behavior defined by Transact SQL. The syntax is:

nullif(*value1, value2*)

If *value1* equals *value2*, **nullif** returns NULL. If *value1* does not equal *value2*, **nullif** returns *value1*. *value1* and *value2* are expressions, and their datatypes must be comparable

At least one result must be non-null. All the result expressions must be compatible. For information about determining the datatype of **nullif**, see “case Expression Results” on page 13-12.

When you use **nullif**, Adaptive Server translates it internally to the following format:

```
case
      when value1 = value2 then NULL
      else value1
end
```

For example, the *titles* table uses the value “UNDECIDED” to represent books whose type category is not yet determined. The following query performs a search on the *titles* table for book types; any book whose type is “UNDECIDED” is returned as type NULL (the following output is reformatted for display purposes):

```
select title, "type"=
   nullif(type, "UNDECIDED")
from titles
```

```
title                                    type
-----                                    --------
The Busy Executive's Database Guide      business
Cooking with Computers: Surreptiti       business
You Can Combat Computer Stress!          business
. . .
The Psychology of Computer Cooking       NULL
Fifty Years in Buckingham Palace K       trad_cook
Sushi, Anyone?                           trad_cook

(18 rows affected)
```

Notice that *The Psychology of Computing* is stored in the table as "UNDECIDED", but the query returns it as type NULL.

begin...end

The **begin** and **end** keywords 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 is 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 applies only to the first Transact-SQL statement. The second statement executes independently of the first.

begin...end blocks can nest within other **begin...end** blocks.

while and *break...continue*

while sets 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, all the prices are 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 more than $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

Local variables are declared, named, and typed with the **declare** keyword and are assigned an initial value with a **select** statement. They must be declared, assigned a value, and used within the same batch or procedure.

See “Local Variables” on page 13-31 for more information.

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]
```

Here 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:

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 Adaptive Server. 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 (""), Adaptive 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, Adaptive 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 *Adaptive 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.

Creating 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
```

You can also bind user-defined messages to constraints, as described in "Creating Error Messages for Constraints" on page 7-34.

To drop a user-defined message, use the system procedure **sp_dropmessage**. To change a message, drop it with **sp_dropmessage** and add it again with **sp_addmessage**.

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}
```

where **delay** *time* instructs Adaptive Server to wait until the specified period of time has passed. **time** *time* instructs Adaptive 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, the following command instructs Adaptive Server to wait until 4:23 p.m.:

```
waitfor time "16:23"
```

This command instructs Adaptive Server to wait 1 hour and 30 minutes:

```
waitfor delay "01:30"
```

For a review of the acceptable formats for **time** values, see "Entering Times" on page 8-5.

errorexit instructs Adaptive 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**.

The following example instructs Adaptive 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.

```
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 Adaptive Server until the time or event that you specified occurs.

Comments

The comment notation attaches comments to statements, batches, and stored procedures. Comments are not executed.

A comment can be inserted on a line by itself or at the end of a command line. Two comment styles are available: the "slash-asterisk" style:

```
/* text of comment */
```

and the "double-hyphen" style:

```
-- text of comment
```

There is no maximum length for comments.

Slash-Asterisk Style Comments

The /* style comment is a Transact-SQL extension. Multiple-line comments are acceptable, as long as each comment starts with "/*" and ends with "*/". Everything between "/*" and "*/" is treated as part of the comment. The /* form permits nesting.

A stylistic convention often used for multiple-line comments is to begin the first line with "/*" and subsequent lines with "**". The comment is ended with "*/" as usual:

```
select * from titles
/* A comment here might explain the rules
** associated with using an asterisk as
** shorthand in the select list.*/
where price > $5
```

This procedure includes a couple of comments:

```
/* this procedure finds rules by user name*/
create procedure findmyrule @nm varchar(30) = null
as
if @nm is null
begin
  print "You must give a user name"
  return
  print "I have returned"
/* this statement follows return,
** so won't be executed */
end
else      /* print the rule names and IDs, and
          the user ID */
  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"
```

Double-Hyphen Style Comments

This comment style begins with two consecutive hyphens followed by a space (--) and terminates with a newline character. Therefore, multiple-line comments are not possible.

Adaptive Server does not interpret two consecutive hyphens within a string literal or within a /*-style comment as signaling the beginning of a comment.

To represent an expression that contains two consecutive minus signs (binary followed by unary), put a space or an opening parenthesis between the two hyphens.

Following are examples:

```
-- this procedure finds rules by user name
create procedure findmyrule @nm varchar(30) = null
as
if @nm is null
begin
  print "You must give a user name"
  return
  print "I have returned"
-- each line of a multiple-line comment
-- must be marked separately.
end
else        -- print the rule names and IDs, and
        -- the user ID
  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"
```

Local Variables

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, non-interactive use by the procedure when it executes. You can use variables nearly anywhere the Transact-SQL syntax indicates that an expression can be used, such as *char_expr, integer_expression, numeric_expr* or *float_expr.*

Declaring Local Variables

Use the following syntax to **declare** a local variable's name and datatype:

```
declare @variable_name datatype
      [, @variable_name datatype]...
```

The variable name must be preceded by the @ sign and conform to the rules for identifiers. Specify either a user-defined datatype or a system-supplied datatype other than *text*, *image*, or *sysname.*

It is more efficient in terms of memory and performance to write:

```
declare @a int, @b char(20), @c float
```

than to write:

```
declare @a int
declare @b char(20)
declare @c float
```

Local Variables and *select* Statements

When you declare a variable, it has the value NULL. Assign values 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]
```

Similar to **declare** statements, it is more efficient to write:

```
select @a = 1, @b = 2, @c = 3
```

than to write:

```
select @a = 1
select @b = 2
select @c = 3
```

Do not use a single **select** statement to assign a value to one variable and then to another whose value is based on the first. Doing so can yield unpredictable results. For example, the following queries both try to find the value of *@c2*. The first query yields NULL, while the second query yields the correct answer, 0.033333:

```
/* this is wrong*/
declare @c1 float, @c2 float
select @c1 = 1000/1000, @c2 = @c1/30
select @c1, @c2

/* do it this way */
declare @c1 float, @c2 float
select @c1 = 1000/1000
select @c2 = @c1/30
select @c1 , @c2
```

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)
```

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**.

If you are using variables in an **update** statement, see "Using the set Clause with update" on page 8-26.

Local Variables and *update* Statements

You can assign variables directly in an **update** statement. You do not need to use a **select** statement to assign a value to a variable. When you declare a variable, it has the value NULL. Here is the syntax:

```
update [[database.]owner.]{table_name | view_name}
  set [[[database.]owner.]{table_name. | view_name.}]
      column_name1 =
          {expression1 | null | (select_statement)} |
      variable_name1 = {expression1 | null}
      [, column_name2 = {expression2 | null |
          (select_statement)}]... |
      [, variable_name2 = {expression2 | null}]...
  [from [[database.]owner.]{table_name | view_name}
          [, [[database.]owner.]{table_name |
          view_name}]]...
  [where search_conditions]
```

Local Variables and Subqueries

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
```

Local Variables and *while* Loops and *if...else* Blocks

The following example uses local variables in a counter in a **while** loop, for doing matching in a **where** clause, in an **if** statement, and for setting and resetting values in **select** statements:

```
/* Determine if a given au_id has a row in au_pix*/
/* Turn off result counting */
set nocount on
/* declare the variables */
declare @c int,
        @min_id varchar(30)
/*First, count the rows*/
select @c = count(*) from authors
/* Initialize @min_id to "" */
select @min_id = ""
/* while loop executes once for each authors row */
while @c > 0
begin
        /*Find the smallest au_id*/
        select @min_id = min(au_id)
            from authors
            where au_id > @min_id
        /*Is there a match in au_pix?*/
        if exists (select au_id
            from au_pix
            where au_id = @min_id)
            begin
            print "A Match! %1!", @min_id
        end
      select @c = @c -1  /*decrement the counter */
end
```

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 the comparison between null-value variables and other null values follow special rules.

Table 13-2 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

Type of Comparison	Using the = Operator	Using the <, >, <=, !=, !<, !>, or <> Operator
Comparing *column_value* to *column_value*	FALSE	FALSE

Table 13-2: Comparing null values (continued)

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

(2 rows affected)
```

Global Variables

Global variables are system-supplied, predefined variables. They are distinguished from local variables by the two @ signs preceding their names—for example, *@@error*. The two @ signs are considered part of the identifier used to define the global variable.

Users cannot create global variables and cannot update the value of global variables directly in a **select** statement. If a user declares a local variable that has the same name as a **global variable**, that variable is treated as a local variable.

Transactions and Global Variables

Some global variables provide information to use in transactions.

Checking for Errors with @@*error*

The *@@error* global variable is commonly used to check the error status of the most recently executed batch in the current user session. *@@error* contains 0 if the last transaction succeeded; otherwise *@@error* contains the last error number generated by the system. A statement such as **if @@error != 0** followed by **return** causes an exit on error.

Every Transact-SQL statement, including **print** statements and **if** tests, resets *@@error*, so the status check must immediately follow the batch whose success is in question.

The *@@sqlstatus* global variable has no effect on *@@error* output. See "Checking the Status from the Last fetch" on page 13-38 for details.

Checking IDENTITY Values with @@*identity*

@@identity contains the last value inserted into an IDENTITY column in the current user session. *@@identity* is set each time an **insert**, **select into**, or **bcp** attempts to insert a row into a table. (The value of *@@identity* is not affected by the failure of an **insert**, **select into**, or **bcp** 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.)

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.

Checking the Transaction Nesting Level with @@*trancount*

@@trancount contains the nesting level of transactions in the current user session. Each **begin transaction** in a batch increments the transaction count. When you query *@@trancount* in chained transaction mode, its value is never 0 because the query automatically initiates a transaction.

Checking the Transaction State with @@*transtate*

@@transtate contains the current state of a transaction after a statement executes in the current user session. However, unlike *@@error*, *@@transtate* does not get cleared for each batch. *@@transtate* may contain the values in Table 13-3:

Table 13-3: @@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.

@@transtate only changes due to execution errors. Syntax and compile errors do not affect the value of *@@transtate*. See “Checking the State of Transactions” on page 18-9 for examples showing @@transtate in transactions.

Checking the Nesting Level with @@*nestlevel*

@@nestlevel contains the nesting level of current execution with the user session, initially 0. 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.

Checking the Status from the Last *fetch*

@@sqlstatus contains status information resulting from the last **fetch** statement for the current user session. *@@sqlstatus* may contain the following values:

Table 13-4: @@sqlstatus values

Value	Meaning
0	The **fetch** statement completed successfully.

Table 13-4: @@sqlstatus values (continued)

Value	Meaning
1	The **fetch** statement resulted in an error.
2	There is no more data in the result set. This warning occurs if the current cursor position is on the last row in the result set and the client submits a **fetch** command for that cursor.

@@sqlstatus has no effect on *@@error* output. For example, the following batch sets *@@sqlstatus* to 1 by causing the **fetch** statement to result in an error. However, *@@error* reflects the number of the error message, not the *@@sqlstatus* output:

```
declare csr1 cursor
for select * from sysmessages
for read only

open csr1

begin
    declare @xyz varchar(255)
    fetch csr1 into @xyz
    select error = @@error
    select sqlstatus = @@sqlstatus
end

Msg 553, Level 16, State 1:
Line 3:
The number of parameters/variables in the FETCH
INTO clause does not match the number of columns
in cursor 'csr1' result set.
```

At this point, the *@@error* global variable is set to 553, the number of the last generated error. *@@sqlstatus* is set to 1.

Global Variables Affected by *set* Options

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

Table 13-5 lists the global variables that contain information about the session options controlled by the **set** command.

Table 13-5: Global variables containing session options

Global Variable	Description
@@char_convert	Contains 0 if character set conversion not in effect. Contains 1 if character set conversion is in effect.
@@isolation	Contains the current isolation level of the Transact-SQL program. *@@isolation* takes the value of the active level (0, 1 or 3).
@@options	Contains a hexadecimal representation of the session's **set** options.
@parallel_degree	Contains the current maximum parallel degree setting.
@@rowcount	Contains the number of rows affected by the last query. *@@rowcount* is set to 0 by any command that does not return rows, such as an **if** statement. With cursors, *@@rowcount* represents the cumulative number of rows returned from the cursor result set to the client, up to the last **fetch** request.
@scan_parallel_degree	Contains the current maximum parallel degree setting for nonclustered index scans.
@@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**.
@@tranchained	Contains the current transaction mode of the Transact-SQL program. *@@tranchained* returns 0 for unchained or 1 for chained.

The *@@options* global variable contains a hexadecimal representation of the session's **set** options.Table 13-6 lists **set** options and values that work with *@@options*.

Table 13-6: set options and values for @@options

Numeric Value	Hexidecimal Value	*set* Option
4	0x04	**showplan**
5	0x05	**noexec**
6	0x06	**arithignore**

Table 13-6: set options and values for @@options (continued)

Numeric Value	Hexidecimal Value	*set* Option
8	0x08	**arithabort**
13	0x0D	**control**
14	0x0E	**offsets**
15	0x0F	**statistics io** and **statistics time**
16	0x10	**parseonly**
18	0x12	**procid**
20	0x14	**rowcount**
23	0x17	**nocount**

For more information on session options, see the **set** command in the *Adaptive Server Reference Manual*.

Language and Character Set Information in Global Variables

Table 13-7 lists the global variables that contain information about languages and character sets. For more information on languages and character sets, see the *System Administration Guide*.

Table 13-7: Global variables for language and character sets

Global Variable	Description
@@char_convert	Contains 0 if character set conversion not in effect. Contains 1 if character set conversion is in effect.
@@client_csid	Contains client's character set ID. Set to -1 if the client character set has never been initialized; otherwise, it contains the most recently used client character set's *id* from *syscharsets*.
@@client_csname	Contains client's character set name. Set to NULL if the client character set has never been initialized; otherwise, it contains the name of the most recently used character set.
@@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*.

Table 13-7: Global variables for language and character sets (continued)

Global Variable	Description
@@maxcharlen	Contains the maximum length, in bytes, of multibyte characters in the default character set.
@@ncharsize	Contains the average length, in bytes, of a national character.
@@char_convert	Contains 0 if character set conversion not in effect. Contains 1 if character set conversion is in effect.

Global Variables for Monitoring System Activity

Many global variables report on system activity occurring from the last time Adaptive Server was started. The system procedure **sp_monitor** displays the current values of some of the global variables.

Table 13-8 lists the global variables that monitor system activity, in the order returned by **sp_monitor**. For complete information on **sp_monitor**, see the *Adaptive Server Reference Manual*.

Table 13-8: Global variables that monitor system activity

Global Variable	Description
@@connections	Contains the number of logins or attempted logins.
@@cpu_busy	Contains the amount of time, in ticks, that the CPU has spent doing Adaptive Server work since the last time Adaptive Server was started.
@@idle	Contains the amount of time, in ticks, that Adaptive Server has been idle since it was last started.
@@io_busy	Contains the amount of time, in ticks, that Adaptive Server has spent doing input and output operations.
@@packet_errors	Contains the number of errors that occurred while Adaptive Server was sending and receiving packets.
@@pack_received	Contains the number of input packets read by Adaptive Server since it was last started.
@@pack_sent	Contains the number of output packets written by Adaptive Server since it was last started.
@@total_errors	Contains the number of errors that occurred while Adaptive Server was reading or writing.
@@total_read	Contains the number of disk reads by Adaptive Server since it was last started.

Table 13-8: Global variables that monitor system activity (continued)

Global Variable	Description
@@total_write	Contains the number of disk writes by Adaptive Server since it was last started.

Server Information Stored in Global Variables

Table 13-9 lists the global variables that contain miscellaneous information about Adaptive Server.

Table 13-9: Global variables containing Adaptive Server information

Global Variable	Description
@@cis_version	Contains the date and version of Component Integration Services, if it is installed.
@@max_connections	Contains the maximum number of simultaneous connections that can be made with Adaptive Server in the current computer environment. You can configure Adaptive Server for any number of connections less than or equal to the value of *@@max_connection*s with the **number of user connections** configuration parameter.
@@procid	Contains the stored procedure ID of the currently executing procedure.
@@servername	Contains the name of this Adaptive Server. Define a server name with **sp_addserver**, and then restart Adaptive Server.
@@spid	Contains the server process ID number of the current process.
@@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.
@@version	Contains the date of the current release of Adaptive Server.

Global Variables and *text* and *image* Data

text and *image* values can be quite large. 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, which contains the limit on the number of bytes of *text* or *image* data a **select** returns. The default limit is 32K bytes for **isql**; the default depends on the client software. Change the value for a session with **set textsize**.

Table 13-10 lists the global variables that contain information about text pointers. Note that these global variables are session-based. For more information about text pointers, see "Changing text and image Data" on page 8-29.

Table 13-10: Text pointer information stored in global variables

Global Variable	Description
@@*textcolid*	Contains the column ID of the column referenced by @@*textptr*.
@@*textdbid*	Contains the database ID of a database containing an object with the column referenced by @@*textptr*.
@@*textobjid*	Contains the object ID of an object containing the column referenced by @@*textptr*.
@@*textpt*	Contains the text pointer of the last *text* or *image* column inserted or updated by a process. (Do not confuse this variable with the **textptr** function.)
@@*textts*	Contains the text timestamp of the column referenced by @@*textptr*.

14 Using Stored Procedures

A **stored procedure** is a named collection of SQL statements or control-of-flow language. You can create stored procedures for commonly used functions and to increase performance. Adaptive Server also provides **system procedures** to perform administrative tasks and to update the system tables.

This chapter discusses:

You can also create and use extended stored procedures to call procedural language functions from Adaptive Server. See Chapter 15, “Using Extended Stored Procedures.”

How Stored Procedures Work

When you run a stored procedure, Adaptive Server prepares an execution plan so that the procedure’s 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 Adaptive 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 Adaptive Servers can be executed if both your server and the remote server are set up to allow remote logins. You can write triggers on your local Adaptive 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, Adaptive 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 instantly.

Adaptive Server supplies a variety of stored procedures as convenient tools for the user. The procedures stored in the *sybsystemprocs* database whose names begin with "sp_" are known as **system procedure**s, because they insert, update, delete and report on data in the system tables.

You create a stored procedure 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 in 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 except for **create** statements. See "Restrictions Associated with Stored Procedures" on page 14-26. 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 use the procedure's name, as long as it is submitted to Adaptive 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 Adaptive 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 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. For information on setting up remote procedure calls on Adaptive Server, see the *System Administration Guide.*

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 **source text** is stored in *syscomments*.

You can display the source 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 the tables or views referenced in it or permission to execute specific commands. For details, see the *Security Features User's Guide*.

You can protect the source text of a stored procedure against unauthorized access by restricting **select** permission on the *text*

column of the *syscomments* table to the creator of the procedure and the System Administrator. This restriction is required to run Adaptive Server in the **evaluated configuration**. To enact this restriction, a System Security Officer must reset the **allow select on syscomments.text column** parameter with the system procedure **sp_configure**. For more information, see the *System Administration Guide*.

Another way to protect access to the source text of a stored procedure is to hide the source text using the system procedure **sp_hidetext**. For information about hiding source text, see **sp_hidetext** in the *Adaptive Server Reference Manual*.

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 *Adaptive Server Reference Manual*.

Creating and Executing Stored Procedures

The complete syntax for **create procedure** is:

```
create procedure [owner.]procedure_name[;number]
   [[(]@parameter_name
       datatype [(length) | (precision [, scale])]
       [= default][output]
   [, @parameter_name
       datatype [(length) | (precision [, scale])]
       [= default][output]]...[)]]
   [with recompile]
   as {SQL_statements | external name dll_name}
```

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**:

```
[exec[ute]] [@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*

When Component Integration Services is not enabled, remote procedure calls (RPCs) are not treated as part of a transaction. If you execute an RPC after **begin transaction**, and then execute **rollback transaction**, Adaptive Server does not roll back any changes made by the RPC on remote data. The stored procedure designer should be sure that all conditions that might trigger a rollback are checked before issuing an RPC that will alter remote data. If Component Integration Services is enabled, use the **set transactional rpc** and **cis rpc handling** commands to use transactional RPCs. For more information on the **transactional rpc** and **cis rpc handling** options, see the **set** command in the *Adaptive Server Reference Manual*.

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 an @ sign and must conform to the rules for identifiers, as discussed under “Identifiers” on page 1-7. Parameter names are local to the procedure that creates them; the same parameter names can be used in other procedures. Enclose any parameter value that includes punctuation (such as an object name qualified by a database name or owner name) in single or double quotes. Parameter names, including the @ sign, can be a maximum of 30 bytes long.

Parameters must be given a system datatype (except *text* or *image*) or a user-defined datatype, and (if required for the datatype) a length or precision and scale in parentheses.

Here is a stored procedure for the *pubs2* database. Given an author’s last and first names, the procedure displays the names of any books written by that person and the name of 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, 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 displays 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 as 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      0
titles      titleind        2

(2 rows affected, return status = 0)
```

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 supply all subsequent parameters this way.

The following procedure shows the datatype of any column. Here, the procedure displays the datatype of the *qty* column from the *salesdetail* table.

```
create procedure showtype @tabname varchar(18),
@colname varchar(18) as
  select syscolumns.name, syscolumns.length,
  systypes.name
  from syscolumns, systypes, sysobjects
  where sysobjects.id = syscolumns.id
  and @tabname = sysobjects.name
  and @colname = syscolumns.name
  and syscolumns.type = systypes.type
```

When the procedure is executed, the *@tabname* and *@colname* values can be given in a different order from the **create procedure** statement if they are specified by name:

```
exec showtype
@colname = qty , @tabname = salesdetail
```

You can use **case** expressions in any stored procedure where you use a value expression. The following example checks the sales for any book in the *titles* table:

```
create proc booksales @titleid tid
as
select title, total_sales,
case
when total_sales != null then "Books sold"
when total_sales = null then "Book sales not available"
end
from titles
where @titleid = title_id
```

For example:

```
booksales MC2222
```

```
title                                  total_sales
------------------------               -----------
Silicon Valley Gastronomic Treats    2032 Books sold

(1 row affected)
```

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 is a procedure that displays the names of all the authors who 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, Adaptive 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, Adaptive 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, Adaptive Server

displays an error message listing the parameters expected by the procedure.

If a user executes a stored procedure and specifies more parameters than the number of parameters expected by the procedure, Adaptive Server ignores the extra parameters.

NULL As the Default Parameter

In the **create procedure** statement, you can declare NULL as the default value for individual parameters:

```
create procedure procedure_name
   @param datatype [ = null ]
   [, @param datatype [ = null ]]...
```

In this case, if the user does not supply a parameter, Adaptive 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, Adaptive Server prints the message from the procedure on the screen.

For other examples of setting the default to NULL, examine the source 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 supply all subsequent parameters 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)
```

If a user executes a stored procedure and specifies more parameters than the number of parameters expected by the procedure, Adaptive Server ignores the extra parameters.

Remember that 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. If you issue a stored procedure followed by a command, Adaptive Server will attempt to execute the procedure and then the command. For example, if you issue the following commands:

```
sp_help checkpoint
```

Adaptive Server returns the output from **sp_help** and runs the **checkpoint** command. Using delimited identifiers for procedure parameters can produce unintended results.

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
```

To run Adaptive Server in the evaluated configuration, you must prohibit grouping of procedures. This prohibition ensures that every stored procedure has a unique object identifier and can be dropped individually. To disallow procedure grouping, a System Security Officer must reset the **allow procedure grouping** configuration parameter. For information, see the *System Administration Guide*.

Using *with recompile* in *create procedure*

In the **create procedure** statement, the optional clause **with recompile** comes just before the SQL statements. It instructs Adaptive 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**, Adaptive Server stores the execution plan that it created. Usually, this execution plan is satisfactory.

However, a change in the data or parameter values supplied for subsequent executions may cause Adaptive Server to create an execution plan that is different from the one it created when the procedure was first executed. In such situations, Adaptive Server needs a new execution plan.

Use **with recompile** in a **create procedure** statement when you think you need a new plan. See the *Adaptive Server Reference Manual* for more information.

Using *with recompile* in *execute*

In the **execute** statement, the optional clause **with recompile** comes after any parameters. It instructs Adaptive Server to compile a new plan, which 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.

You can call another procedure by name or by a variable name in place of the actual procedure name. For example:

```
create procedure test1 (@var char(10)
    as exec @au_info2
```

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, Adaptive 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)
   ```

   When you run the *inv_proc* procedure, it creates the table, but it only exists during the procedure's execution. Try inserting values into the *#tempstores* table or running the *inv_amounts* procedure:

   ```
   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 cannot, because the *#tempstores* table no longer exists.

   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.

Setting Options in Stored Procedures

You can use some of the **set** command options the inside a stored procedure. The **set** option remains in effect during the execution of the procedure and most options revert to the former setting at the close of the procedure. Only the **dateformat**, **datefirst**, and **language** options do not revert to their former settings.

However, if you use a **set** option (such as **identity_insert**) that requires the user to be the object owner, a user who is not the object owner cannot execute the stored procedure.

After Creating a Stored Procedure

After you create a stored procedure, the **source text** describing the procedure is stored in the *text* column of the *syscomments* system table. In previous releases of SQL Server, users often deleted the source text from *syscomments*, in order to save disk space and remove confidential information from this public area. **Do not remove this information from *syscomments***; doing so can cause problems for

future upgrades of Adaptive Server. Instead, encrypt the text in *syscomments* by using the **sp_hidetext** system procedure, described in the *Adaptive Server Reference Manual*. For more information, see "Compiled Objects" on page 1-3.

Executing Stored Procedures

You can execute stored procedures after a time delay or remotely.

Executing Procedures After a Time Delay

The **waitfor** command delays execution of a stored procedure at a specified time or until a specified amount of time has passed.

For example, to execute the procedure **testproc** in half an hour:

```
begin
    waitfor delay "0:30:00"
    exec testproc
end
```

After issuing the **waitfor** command, you cannot use that connection to Adaptive Server until the specified time or event occurs.

Executing Procedures Remotely

You can execute procedures on a remote Adaptive Server from your local Adaptive Server. Once both servers are properly configured, you can execute any procedure on the remote Adaptive 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 Adaptive 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 Adaptive Server appear on your local terminal.

The return status from procedures can be used to capture and transmit information messages about the execution status of your procedures. For more information, see "Return Status" on page 14-18.

◆ ***WARNING!***

If Component Integration Services is not enabled, Adaptive Server does not treat remote procedure calls (RPCs) as part of a transaction. Therefore, if you execute an RPC as part of a transaction, and then roll back the transaction, Adaptive Server does not roll back any changes made by the RPC. When Component Integration Services is enabled, use set transactional rpc and set cis rpc handling to use transactional RPCs. For more information on the transactional rpc and cis rpc handling options, see the set command in the *Adaptive Server Reference Manual*.

Returning Information from Stored Procedures

Stored procedures can return the following types of information:

- Return status – indicates whether or not the stored procedure completed successfully.
- **proc role** function – checks whether the procedure was executed by a user with **sa_role**, **sso_role**, or **ss_oper** privileges.
- Return parameters – report the parameter values back to the caller, who 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 that returns its execution status or the values of its parameters to the calling procedure. For example, many Adaptive Server system procedures execute a procedure that verifies that certain parameters are valid identifiers.

Remote procedure calls, which are stored procedures that run on a remote Adaptive Server, also return both kinds of information. All 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 report a **return status** that indicates whether or not they completed successfully, and if they did not, the reasons for failure. This value can be stored in a variable when a procedure is

called, and used in future Transact-SQL statements. Adaptive Server-defined return status values for failure range from -1 through -99; users can define their own return status values outside this range.

Here is an example of a batch that uses the form of the **execute** statement that returns the status:

```
declare @status int
execute @status = byroyalty 50
select @status
```

The execution status of the *byroyalty* procedure is stored in the variable *@status*. "50" is the supplied parameter, based on the *royaltyper* column of the *titleauthor* table. This example prints the value with a **select** statement; later examples use this return value in conditional clauses.

Reserved Return Status Values

Adaptive Server reserves 0, to indicate a successful return, and negative values from -1 through -99, to indicate the reasons for failure. Numbers 0 and -1 through -14 are currently used in version 12, as shown in Table 14-1:

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 through -99 are reserved for future use by Adaptive 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 0 through -99 are reserved for use by Adaptive 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
```

For example:

```
checkcontract MC2222

(return status = 1)
```

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 want 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 do not have the right role."
    return -1
end
else
    print "You have SA role."
    return 0
```

For example:

```
test_proc

You have SA role.
```

Return Parameters

Another way that stored procedures can return information to the caller is through **return parameters**. The caller can then use conditional statements to check the returned value.

When both a **create procedure** statement and an **execute** statement 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 would not see the results of the multiplication. The **select** statement in the procedure assigns values, but does not 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*. Adaptive 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 or not its value has changed. 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, *@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 bulleted items, above, 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 "Bingo!"
else
    print "Wrong, wrong, wrong!"
(1 row affected)
(1 row affected)
(return status = 0)

@result
-----------
         30

 Your_answer Right_answer
 ----------- ------------
          32           30

Wrong, wrong, wrong!
```

This stored procedure checks to determine 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)

@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

Use the following format to pass values in the parameters:

@parameter = @variable

You cannot pass constants; there must be a variable name to "receive" the return value. The parameters can be of any Adaptive Server datatype except *text* or *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 are 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.

Restrictions Associated with Stored Procedures

Some additional restrictions on creating stored procedures are as follows:

- You cannot combine **create procedure** statements with other statements in the same batch.
- The **create procedure** definition itself can include any number and kind of SQL statements, except for **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 you create it before you reference it. 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.
- Adaptive Server creates the 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 with the #prefix inside a procedure, the temporary table exists only for purposes of the procedure—it disappears when you exit the procedure. Temporary tables created using **create table** ***tempdb..tablename*** do not disappear unless you explicitly drop them.
- 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 **create table** and **dbcc** must be **qualified** with the object owner's name, if other users are to use the stored procedure. Object names used with other statements, like **select** and **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 **select** or **insert** if she wants other users to execute the procedure in which the table is used. 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, Adaptive Server looks for a table called *marytab* owned by John.

Renaming Stored Procedures

Use the system procedure **sp_rename** to rename stored procedures. Its syntax is:

```
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 works only until Adaptive 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 permissions on those tables.

For information about granting and revoking permissions of stored procedures and other database objects, see the *Security Features User's Guide*.

Dropping Stored Procedures

Use the **drop procedure** command to remove stored procedures. Its syntax is:

```
drop proc[edure] [owner.]procedure_name
   [, [owner.]procedure_name] ...
```

If a stored procedure that was dropped is called by another stored procedure, Adaptive 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 the original procedure can call it successfully.

A procedure group, that is, more than one procedure with the same name but with different numbered suffixes, can be dropped with a single **drop procedure** statement. Once procedures have been grouped, procedures within the group cannot be dropped individually.

System Procedures

System procedures are:

- 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 system procedures begin with "sp_". They are created by the **installmaster** script in the *sybsystemprocs* database during Adaptive Server installation.

Executing System Procedures

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*. To run a system procedure in a specific database, either open that database with the **use** command and execute the procedure, or qualify the procedure name with the database name.

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.

Permissions on System Procedures

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 ensure 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.

Types of System Procedures

System procedures can be grouped by what they are used for, such as auditing, security administration, data definition, and so on. The following sections list the types of system procedures.

For complete information about the system procedures, see the *Adaptive Server Reference Manual.*

System Procedures for Auditing

These system procedures are used for:

- Controlling audit settings
- Creating and managing the tables which hold the audit queue

The procedures in this category are:

sp_addauditrecord **sp_addaudittable** **sp_audit** **sp_displayaudit**

For more information on auditing, see the *Security Administration Guide.*

System Procedures Used for Security Administration

These system procedures are used for:

- Adding, dropping, and reporting on logins on Adaptive Server
- Adding, dropping, and reporting on users, groups, roles, and aliases in a database
- Changing passwords and default databases

- Adding, dropping, and reporting on remote servers that can access the current Adaptive Server
- Adding the names of users from remote servers who can access the current Adaptive Server

The procedures in this category are:

sp_activeroles	**sp_changegroup**	**sp_droplogin**	**sp_locklogin**
sp_addalias	**sp_displaylogin**	**sp_dropuser**	**sp_modifylogin**
sp_addgroup	**sp_displayroles**	**sp_helpgroup**	**sp_password**
sp_addlogin	**sp_dropalias**	**sp_helprotect**	**sp_role**
sp_adduser	**sp_dropgroup**	**sp_helpuser**	

Additional system procedures are available for Windows NT. The procedures in this category are:

sp_grantlogin	**sp_logininfo**	**sp_revokelogin**
sp_loginconfig	**sp_processmail**	

System Procedures Used for Remote Servers

These system procedures are used for:

- Adding, dropping and reporting on remote servers that can access the current Adaptive Server
- Adding the names of users from remote servers who can access the current Adaptive Server

The procedures in this category are:

sp_addremotelogin	**sp_dropserver**	**sp_remoteoption**
sp_addserver	**sp_helpremotelogin**	**sp_serveroption**
sp_dropremotelogin	**sp_helpserver**	

Additional system procedures are available if you have Component Integration Services installed. The procedures in this category are:

sp_addexternlogin	**sp_dropexternlogin**	**sp_passthru**
sp_addobjectdef	**sp_dropobjectdef**	**sp_remotesql**
sp_autoconnect	**sp_helpexternlogin**	
sp_defaultloc	**sp_helpobjectdef**	

System Procedures for Managing Databases

These system procedures are used for:

- Changing the owner of a database

- Displaying and changing database options

The procedures in this category are:

sp_changedbowner	**sp_dbremap**	**sp_renamedb**
sp_dboption	**sp_helpdb**	

System Procedures Used for Data Definition and Database Objects

These system procedures are used 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
- Binding and unbinding objects to data caches
- Reporting on database objects, user-defined datatypes, dependencies among database objects, databases, indexes, and space used by tables and indexes
- Getting help on command syntax

The procedures in this category are:

sp_addextendedproc	**sp_dropkey**	**sp_helptext**
sp_addtype	**sp_droptype**	**sp_hidetext**
sp_bindcache	**sp_estspace**	**sp_poolconfig**
sp_bindefault	**sp_foreignkey**	**sp_primarykey**
sp_bindrule	**sp_freedll**	**sp_procxmode**
sp_cacheconfig	**sp_help**	**sp_recompile**
sp_cachestrategy	**sp_helpartition**	**sp_rename**
sp_chgattribute	**sp_helpcache**	**sp_spaceused**
sp_checksource	**sp_helpconstraint**	**sp_unbindcache**
sp_commonkey	**sp_helpextendedproc**	**sp_unbindcache_all**
sp_cursorinfo	**sp_helpindex**	**sp_unbindefault**
sp_depends	**sp_helpjoins**	**sp_unbindrule**
sp_dropextendedproc	**sp_helpkey**	

System Procedures Used for User-Defined Messages

These system procedures are used 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_bindmsg**	**sp_getmessage**
sp_altermessage	**sp_dropmessage**	**sp_unbindmsg**

System Procedures for Languages

These system procedures are used for:

- Managing character sets
- Adding and dropping alternate languages

The procedures in this category are:

sp_addlanguage	**sp_helplanguage**	**sp_setlangalias**
sp_checknames	**sp_helpsort**	
sp_droplanguage	**sp_indsuspect**	

System Procedureds Used for Device Management

These system procedures are used for:

- Adding, dropping, and reporting on database and dump devices
- Adding, dropping, and extending database segments

The procedures in this category are:

sp_addsegment	**sp_dropsegment**	**sp_helpsegment**
sp_addumpdevice	**sp_extendsegment**	**sp_logdevice**
sp_diskdefault	**sp_helpdevice**	**sp_logiosize**
sp_dropdevice	**sp_helplog**	**sp_placeobject**

System Procedureds Used for Backup and Recovery

These system procedures are used for:

- Creating and managing thresholds

- Displaying and setting recovery isolation mode
- Communicating with Backup Server™

The procedures in this category are:

sp_addthreshold	**sp_helpthreshold**	**sp_setsuspect_granularity**
sp_dropthreshold	**sp_listsuspect_db**	**sp_setsuspect_threshold**
sp_forceonline_db	**sp_listsuspect_page**	**sp_thresholdaction**
sp_forceonline_page	**sp_modifythreshold**	**sp_volchanged**

System Procedureds Used for Configuration and Tuning

These system procedures are used for:

- Changing and reporting on configuration parameters
- Reporting on system usage
- Reporting performance behavior
- Monitoring Adaptive Server activity

The procedures in this category are:

sp_clearstats	**sp_displaylevel**	**sp_monitorconfig**
sp_configure	**sp_helpconfig**	**sp_reportstats**
		sp_showplan
sp_countmetadata	**sp_monitor**	**sp_sysmon**

System Procedures Used for System Administration

These system procedures are used for:

- Reporting on locks and the users currently running processes
- Altering the lock promotion thresholds of a table or database
- Creating and reporting on resource limits and time ranges
- Creating and reporting on execution classes and engine groups
- Planning the layout of the *dbccdb* database

The procedures in this category are:

sp_addengine	**sp_drop_resource_limit**	**sp_setpsexe**
sp_addexeclass	**sp_drop_time_range**	**sp_showcontrolinfo**
sp_add_resource_limit	**sp_familylock**	**sp_showexeclass**
sp_add_time_range	**sp_help_resource_limit**	**sp_showpsexe**
sp_bindexeclass	**sp_lock**	**sp_unbindexeclass**
sp_clearpsexe	**sp_modify_resource_limit**	**sp_setpsexe**
sp_dropexeclass	**sp_modify_time_range**	**sp_who**
sp_dropengine	**sp_setpglockpromote**	
sp_dropglockpromote	**sp_plan_dbccdb**	

See the *System Administration Guide* for more information about the system procedures that accomplish these administrative tasks.

System Procedures for Upgrade

These system procedures are used for:

- Checking for reserved words
- Remapping objects
- Checking the query processing mode of an object

The procedures in this category are:

sp_checkreswords	**sp_remap**	**sp_procqmode**

Other Sybase-supplied Procedures

Sybase provides catalog stored procedures, system extended stored procedures (system ESPs) and **dbcc** procedures.

Catalog Stored Procedures

Catalog stored procedures are system procedures which retrieve information from the system tables in tabular form. The catalog stored procedures are:

sp_column_privileges	**sp_pkeys**	**sp_stored_procedures**
sp_columns	**sp_server_info**	**sp_table_privileges**
sp_databases	**sp_special_columns**	**sp_tables**
sp_datatype_info	**sp_sproc_columns**	
sp_fkeys	**sp_statistics**	

System Extended Stored Procedures

Extended stored procedures (ESPs) call procedural language functions from Adaptive Server. The system extended stored procedures, created by **installmaster** at installation, are located in the *sybsystemprocs* database and are owned by the System Administrator. They can be run from any database and their names begin with "xp_". The system extended stored procedures are:

xp_cmdshell	**xp_findnextmsg**	**xp_sendmai**
xp_deletemail	**xp_logevent**	**xp_startmai**
xp_enumgroups	**xp_readmail**	**xp_stopmai**

See See Chapter 15, "Using Extended Stored Procedures," for more information on creating and using ESPs.

dbcc Procedures

The **dbcc** procedures, created by *installdbccdb*, are stored procedures for generating reports on information created by **dbcc checkstorage.** These procedures reside in the *dbccdb* database or in the alternate database, *dbccalt*. The **dbcc** procedures are:

sp_dbcc_alterws	**sp_dbcc_differentialreport**	**sp_dbcc_statisticsreport**
sp_dbcc_configreport	**sp_dbcc_evaluatedb**	**sp_dbcc_summaryreport**
sp_dbcc_createws	**sp_dbcc_faultreport**	**sp_dbcc_updateconfig**
sp_dbcc_deletedb	**sp_dbcc_fullreport**	
sp_dbcc_deletehistory	**sp_dbcc_runcheck**	

Getting Information About Stored Procedures

Several system procedures provide information from the system tables about stored procedures.

System procedures are briefly discussed in "System Procedures" on page 14-29. For complete information about system procedures, see the *Adaptive Server Reference Manual*.

Getting a Report with *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
--------   ------  ---------------
byroyalty  dbo     stored procedure

Data_located_on_segment         When_created
---------------------------     --------------------
not applicable                  Jul 10, 1997 4:57PM

Parameter_name Type    Length   Prec  Scale  Param_order
-------------- ------  ------   ----  -----  -----------
@percentage    int          4   NULL   NULL            1

(return status = 0)
```

You can get help on a system procedure by executing **sp_help** when using the *sybsystemprocs* database.

Viewing the Source Text of a Procedure with *sp_helptext*

To display the source 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 source text of a system procedure by executing **sp_helptext** when using the *sybsystemprocs* database.

If the source text of a stored procedure was encrypted using the system procedure **sp_hidetext**, Adaptive Server displays a message advising you that the text is hidden. For information about hiding source text, see **sp_hidetext** in the *Adaptive Server Reference Manual*.

Identifying Dependent Objects with *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.

For example, 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.byroyalty   stored procedure
dbo.titleview   view

(return status = 0)
```

You must drop and re-create the procedure if any of its referenced objects have been renamed.

Identifying Permissions with *sp_helprotect*

The system procedure **sp_helprotect** reports permissions on a stored procedure (or any other database object). For example:

```
sp_helprotect byroyalty

grantor    grantee    type  action    object    column  grantable
---------  ---------  ----  --------- -------   -----   ---------
dbo        public     Grant Execute   byroyalty All     FALSE

(return status = 0)
```

15 Using Extended Stored Procedures

Extended stored procedures (ESPs) provide a mechanism for calling external procedural language functions from within Adaptive Server. Users invoke ESPs using the same syntax as they use for stored procedures. The difference is that an ESP executes procedural language code rather than Transact-SQL statements.

This chapter discusses:

Why Use Extended Stored Procedures?

Extended stored procedures provide a way of dynamically loading and executing external procedural language functions from within Adaptive Server. Each ESP is associated with a corresponding function, which is executed when the ESP is invoked from Adaptive Server.

An ESP allows Adaptive Server to perform a task outside Adaptive Server in response to an event occurring within Adaptive Server. For example, you could create an ESP function to sell a security (a task performed outside Adaptive Server). This function is invoked in response to a trigger that is fired when the price of the security reaches a certain value. Or you could create an ESP function that sends an email notification or a network-wide broadcast in response to an event occurring within the relational database system.

For the purposes of ESPs, “a procedural language” is a programming language that is capable of calling a C language function and manipulating C language datatypes.

After a function has been registered in a database as an ESP, it can be invoked just like a stored procedure from **isql**, from a trigger, from another stored procedure, or from a client application.

ESPs can:

- Take input parameters
- Return a status value indicating success or failure and the reason for the failure
- Return values of output parameters
- Return result sets

Adaptive Server supplies some system ESPs as a convenience to users. For example, one system ESP, **xp_cmdshell**, executes an operating system command from within Adaptive Server. Customers can also write their own ESPs using a subset of the Open Server application programming interface (API).

ESP Overview

This section introduces you to ESPs. It covers:

- XP Server
- Dynamic Link Library Support
- Open Server API
- Example of Creating and Using ESPs
- ESPs and Permissions
- ESPs and Performance

Subsequent sections go into more detail about how to create and use ESPs.

XP Server

Extended stored procedures are implemented by an Open Server application called XP Server, which runs on the same machine as Adaptive Server. Adaptive Server and XP Server communicate through remote procedure calls (RPCs). Running ESPs in a separate process protects Adaptive Server from a failure resulting from faulty ESP code. The advantage of using ESPs instead of RPCs is that the ESP runs in Adaptive Server the same way a stored procedure runs; you do not need to have Open Server just to run the ESP itself.

XP Server is automatically installed when Adaptive Server is installed.

XP Server must be running for Adaptive Server to execute an ESP. Adaptive Server starts XP Server the first time an ESP is invoked and shuts down XP Server when Adaptive Server itself exits.

On Windows NT, if the **start mail session** configuration parameter is set to 1, XP Server automatically starts when Adaptive Server starts.

Dynamic Link Library Support

The procedural functions that contain the ESP code are compiled and linked into dynamic link libraries (DLLs), which are loaded into XP Server memory in response to an ESP execution request. The library remains loaded unless one of the following occurs:

- XP Server exits
- The **sp_freedll** system procedure is invoked
- The **esp unload dll** configuration parameter is set using **sp_configure**

Open Server API

Adaptive Server uses the Open Server API, which allows users to run the system ESPs provided with Adaptive Server. Users can also implement their own ESPs using the Open Server API.

Table 15-1 lists the Open Server routines required for ESP development. For complete documentation of these routines, see the *Open Server Server-Library/C Reference Manual*.

Table 15-1: Open Server routines for ESP support

Function	Purpose
srv_bind	Describes and binds a program variable to a parameter.
srv_descfmt	Describes a parameter.
srv_numparams	Returns the number of parameters in the ESP client request.
srv_senddone	Sends results completion message.
srv_sendinfo	Sends messages.
srv_sendstatus	Sends status value.
srv_xferdata	Sends and receives parameters or data.
srv_yield	Suspends execution of the current thread and allows another thread to execute.

Example of Creating and Using ESPs

After an ESP function has been written, compiled, and linked into a DLL, you can create an ESP for the function using the **as external name** clause of the **create procedure** command:

```
create procedure procedure_name [parameter_list]
   as external name dll_name
```

The *procedure_name* is the name of the ESP. The ESP name must be the same as the name of its implementing function in the DLL. ESPs are database objects, and their names must follow the rules for identifiers.

The *dll_name* is the name of the DLL in which the implementing function is stored.

The following statement creates an ESP named **getmsgs**, which is in the *msgs.dll*. The **getmsgs** ESP takes no parameters. This example is for a Windows NT Adaptive Server:

```
create procedure getmsgs
as external name "msgs.dll"
```

On a Solaris Adaptive Server, the statement would be:

```
create procedure getmsgs
as external name "msgs.so"
```

to reflect the Solaris naming conventions.

The next statement creates an ESP named **getonemsg**, which is also in the *msgs.dll*. The **getonemsg** ESP takes a message number as a single parameter.

```
create procedure getonemsg @msg int
as external name "msgs.dll"
```

The platform-specific naming conventions for the DLL extension are summarized in Table 15-2.

Table 15-2: Naming conventions for DLL extensions

Platform	DLL Extension
Digital UNIX	.so
HP 9000/800 HP-UX	.sl
Sun Solaris	.so
Windows NT	.dll

When Adaptive Server creates an ESP, it stores the procedure's name in the *sysobjects* system table, with an object type of "XP" and the name of the DLL containing the ESP's function in the *text* column of the *syscomments* system table.

Execute an ESP as if it were a user-defined stored procedure or system procedure. You can 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 Adaptive Server by itself or is the first statement in a batch. For example, you can execute **getmsgs** in any of these ways:

```
getmsgs
execute getmsgs
exec getmsgs
```

You can execute **getonemsg** in any of the following ways:

```
getonemsg 20
getonemsg @msg=20
execute getonemsg 20
execute getonemsg @msg=20
exec getonemsg 20
exec getonemsg @msg=20
```

After Creating an ESP

After you create an ESP, the **source text** describing the ESP is stored in the *text* column of the *syscomments* system table. In previous releases of SQL Server, users often deleted the source text from *syscomments*, in order to save disk space and remove confidential information from this public area. **Do not remove this information from *syscomments***; doing so can cause problems for future upgrades of Adaptive Server. Instead, encrypt the text in *syscomments* by using the **sp_hidetext** system procedure, described in the *Adaptive Server Reference Manual*. For more information, see "Compiled Objects" on page 1-3.

ESPs and Permissions

You can grant and revoke permissions on an ESP as you would on a regular stored procedure.

In addition to the normal Adaptive Server security, you can use the **xp_cmdshell context** configuration parameter to restrict execution

permission of the **xp_cmdshell** system ESP to users who have system administration privileges. Use this configuration parameter to prevent ordinary users from using **xp_cmdshell** to execute operating system commands that they would not have permission to execute directly from the command line. The behavior of the **xp_cmdshell** configuration parameter is platform-specific.

By default, a user must have the **sa_role** to execute **xp_cmdshell**. To grant permission to other users to use **xp_cmdshell**, use the **grant** command. You can revoke the permission with **revoke**. The **grant** or **revoke** permission is applicable irrespective of whether **xp_cmdshell** context is set to 0 or 1.

ESPs and Performance

Since both Adaptive Server and XP Server reside on the same machine, they can affect each other's performance when XP Server is executing a function that consumes significant resources.

You can use the **sp_configure** system procedure to set two configuration parameters, **esp execution priority** and **esp unload dll**, to control the impact of XP Server on Adaptive Server by setting priorities for ESP execution and by freeing XP Server memory.

Setting Priority

Use the **esp execution priority** configuration parameter to set the priority of the XP Server thread high, so that the Open Server scheduler runs it before other threads on its run queue, or low, so that the scheduler runs XP Server only when there are no other threads to run. The default value of **esp execution priority** is 8, but you can set it anywhere from 0 to 15. See the discussion of multithread programming in the *Open Server Server-Library/C Reference Manual* for information about scheduling Open Server threads.

Freeing Memory

You can minimize the amount of memory XP Server uses by unloading a DLL from XP Server memory after the ESP request that loaded it terminates. To do so, set the **esp unload dll** configuration parameter beforehand so that the DLLs are automatically unloaded after ESP execution finishes. As an alternative, if **esp unload dll** is not set, you can free DLLs explicitly using the **sp_freedll** system procedure.

You cannot unload DLLs that support system ESPs.

Creating Functions for ESPs

There are no restrictions on the contents of a function that implements an ESP. The only requirement is that it be written in a procedural programming language capable of:

- Calling a C language function
- Manipulating C language datatypes
- Linking with the Open Server API

By using the Open Client API, an ESP function can send requests to Adaptive Server, either to the one from which it was originally invoked or to another one.

An exception is that an ESP function should not call a C run-time signal routine on Windows NT. This can cause XP Server to fail, because Open Server does not support signal handling on Windows NT.

Files for ESP Development

The header files needed for ESP development are in *$SYBASE/include*. To include these files in your source files, include the following in the source code:

- *ospublic.h*
- *osperror.h*

The Open Server library is in *$SYBASE/lib*.

The source for the sample program shown in “ESP Function Example” on page 15-9 is in *$SYBASE/samples/esp*.

Open Server Data Structures

Three data structures are useful for writing ESP functions:

- SRV_PROC
- CS_SERVERMSG
- CS_DATAFMT

SRV_PROC

All ESP functions are coded to accept a single parameter, a pointer to a SRV_PROC structure. The SRV_PROC structure passes

information between the function and its calling process. ESP developers cannot manipulate this structure directly.

The ESP function passes the SRV_PROC pointer to Open Server routines that get parameter types and data and return output parameters, status codes, and result sets.

CS_SERVERMSG

Open Server uses the CS_SERVERMSG structure to send error messages to a client using the *srv_sendinfo* routine. See the *Open Server-Library/C Reference Manual* for information about CS_SERVERMSG.

CS_DATAFMT

Open Server uses the CS_DATAFMT structure to describe data values and program variables.

Open Server Return Codes

Open Server functions return a code of type CS_RETCODE. The most common CS_RETCODE values for ESP functions are:

- CS_SUCCEED
- CS_FAIL

Outline of a Simple ESP Function

An ESP function should have the following basic structure with regard to its interaction with the Open Server API:

1. Get the number of parameters.
2. Get the values of the input/output parameters and bind them to local variables.
3. Perform the processing using the input parameter values and store the results in local variables.
4. Initialize any output parameters with appropriate values, bind them with local variables, and transfer them to the client.
5. Use **srv_sendinfo()** to send the returned row to the client.
6. Use **srv_sendstatus()** to send the status to the client.

7. Use **srv_senddone()** to inform the client that the processing is done.
8. If there is an error condition, use **srv_sendinfo()** to send the error message to the client.

See the *Open Server Server-Library/C Reference Manual* for documentation of the Open Server routines.

Multithreading

Since Open Server is currently non-preemptive, all ESPs running on the same server must yield to one another, using the Open Server **srv_yield()** routine to suspend their XP Server thread and allow another thread to execute.

See the chapter on Multithread Programming in the *Open Server Server-Library/C Reference Manual* for more information.

ESP Function Example

xp_echo.c has an ESP that accepts a user-supplied input parameter and echoes it to the ESP client, which is the process invoking the ESP. This example includes the *xp_message* function, which sends messages and status, and the *xp_echo* function which processes the input parameter and performs the echoing. You can use this example as a template for building your own ESP functions. The source is in *$SYBASE/esp/samples*.

```
/*
** xp_echo.c
**
**      Description:
**              The following sample program is generic in
**              nature. It echoes an input string which is
**              passed as the first parameter to the xp_echo
**              ESP. This string is retrieved into a buffer
**              and then sent back (echoed) to the ESP client.
*/

#include <string.h>
#include <stdlib.h>
#include <malloc.h>

/* Required Open Server include files.*/
#include <ospublic.h>
#include <oserror.h>
```

```
/*
** Constant defining the length of the buffer that receives the
** input string. All of the Adaptive Server parameters related
** to ESP may not exceed 255 char long.
*/

#define ECHO_BUF_LEN    255

/*
** Function:
**      xp_message
**      Purpose: Sends information, status and completion of the
**      command to the server.
** Input:
**      SRV_PROC *
**      char *  a message string.
** Output:
**      void
*/

void xp_message
(
        SRV_PROC *srvproc,  /* Pointer to Open Server thread
                            control structure */
        char     *message_string /* Input message string */
)

{
        /*
        ** Declare a variable that will contain information
        ** about the message being sent to the SQL client.
        */
        CS_SERVERMSG    *errmsgp;

        /*
        ** Allocate memory for this variable.
        */
        errmsgp = (CS_SERVERMSG *) malloc((CS_INT
           sizeof(CS_SERVERMSG));
        if (errmsgp == NULL)
                return;

        /*
        ** clean the structure */
        */
        memset(errmsgp,(CS_INT)0,(CS_INT) sizeof(CS_SERVERMSG));

        /*
        ** Put you rnumber in as the message number.
        */
        errmsgp->msgnumber = 25000;
```

```
        errmsgp->state = 0;

        /*
        **The message is fatal.
        */
        errmsgp->severity = SRV_FATAL_SERVER;

        /*
        ** Let's copy the string over.
        */
        errmsgp->textlen = strlen(message_string);
        if (errmsgp->textlen >= CS_MAX_MSG )
                return;
        strncpy(errmsgp->text, message_string, errmsgp->textlen);
        errmsgp->status = (CS_FIRST_CHUNK | CS_LAST_CHUNK);

        srv_sendinfo(srvproc, errmsgp, CS_TRAN_UNDEFINED);

        /* Send the status to the client. */
        srv_sendstatus(srvproc, 1);

        /*
        ** A SRV_DONE_MORE instead of a SRV_DONE_FINAL must
        ** complete the result set of an Extended Stored
        ** Procedure.
        */
        srv_senddone(srvproc, SRV_DONE_MORE, 0, 0);

        free(errmsgp);

}


/*
** Function: xp_echo
** Purpose:
**      Given an input string, this string is echoed as an output
**      string to the corresponding SQL (ESP) client.
** Input:
**      SRV_PROC *
** Output
**      SUCCESS or FAILURE
*/
```

```
CS_RETCODE xp_echo
(
        SRV_PROC        *srvproc
)
{
        CS_INT          paramnum;  /* number of parameters */
        CS_CHAR         echo_str_buf[ECHO_BUF_LEN + 1];
                               /* buffer to hold input string */
        CS_RETCODE      result = CS_SUCCEED;
        CS_DATAFMT      paramfmt; /* input/output param format */
        CS_INT          len;      /* Length of input param */
        CS_SMALLINT     outlen;

        /*
        ** Get number of input parameters.*/
        */
        srv_numparams(srvproc, &paramnum);

        /*
        ** Only one parameter is expected.*/
        */
        if (paramnum != 1)
        {
                /*
                ** Send a usage error message.*/
                */
                xp_message(srvproc, "Invalid number of
                 parameters");
                result = CS_FAIL;
        }

        else
        {
                /*
                ** Perform initializations.
                */
                outlen = CS_GOODDATA;
                memset(&paramfmt, (CS_INT)0,
                  (CS_INT)sizeof(CS_DATAFMT));
```

```
/*
** We are receiving data through an ESP as the
** first parameter. So describe this expected
** parameter.
*/
if ((result == CS_SUCCEED) &&
  srv_descfmt(srvproc, CS_GET
  SRV_RPCDATA, 1, &paramfmt) != CS_SUCCEED)
{
        result = CS_FAIL;
}
/*
** Describe and bind the buffer to receive the
** parameter.
*/
if ((result == CS_SUCCEED) &&
  (srv_bind(srvproc, CS_GET, SRV_RPCDATA,
   1, &paramfmt,(CS_BYTE *) echo_str_buf,
   &len, &outlen) != CS_SUCCEED))
{
        result = CS_FAIL;
}
/* Receive the expected data.*/
if ((result == CS_SUCCEED) &&
  srv_xferdata(srvproc,CS_GET,SRV_RPCDATA)
  != CS_SUCCEED)
{
        result = CS_FAIL;
}
/*
** Now we have the input info and are ready to
** send the output info.
*/
if (result == CS_SUCCEED)
{
        /*
        ** Perform initialization.
        */
        if (len == 0)
                outlen = CS_NULLDATA;
        else
                outlen = CS_GOODDATA;
        memset(&paramfmt, (CS_INT)0,
          (CS_INT)sizeof(CS_DATAFMT));
```

```
        strcpy(paramfmt.name, "xp_echo");
        paramfmt.namelen = CS_NULLTERM;
        paramfmt.datatype = CS_CHAR_TYPE;
        paramfmt.format = CS_FMT_NULLTERM;
        paramfmt.maxlength = ECHO_BUF_LEN;
        paramfmt.locale = (CS_LOCALE *) NULL;
        paramfmt.status |= CS_CANBENULL;

        /*
        ** Describe the data being sent.
        */
        if ((result == CS_SUCCEED) &&
                srv_descfmt(srvproc, CS_SET,
                  SRV_ROWDATA, 1, &paramfmt)
                  != CS_SUCCEED)
        {
                result = CS_FAIL;
        }

        /*
        ** Describe and bind the buffer that
        ** contains the data to be sent.
        */
        if ((result == CS_SUCCEED) &&
                (srv_bind(srvproc, CS_SET,
                SRV_ROWDATA, 1,
                &paramfmt, (CS_BYTE *)
                echo_str_buf, &len, &outlen)
                != CS_SUCCEED))
        {
                result = CS_FAIL;
        }

        /*
        ** Send the actual data.
        */
        if ((result == CS_SUCCEED) &&
                srv_xferdata(srvproc, CS_SET,
                SRV_ROWDATA)!= CS_SUCCEED)
        {
                result = CS_FAIL;
        }
}
```

```
                /*
                ** Indicate to the ESP client how the
                ** transaction was performed.
                */
                if (result == CS_FAIL)
                        srv_sendstatus(srvproc, 1);
                else
                        srv_sendstatus(srvproc, 0);
                /*
                ** Send a count of the number of rows sent to
                ** the client.
                */
                srv_senddone(srvproc,(SRV_DONE_COUNT |
                  SRV_DONE_MORE), 0, 1);
        }
        return result;
}
```

Building the DLL

You can use any compiler that can produce the required DLL on your server platform.

For general information about compiling and linking a function that uses the Open Server API, see the *Open Client/Server Supplement*.

Search Order for DLLs

Windows NT searches for a DLL in the following order:

1. The directory from which the application was invoked
2. The current directory
3. The system directory (SYSTEM32)
4. Directories listed in the PATH environment variable

UNIX searches for the library in the directories listed in the LD_LIBRARY_PATH environment variable (on Solaris and Digital UNIX) or SH_LIBRARY_PATH (on HP) in the order in which they are listed.

If XP Server does not find the library for an ESP function in the search path, it attempts to load it from *$SYBASE/DLL* on Windows NT or *$SYBASE/lib* on other platforms.

Absolute path names for the DLL are not supported.

Sample Makefile (UNIX)

The following makefile, *make.unix*, was used to create the dynamically linked shared library for the *xp_echo* program on UNIX platforms. It generates a file named *examples.so* on Solaris and Digital UNIX and *examples.sl* on HP. The source is provided in *$SYBASE/samples/esp*, so you can modify it for your own use.

To build the example library using this makefile, enter:

```
make -f make.unix
```

```
#
# This makefile creates a shared library.  It needs the open
# server header
# files usually installed in $SYBASE/include directory.
# This make file can be used for generating the template ESPs.
# It references the following macros:
#

# PROGRAM is the name of the shared library you may want to
create.

PROGRAM    = example.so
BINARY     = $(PROGRAM)
EXAMPLEDLL= $(PROGRAM)


# Include path where ospublic.h etc reside. You may have them in
# the standard places like /usr/lib etc.


INCLUDEPATH= $(SYBASE)/include


# Place where the shared library will be generated.
DLLDIR     = .


RM         = /usr/bin/rm
ECHO       = echo
MODE       = normal


# Directory where the source code is kept.
SRCDIR     = .


# Where the objects will be generated.
OBJECTDIR= .
```

```
OBJS       = xp_echo.o

CFLAGS     = -I$(INCLUDEPATH)
LDFLAGS    =  $(GLDFLAGS) -Bdynamic

DLLLDFLAGS=  -dy -G


#=============================================================

$(EXAMPLEDLL) : $(OBJS)
      -@$(RM) -f $(DLLDIR)/$(EXAMPLEDLL)
      -@$(ECHO) "Loading $(EXAMPLEDLL)"
      -@$(ECHO) " "
      -@$(ECHO) "MODE:       $(MODE)"
      -@$(ECHO) "OBJS:       $(OBJS)"
      -@$(ECHO) "DEBUGOBJS:  $(DEBUGOBJS)"
      -@$(ECHO) " "


      cd $(OBJECTDIR); \
      ld -o $(DLLDIR)/$(EXAMPLEDLL) $(DEBUGOBJS) $(DLLLDFLAGS)
$(OBJS)
      -@$(ECHO) "$(EXAMPLEDLL) done"
      exit 0


#=============================================================

$(OBJS) : $(SRCDIR)/xp_echo.c
      cd $(SRCDIR); \
      $(CC) $(CFLAGS) -o $(OBJECTDIR)/$(OBJS) -c xp_echo.c
```

Sample Makefile (Windows NT)

The following makefile, *xp_echo.mak*, was used to create the DLL for the *xp_echo* program on Windows NT using Microsoft Visual C++. It generates a file named *xp_echo.dll*. The source for the makefile is provided in *$SYBASE/samples/esp*, so you can modify it for your own use.

To build the example DLL using this makefile, enter:

```
nmake /f xp_echo.mak
```

```
# Nmake macros for building Windows 32-Bit apps

!include <ntwin32.mak>

all: xp_echo.dll
```

```
# Update the object file if necessary

xp_echo.obj: xp_echo.c
     $(cc) $(cflags) $(cvars) $(cdebug) xp_echo.c


# Update the xp_echo.dll if necessary

xp_echo.dll: xp_echo.obj
     $(link) $(linkdebug) -def:xp_echo.def -out:xp_echo.dll \
     xp_echo.obj shell32.lib libsrv.lib
```

Sample Definitions File

The following file, *xp_echo.def*, must be in the same directory as *xp_echo.mak*. It lists every function to be used as an ESP function in the EXPORTS section.

```
LIBRARY   examples


CODE      PRELOAD MOVEABLE DISCARDABLE
DATA      PRELOAD SINGLE


EXPORTS
        xp_echo  .   1
```

Creating and Removing ESPs

Once you have created an ESP function and linked it into a DLL, register it as an ESP in a database. This lets the user execute the function as an ESP.

Use either of these methods to register an ESP:

- The Transact-SQL **create procedure** command
- The **sp_addextendedproc** system procedure

Using *create procedure*

The **create procedure** command syntax for creating an ESP is compatible with proposed ANSI SQL3 syntax. The syntax is:

```
create procedure [owner.]procedure_name
   [[(]@parameter_name datatype [= default] [output]
   [, @parameter_name datatype [= default]
   [output]]...[)]] [with recompile]
   as external name dll_name
```

The *procedure_name* is the name of the ESP as it is known in the database. It must be the same as the name of its supporting function.

The parameter declarations are the same as for stored procedures, as described in Chapter 14, "Using Stored Procedures."

Adaptive Server ignores the **with recompile** clause if it is included in a **create procedure** command used to create an ESP.

The *dll_name* is the name of the library containing the ESP's supporting function.

The *dll_name* can be specified as a name with no extension (for example, *msgs*) or a name with a platform-specific extension, such as *msgs.dll* on Windows NT or *msgs.so* on Solaris.

In either case, the platform-specific extension is assumed to be part of the library's actual file name.

Since **create procedure** registers an ESP in a specific database, you should specify the database in which you are registering the ESP before invoking the command. From **isql**, specify the database with the **use** *database* command, if you are not already working in the target database.

The following statements register an ESP supported by the *xp_echo* routine described in the example on page 15-9, assuming that the function is compiled in a DLL named *examples.dll*. The ESP is registered in the *pubs2* database.

```
use pubs2

create procedure xp_echo @in varchar(255)
as external name "examples.dll"
```

See **create procedure** in the *Adaptive Server Reference Manual* for more information.

Using *sp_addextendedproc*

The **sp_addextendedproc** system procedure is compatible with the syntax used by Microsoft SQL Server. You can use it as an alternative to the **create procedure** command. The syntax is:

```
sp_addextendedproc esp_name, dll_name
```

esp_name is the name of the ESP. It must match the name of the function that supports the ESP.

dll_name is the name of the DLL containing the ESP's supporting function.

The **sp_addextendedproc** must be executed in the *master* database by a user who has the **sa_role**. Therefore, **sp_addextendedproc** always registers the ESP in the *master* database, unlike **create procedure**, which registers the ESP in the current database, which can be any database. Also, unlike the **create procedure** command, the syntax of **sp_addextendedproc** does not allow for parameter checking in Adaptive Server or for default values for parameters.

The following statements register in the *master* database an ESP supported by the *xp_echo* routine described in the example on page 15-9, assuming that the function is compiled in a DLL named *examples.dll*:

```
use master
sp_addextendedproc "xp_echo", "examples.dll"
```

See the *Adaptive Server Reference Manual* for a complete description of the **sp_addextendedproc** command.

Removing an ESP

You can remove an ESP from the database using either the **drop procedure** command or the **sp_dropextendedproc** system procedure.

The syntax for the **drop procedure** command is the same as for stored procedures:

```
drop procedure [owner.]procedure_name
```

The following command drops the *xp_echo* ESP using the Transact-SQL command:

```
drop procedure xp_echo
```

The syntax for the **sp_dropextendedproc** system procedure is:

```
sp_dropextendedproc esp_name
```

For example:

```
sp_dropextendedproc xp_echo
```

Both methods drop the ESP from the database by removing references to it in the *sysobjects* and *syscomments* system tables. They have no effect on the underlying DLL.

Renaming ESPs

Because an ESP name is bound to the name of its function, you cannot rename an ESP using **sp_rename**, as you can with a stored procedure. To change the name of an ESP:

1. Remove the ESP with **drop procedure** or **sp_dropextendedproc**.
2. Rename and recompile the supporting function.
3. Re-create the ESP with the new name using **create procedure** or **sp_addextendedproc**.

Executing ESPs

You execute an ESP using the same **execute** command that you use to execute a regular stored procedure. See “Creating and Executing Stored Procedures” on page 14-5 for syntax and other information about executing stored procedures.

You can also execute an ESP remotely, in the same way that you execute a regular stored procedure remotely. See “Executing Procedures Remotely” on page 14-17 for information about executing remote stored procedures.

Because the execution of any ESP involves a remote procedure call between Adaptive Server and XP Server, you cannot combine parameters by name and parameters by value in the same **execute** command. All the parameters must be passed by name, or all must be passed by value. This is the only way in which the execution of extended stored procedures differs from that of regular stored procedures.

ESPs can return:

- A status value indicating success or failure, and the reason for failure
- Values of output parameters
- Result sets

An ESP function reports the return status value with the **srv_sendstatus** Open Server routines. The return status values from **srv_sendstatus** are application-specific. However, a status of zero indicates that the request completed normally.

An ESP function returns the values of output parameters and result sets using the **srv_descfmt**, **srv_bind**, and **srv_xferdata** Open Server routine. See the “ESP Function Example” on page 15-9 and the *Open*

Server Server-Library/C Reference Manual for more information about passing values from an ESP function. From the Adaptive Server side, returned values from an ESP are handled as they are for a regular stored procedure.

System ESPs

The system ESP **xp_cmdshell** lets you execute an operating system command on the server machine from within Adaptive Server.

In addition, there are several system ESPs that support features that are available on Windows NT, such as the integration of Adaptive Server with the Windows NT Event Log or Mail System.

The names of all system ESPs begin with "xp_". They are created in the *sybsystemprocs* database during Adaptive Server installation. Since system ESPs are located in the *sybsystemprocs* database, their permissions are set there. However, you can run system ESPs from any database. The system ESPs are as follows:

- **xp_cmdshell**
- **xp_deletemail**
- **xp_enumgroups**
- **xp_findnextmsg**
- **xp_logevent**
- **xp_readmail**
- **xp_sendmail**
- **xp_startmail**
- **xp_stopmail**

See the *Adaptive Server Reference Manual* for documentation of the system ESPs. In addition, *Configuring Adaptive Server for Windows NT* discusses some NT-specific features in more detail, such as MAPI and NT Event Log integration, which you implement using the NT-specific system ESPs.

Getting Information About ESPs

Use the **sp_helpextendedproc** system procedure to get information about ESPs registered in the current database.

With no parameters, **sp_helpextendedproc** displays all the ESPs in the database, with the names of the DLLs containing their associated functions. With an ESP name as a parameter, it provides this information only for the specified ESP.

```
sp_helpextendedproc getmsgs

ESP Name   DLL
--------   ---------
getmsgs    msgs.dll
```

Since the system ESPs are in the *sybsystemprocs* database, you must be using the *sybsystemprocs* database to display their names and DLLs:

```
use sybsystemprocs
sp_helpextendedproc

ESP Name     DLL
--------     ---------
xp_freedll   sybyesp

xp_cmdshell  sybyesp
```

If the source text of an ESP was encrypted using **sp_hidetext**, Adaptive Server displays a message advising you that the text is hidden. For information about hiding source text, see **sp_hidetext** in the *Adaptive Server Reference Manual*.

ESP Exceptions and Messages

Adaptive Server handles all messages and exceptions from XP Server. It logs standard ESP messages in the log file in addition to sending them to the client. User-defined messages from user-defined ESPs are not logged, but they are sent to the client.

ESP-related messages may be generated by XP Server, by a system procedure that creates or manipulates ESPs, or by a system ESP. See the *Error Messages* manual for the list of ESP-related messages.

A function for a user-defined ESP can generate a message using the **srv_sendinfo** Open Server routine. See the sample **xp_message** function in the “ESP Function Example” on page 15-9.

16 Triggers: Enforcing Referential Integrity

A **trigger** is a stored procedure that goes into effect when you insert, delete, or update data in a table. You can use triggers to perform a number of automatic actions, such as cascading changes through related tables, enforcing column restrictions, comparing the results of data modifications, and maintaining the referential integrity of data across a database.

This chapter discusses:

How Triggers Work

Triggers can help maintain the referential integrity of your data by maintaining consistency among logically related data in different tables. You have **referential integrity** when the primary key values match the corresponding foreign key values.

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**, and **delete**. A trigger is executed once for each SQL statement.

For example, to prevent users from removing any publishing companies from the *publishers* table, you could write the following trigger:

```
create trigger del_pub
on publishers
for delete
as
begin
    rollback transaction
    print "You cannot delete any publishers!"
end
```

The next time someone tries to remove a row from the *publishers* table, the *del_pub* trigger cancels the deletion, rolls back the transaction, and prints a message to that effect.

You can remove a trigger at any time, for example:

```
drop trigger del_pub
```

A trigger "fires" only after the data modification statement has completed and Adaptive Server has checked for any datatype, rule, or integrity constraint violation. The trigger and the statement that fires it are treated as a single transaction that can be rolled back from within the trigger. If Adaptive Server detects a severe error, the entire transaction is rolled back.

What Triggers Can Do

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 delete matching rows in other tables, using the *title_id* column as a unique key to locating rows in *titleauthor* 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 have a matching value in *titles.title_id*.
- Triggers can enforce restrictions that are much more complex than those that are 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 action based on that comparison.

Using Triggers vs. 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**:

- Cascade changes through related tables in the database
- Enforce complex restrictions by referencing other columns or database objects
- Perform "what if" analyses

Also, referential integrity constraints do not roll back the current transaction as a result of enforcing data integrity. With triggers, you can either roll back or continue the transaction, depending on how you handle referential integrity. For information about transactions, see Chapter 18, "Transactions: Maintaining Data Consistency and Recovery."

If your application requires one of the above tasks, use a trigger. Otherwise, use a referential integrity constraint to enforce data integrity. Adaptive Server checks referential integrity constraints before it checks triggers so that 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(s) the trigger is to take.

For 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."
```

➤ ***Note***

Except for the trigger named *deltitle*, the triggers discussed in this chapter are not included in the *pubs2* database shipped with Adaptive 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.

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 in 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 **insert**, **delete**, or **update** will cause the trigger actions to be carried out. You group multiple trigger actions in an **if** clause with **begin** and **end**.

An **if update** clause tests for an insert or update to a specified column. For updates, the **if update** clause evaluates 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. Do not use the **if update** clause with **delete**. You can specify more than one column, 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 That Are 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**

After Creating a Trigger

After you create a trigger, the **source text** describing the trigger is stored in the *text* column of the *syscomments* system table. In previous releases of SQL Server, users often deleted the source text from *syscomments*, in order to save disk space and remove confidential information from this public area. **Do not remove this information from *syscomments***; doing so can cause problems for future upgrades of Adaptive Server. Instead, encrypt the text in *syscomments* by using the **sp_hidetext** system procedure, described in the *Adaptive Server Reference Manual*. For more information, see "Compiled Objects" on page 1-3.

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 a column or combination of columns whose values uniquely identify a row. The value cannot be NULL and must have a unique index. A table with a primary key is eligible for joins with foreign keys in other tables. Think of the primary key table 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.

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 of the pubs2 Database" on page A-27 shows how the *pubs2* tables are related. The "Diagram of the pubs3 Database" on page B-29 provides the same information for the *pubs3* database.

The **foreign key** is a column or combination of columns whose values match the primary key. A foreign key does not have to be unique. It is 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 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** tables 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; the tables are detail tables.

How Referential Integrity Triggers Work

In most cases, you can enforce referential integrity between tables using the referential constraints described under "Specifying Referential Integrity Constraints" on page 7-37, because the maximum number of references allowed for a single table is 200. If a table exceeds that limit, or has special referential integrity needs, use referential integrity triggers.

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

Adaptive Server uses two special tables in trigger statements: the *deleted* table and the *inserted* table. These are temporary tables used in trigger tests. When you write triggers, you can use these tables to test the effects of a 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* and trigger tables 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 the *inserted* and trigger tables at the same time. The rows in *inserted* are copies of the new rows in the trigger table.

The following trigger fragment uses the *inserted* table to test for changes to the *titles* table *title_id* column:

```
if (select count(*)
    from titles, inserted
    where titles.title_id = inserted.title_id) !=
    @@rowcount
```

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**:

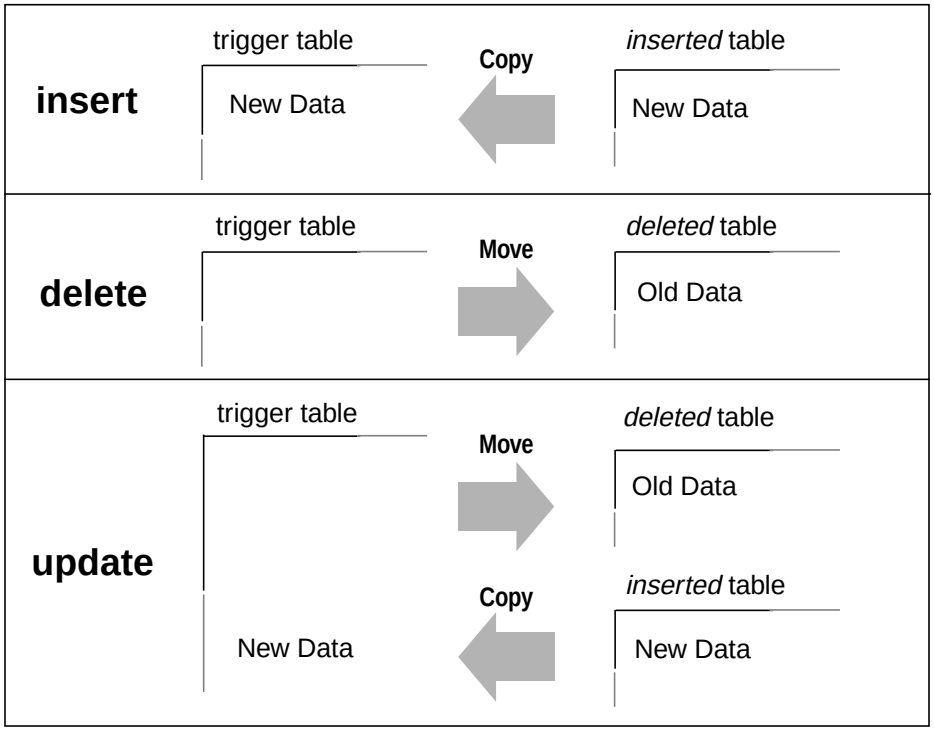


Figure 16-1: Trigger test tables during insert, delete, and update operations

When setting trigger conditions, use the trigger test tables that are appropriate for the data modification. It is not an error to reference *deleted* while testing an **insert** or *inserted* while testing a **delete**; however, those trigger test tables will not contain any rows.

➤ ***Note***

A given trigger fires only once per query. If trigger actions depend on the number of rows affected by a data modification, 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, use local variables to store the value for later examination. All Transact-SQL statements that do not return values reset *@@rowcount* to 0.

Insert Trigger Example

When you insert a new foreign key row, make sure the foreign key matches a primary key. The trigger should check for joins between the inserted rows (using the *inserted* table) 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.

The following trigger compares the *title_id* values from the *inserted* table with those from the *titles* table. It assumes that you are making an 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, the title_id does not exist in
    titles."
  end
/* Otherwise, allow it. */
else
  print "Added! All title_id's exist in titles."
```

In the above 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 trigger joins *titles* and *inserted* to determine whether all the *title_id*'s added to *salesdetail* exist in the *titles* table. If the number of joined rows, which is determined by the **select count(*)** query, differs from *@@rowcount*, then one or more of the inserts is incorrect, and the transaction is canceled.

This trigger prints one message if the insert is rolled back and another if it is accepted. To test for the first condition, try the following **insert** statement:

```
insert salesdetail
values ("7066", "234517", "TC9999", 70, 45)
```

To test for the second condition, enter this statement:

```
insert salesdetail
values ("7896", "234518", "TC3218", 75, 80)
```

Delete Trigger Examples

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 you do not delete the corresponding rows in the dependent tables, you could end up with a database that had detail rows that could not be retrieved or identified. To properly delete the dependent foreign key rows, use a trigger that performs a cascading **delete**.

Cascading Delete 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 is 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.*/
```

Restricted Delete Examples

In practice, you may want to keep some of the detail rows, 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.

Preventing Primary Key Deletions

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 cannot 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 a join is found, the transaction is canceled.

Similarly, the following restricted delete prevents deletes if the primary table, *titles*, has dependent children in *titleauthor*. Instead of counting the rows from *deleted* and *titleauthor*, it checks to see if *title_id* was deleted. This method is more efficient for performance reasons because it checks for the existence of a particular row rather than going through the entire table and counting all the rows.

Recording Errors That Occur

The following trigger uses the **raiserror** command for the error message 35003. **raiserror** sets a system flag to record that the error occurred. Before trying this example, add error message 35003 to the *sysusermessages* system table:

```
sp_addmessage 35003, "restrict_dtrig - delete
failed: row exists in titleauthor for this
title_id."
```

The trigger is as follows:

```
create trigger restrict_dtrig
on titles
for delete as
if exists (select * from titleauthor, deleted where
          titleauthor.title_id = deleted.title_id)
   begin
           rollback transaction
           raiserror 35003
           return
   end
```

To test this trigger, try the following **delete** statement:

```
delete titles
where title_id = "PS2091"
```

Update Trigger Examples

The following example cascades an update from the primary table *titles* to the dependent tables *titleauthor* and *roysched*.

```
create trigger cascade_utrig
on titles
for update as
if update(title_id)
begin
     update titleauthor
           set title_id = inserted.title_id
           from titleauthor, deleted, inserted
           where deleted.title_id = titleauthor.title_id
     update roysched
           set title_id = inserted.title_id
           from roysched, deleted, inserted
           where deleted.title_id = roysched.title_id
     update salesdetail
           set title_id = inserted.title_id
           from salesdetail, deleted, inserted
           where deleted.title_id = salesdetail.title_id
end
```

To test this trigger, suppose that the book *Secrets of Silicon Valley* was reclassified to a psychology book from popular_comp. The following query updates the *title_id* PC8888 to PS8888 in *titleauthor*, *roysched*, and *titles*.

```
update titles
set title_id = "PS8888"
where title_id = "PC8888"
```

Restricted Update Triggers

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 need to protect referential integrity by rolling back the update unless specified conditions are met.

It is best to prohibit any editing changes to a primary key, for example by revoking all permissions on that column. However, if

you want to prohibit updates only under certain circumstances, use a trigger.

Restricted Update Trigger Using Date Functions

The following 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 do not allow changes to "
    print "primary keys on the weekend."
  end
```

Restricted Update Triggers with Multiple Actions

You can 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*. In addition to preventing updates to the primary key on weekends, it 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 because 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 do not allow changes to"
    print "primary keys on the weekend!"
  end
if update (price) or update (advance)
  if exists (select * from inserted
    where (inserted.price * inserted.total_sales)
    < inserted.advance)
    begin
      rollback transaction
      print "We do not allow changes to price or"
      print "advance for a title until its total"
      print "revenue exceeds its latest advance."
    end
```

The next example, created on *titles*, prevents **update** if any of the following conditions is true:

- The user tries to change a value in the primary key *title_id* in *titles*,
- The dependent key *pub_id* is not found in *publishers*, or
- The target column does not exist or is null.

This example assumes that the following error messages have been added to *sysusermessages*:

```
sp_addmessage 35004, "titles_utrg - Update Failed: update of
primary keys %1! is not allowed."

sp_addmessage 35005, "titles_utrg - Update Failed: %1! not found
in authors."
```

The trigger is as follows:

```
create trigger title_utrg
on titles
for update as
begin
     declare @num_updated int,
             @col1_var varchar(20),
             @col2_var varchar(20)
/* Determine how many rows were updated. */
select @num_updated = @@rowcount
    if @num_updated = 0
    return
/* Ensure that title_id in titles is not changed. */
if update(title_id)
    begin
       rollback transaction
     select @col1_var = title_id from inserted
     raiserror 35004 , @col1_var
     return
   end
 /* Make sure dependencies to the publishers table are accounted
for. */
 if update(pub_id)
   begin
     if (select count(*) from inserted, publishers
         where inserted.pub_id = publishers.pub_id
         and inserted.pub_id is not null) != @num_updated
     begin
         rollback transaction
         select @col1_var = pub_id from inserted
         raiserror 35005, @col1_var
         return
     end
   end
/* If the column is null, raise error 24004 and rollback the
** trigger. If the column is not null, update the roysched table
** restricting the update. */
   if update(price)
     begin
         if exists (select count(*) from inserted
         where price = null)
     begin
         rollback trigger with
         raiserror 24004 "Update failed : Price cannot be null. "
     end
```

```
        else
        begin
            update roysched
            set lorange = 0,
            hirange = price * 1000
            from inserted
            where roysched.title_id =  inserted.title_id
        end
       end
end
```

To test for the first error message, 35004, try the following query:

```
update titles
set title_id = "BU7777"
where title_id = "BU2075"
```

To test for the second error message, 35005, try this query:

```
update titles
set pub_id = "7777"
where pub_id = "0877"
```

To test for the third error, which generates message 24004, try this query:

```
update titles
set price = 10.00
where title_id = "PC8888"
```

This query fails because the *price* column in *titles* is null. If it were not null, it would have updated the price for title PC8888 and performed the necessary recalculations for the *roysched* table. Error 24004 is not in *sysusermessages* but it is valid in this case. It demonstrates the "rollback trigger with raiserror" section of the code.

Updating a Foreign Key

A change or an update to a foreign key by itself is probably an error. A foreign key is just a copy of the primary key. Never design the two to be independent. If you want to allow updates of a foreign key, you should protect integrity by creating a trigger that checks updates against the master table and rolls them back if they do not 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 or it is 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 a 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, the old title_id must be in"
        print "salesdetail."
      end
    else
      if (select count(*)
          from titles, inserted
          where titles.title_id =
          inserted.title_id) != @row
        begin
          rollback transaction
          print "No, the new title_id is not in"
          print "titles."
        end
      else
        print "salesdetail table updated"
  end
```

Multirow Considerations

Multirow considerations are particularly important when the function of a trigger is to recalculate summary values, or provide ongoing tallies.

Triggers used to maintain summary values should contain **group by** clauses or subqueries that perform implicit grouping. This creates 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 *@@rowcount* = 1, meaning that only one row in the trigger table was affected. If *@@rowcount* = 1, the trigger actions take effect without a **group by** clause.

Insert Trigger Example Using Multiple Rows

The following 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)
```

Delete Trigger Example Using Multiple Rows

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*.

Update Trigger Example Using Multiple Rows

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
```

Conditional Insert Trigger Example Using Multiple Rows

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 “Using Correlated Subqueries” on page 5-28 for more information on correlated subqueries. The trigger can then take different actions on different rows.

The following trigger example assumes the existence of a table called *junesales*. Here is its **create** statement:

```
create table junesales
(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 *junesales* table, in order to test the conditional trigger. Two of the *junesales* rows have *title_ids* that do not match any of those already in the *titles* table.

```
insert junesales values ("7066", "BA27619", "PS1372", 75, 40)
insert junesales values ("7066", "BA27619", "BU7832", 100, 40)
insert junesales values ("7067", "NB-1.242", "PSxxxx", 50, 40)
insert junesales values ("7131", "PSyyyy", "PSyyyy", 50, 40)
```

When you insert data from *junesales* into *salesdetail*, the statement looks like this:

```
insert salesdetail
select * from junesales
```

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*:

```
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 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 that 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.

Following is the syntax for **rollback trigger:**

```
rollback trigger
   [with raiserror_statement]
```

The syntax for **rollback transaction** is described in Chapter 18, "Transactions: Maintaining Data Consistency and Recovery."

The *raiserror_statement* specifies a **raiserror** statement that 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, Adaptive 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, Adaptive Server rolls back all the work done in these triggers up to and including the update that caused the first trigger to fire.

The following example of an insert trigger performs a similar task to the trigger *forinsertrig1* described on page 16-10. 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)
commit tran
```

See Chapter 18, "Transactions: Maintaining Data Consistency and Recovery," for information on user-defined transactions.

Adaptive 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 a trigger but inside a transaction generates an error that causes Adaptive Server to roll back the transaction and abort the current statement batch.

Nesting Triggers

Triggers can nest to a depth of 16 levels. The current nesting level is stored in the *@@nestlevel* global variable. Nesting is enabled at installation. A System Administrator can turn trigger nesting on and off with the **allow nested triggers** configuration parameter.

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 was deleted by the *delcascadetrig* trigger. With the *delcascadetrig* trigger in effect, deleting the *title_id* "PS2091" from *titles* also deletes the corresponding row(s) 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*, described under "Cascading Delete Example" on page 16-11.

➤ ***Note***

When you put triggers into a transaction, a failure at any level of a set of nested triggers cancels the transaction and rolls back all data modifications. Use **print** or **raiserror** statements in your triggers to determine where failures occur.

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. 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 it to fire again, but it cannot exceed the limit of 16 nesting levels.

For example, assume that 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 you update *new_budget.budget* by increasing the budget of unit *one_department* by 3, Adaptive Server behaves as follows (assuming that 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 (the value is "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 cause the trigger to fire again (it is reactivated only if nested triggers are enabled). Unless conditions within the trigger limit the number of recursions, the nesting level can overflow.

For example, if an update trigger calls a stored procedure that performs an update, the trigger and stored procedure execute only once if **nested triggers** is set to **off**. If **nested triggers** is set to **on**, and the number of updates exceeds 16 by some condition in the trigger or procedure, the loop continues until the trigger or procedure 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, consider the following factors when you are 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.

Adaptive Server will accept a trigger definition that attempts actions for which you do not have permission. The existence of such a trigger aborts any attempt to modify the trigger table because incorrect permissions will cause the trigger to fire and fail. The transaction will be canceled. You must rectify the permissions 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*, Adaptive 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

Adaptive Server imposes the following limitations on triggers:

- 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 can 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.
- The **writetext** statement will not activate insert or update triggers.
- 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.
- You cannot create a trigger or build an index or a view on a temporary object (*@object*)
- You cannot create triggers on system tables. If you try to create a trigger on a system table, Adaptive Server returns an error message and cancels the trigger.
- You cannot use triggers that select from a *text* column or an *image* column of the inserted or deleted table.
- If Component Integration Services is enabled, triggers have limited usefulness on proxy tables because you cannot examine the rows being inserted, updated, or deleted (via the *inserted* and *deleted* tables). You can create a trigger on a proxy table, and it can be invoked. However, deleted or inserted data is not written to the transaction log for proxy tables because the **insert** is passed to the remote server. Hence, the inserted and deleted tables, which are actually views to the transaction log, contain no data for proxy tables.

Implicit and Explicit Null Values

The **if update**(*column_name*) clause 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 does not.

For example, suppose you create the following table:

```
create table junk
(a int null,
 b int not null)
```

and then write the following trigger:

```
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 terms of performance, 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.

For more information on how triggers affect performance, see the *Performance and Tuning Guide*.

set Commands in Triggers

You can use the **set** command inside a trigger. The **set** option you invoke remains in effect during execution of the trigger. Then, the trigger 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 source text reflects the new name of the object being referenced. Use **sp_depends** to get a report of the objects referenced by a trigger. The safest course of action is not to rename any tables or views that are referenced by a trigger.

Trigger Tips

Consider the following tips when creating triggers:

- Suppose you have an **insert** or **update** trigger that calls a stored procedure, which in turn updates the base table. If the **nested triggers** configuration parameter is set to true, the trigger will enter an infinite loop. Before executing an **insert** or **update** trigger, set **sp_configure "nested triggers"** to false.
- When you execute **drop table**, any triggers dependent on that table are also dropped. If you want to preserve any such triggers, change their names with **sp_rename** before dropping the table.
- Use **sp_depends** to see a report on the tables and views referred to in a trigger.
- Use **sp_rename** to rename a trigger.

- A trigger fires only once per query (a single data modification such as an **insert** or **update**). If the query is repeated in a loop, the trigger fires as many times as the query is repeated.

Disabling Triggers

Loading a database from a previous dump causes any triggers defined in the database to fire, which can increase the amount of time required to load the database. Use the **disable trigger** option of the **alter table** command to disable any triggers in a database before you load the database. If the database that contains the table has **select into/bulkcopy/pllsort** turned on, and the table does not contain any indexes, **bcp** automatically runs in fast **bcp** mode after you issue **alter table... disable trigger.** Use **alter table... enable trigger** to re-enable the triggers after the load database is complete. **alter table... disable trigger** uses the following syntax

```
alter table [database_name.[owner_name].]table_name
{enable | disable } trigger [trigger_name]
```

Where *table_name* is the name of the table for which you are disabling triggers, and *trigger_name* is the name of the trigger you are disabling. If you do not specify a trigger, **alter table** disables all the triggers defined in the table. For example, to disable the *del_pub* trigger in the *pubs2* database:

```
alter table pubs2
disable del_pubs
```

To re-enable the *del_pub* trigger, issue:

```
alter table pubs2
enable del_pubs
```

➤ ***Note***

You can only use the disable trigger feature if you are the table owner or database administrator.

If a trigger includes any insert, update, or delete statements, these statements will not run while the trigger is disabled, which may affect the referential integrity of a table.

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 you drop a table, Adaptive Server drops any triggers associated with it. **drop trigger** permission defaults to the trigger table owner and is not transferable.

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 source text for each trigger is stored in *syscomments*. Execution plans for triggers are stored in *sysprocedures*. The system procedures described in the following sections 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* as follows:

```
sp_help deltitle
Name        Owner   Type
----------- ------- -----------
deltitle    dbo     trigger

Data_located_on_segment  When_created
-----------------------  -----------------
not applicable           Jul 10 1997 3:56PM

(return status = 0)
```

You can also use **sp_help** to report the status of a disabled or enabled trigger:

```
1> sp_help trig_insert
2> go
Name Owner
Type
-----------------------------------
trig_insert dbo
trigger
(1 row affected)
data_located_on_segment When_created
-----------------------------------
not applicable Aug 30 1998 11:40PM
Trigger enabled
(return status = 0)
```

sp_helptext

To display the source text of a trigger, execute the system procedure **sp_helptext**, as follows:

```
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
```

If the source text of a trigger was encrypted using **sp_hidetext**, Adaptive Server displays a message advising you that the text is hidden. For information about hiding source text, see **sp_hidetext** in the *Adaptive Server Reference Manual*.

If the System Security Officer has reset the **allow select on syscomments.text column** parameter with the system procedure **sp_configure** (as required to run Adaptive Server in the evaluated configuration), you must be the creator of the trigger or a System Administrator to view the source text of a trigger through **sp_helptext**.

(See **evaluated configuration** in the *Adaptive Server Glossary* for more information.)

sp_depends

The system procedure **sp_depends** lists the triggers that reference an 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)
```

17 Cursors: Accessing Data Row by Row

A **cursor** accesses the results of a SQL **select** statement one or more rows at a time. Cursors allow you to modify or delete rows on an individual basis.

This chapter discusses:

For information on how cursors affect performance, see the *Performance and Tuning Guide*.

How Cursors Work

A cursor is a symbolic name that is associated with a **select** statement. It consists of the following parts:

- **Cursor result set** – the set (table) of rows resulting from the execution of a query that is 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 **update** or **delete** statements with a clause naming the cursor.

You can change the current cursor position through an operation called a **fetch**. The **fetch** command moves the current cursor position one or more rows down the cursor result set.

Figure 17-1 illustrates how the cursor result set and cursor position work when a **fetch** command is performed. In this example, the cursor is defined as follows:

```
declare cal_authors_crsr cursor
for select au_id, au_lname, au_fname
from authors
where state = "CA"
for update
```

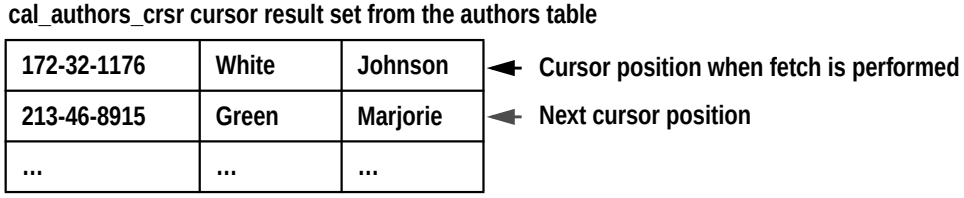


Figure 17-1: How the cursor result set and cursor position work for a fetch

You might think of a cursor as a "handle" on the result set of a **select** statement. It enables you to examine and possibly manipulate one row at a time. However, cursors support only 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.

How Adaptive Server Processes Cursors

Figure 17-2 shows the steps involved in using cursors. The whole point of cursors is to get to the middle box, where the user examines a row and, based on its values, decides what to do.

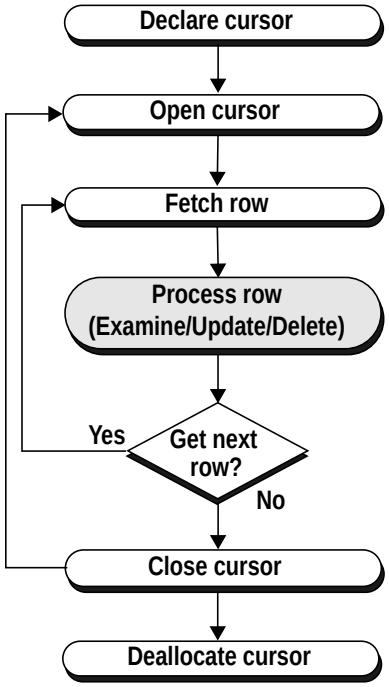


Figure 17-2: Steps for using cursors

When accessing data using cursors, Adaptive Server divides the process into the following operations:

- Declaring the cursor

  Adaptive Server creates the cursor structure and, if the cursor is declared within a stored procedure, compiles the query defined for that cursor. It stores the compiled query plan but does not execute it.

  For example, the following cursor declaration, *business_crsr*, finds the titles and identification numbers of all business books in the *titles* table. It also allows you to update the *price* column in the future through the cursor:

  ```
  declare business_crsr cursor
  for select title, title_id
  from titles
  where type = "business"
  for update of price
  ```

  Be sure to use the **for update** clause when declaring a cursor, to ensure that Adaptive Server performs the positioned updates correctly.

- Opening the cursor

  When a cursor that has been declared outside of a stored procedure is opened, Adaptive Server generates an optimized query plan and compiles the cursor. It then performs preliminary operations for executing the scan defined in the cursor and returning a result set. For example, when a cursor is opened, Adaptive Server may place a shared intent lock lock on a table or create a work table.

  If a cursor has been declared within a stored procedure, it already exists in compiled form. When the cursor is opened, Adaptive Server only needs to perform preliminary operations for executing a scan and returning a result set.

- Fetching from the cursor

  The **fetch** command executes the compiled cursor to return one or more rows meeting the conditions defined in the cursor. By default, a **fetch** returns only a single row. The first **fetch** returns the first row that meets the cursor's search conditions and stores the current position of the cursor. The second fetch uses the cursor position from the first fetch, returns the next row that meets the search conditions, and stores its current position. Each subsequent fetch uses the cursor position of the previous fetch to locate the next cursor result.

  The number of rows returned by a **fetch** can be specified using the **set cursor rows** command. See "Getting Multiple Rows with Each fetch" on page 17-15.

  In the following example, the **fetch** command displays the title and identification number of the first row in the *titles* table containing a business book:

  ```
  fetch business_crsr
  title                                 title_id
  ------------------------------------  --------
  The Busy Executive's Database Guide   BU1032

  (1 row affected)
  ```

  Running **fetch business_crsr** a second time displays the title and identification number of the next business book in *titles*.

- Processing the row by examining, updating, or deleting it through the cursor

Adaptive 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.

The following **update** statement raises the price of business books by 5 percent; it affects only the book currently pointed to by the *business_crsr* cursor:

```
update titles
set price = price * .05 + price
where current of business_crsr
```

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.

- Closing the cursor

  Adaptive 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. For example:

  ```
  close business_crsr
  ```

  When you close a cursor and then reopen it, Adaptive Server is ready to re-create the cursor result set. When you perform the **fetch**, Adaptive Server positions the cursor before the first valid row. This allows you to process a cursor result set as many times as necessary. You can close the cursor at any time; you do not have to go through the entire result set.

- Deallocating the cursor

  Adaptive Server dumps the query plan from memory and eliminates all trace of the cursor structure. For example:

  ```
  deallocate cursor business_crsr
  ```

  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, Adaptive Server determines whether the cursor is updatable based on the type of query that defines the cursor result set. However, it is a good idea

to explicitly specify one or the other; for updates, this ensures that Adaptive Server performs the positioned updates correctly. 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 *cursor_name* is the name of the cursor. It must be a valid Adaptive Server identifier containing no more than 30 characters, and it must start with a letter, a pound sign (#), or an underscore (_).

The *select_statement* is the query that defines the cursor result set. See **select** in the *Adaptive Server Reference Manual* for information about its options. 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.

The **for read only** option specifies that the cursor result set cannot be updated. In contrast, the **for update** option specifies that the cursor result set is updatable. You can specify **of** *column_name_list* after **for update** with the list of columns from the *select_statement* defined as updatable.

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.

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"
for update
```

The *select_statement* can contain references to Transact-SQL parameter names or local variables. However, the names can reference only the 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."

Types of Cursors

There are four types of cursors:

- **Client cursors** – are declared through Open Client calls (or Embedded SQL). Open Client keeps track of the rows returned from Adaptive Server and buffers them for the application. Updates and deletes to the result set of client cursors can be done only through the Open Client calls.
- **Execute cursors** – are a subset of client cursors whose result set is defined by a stored procedure. The stored procedure can use parameters. The values of the parameters are sent through Open Client calls.
- **Server cursors** – are declared in SQL. If they are used in stored procedures, the client executing the stored procedure is not aware of them. Results returned to the client for a **fetch** are the same as the results from a normal **select**.
- **Language cursors** – are declared in SQL without using Open Client. As with server cursors, the client is not aware of the cursors, and the results are returned to the client in the same format as a normal **select**.

Client cursors, through the use of applications using Open Client calls or Embedded-SQL, are the most frequently used form of cursors. To simplify the discussion of cursors, the examples in this manual are for language and server cursors only. For examples of client or execute cursors, see your Open Client or Embedded-SQL documentation.

Cursor Scope

A cursor's existence depends on its **scope**. The scope refers to the context in which the cursor is used: within a user session, a stored procedure, or a trigger.

Within a user session, the cursor exists only until the user ends the session. The cursor does not exist for any additional sessions that

other users start. After the user logs off, Adaptive Server deallocates the cursors created in that session.

If a **declare cursor** statement is part of a stored procedure or trigger, the cursor created within it applies to that scope and to the scope that launched the stored procedure or trigger. However, cursors declared inside a trigger on an *inserted* or a *deleted* table are not accessible to any nested stored procedures or triggers. Such cursors **are** accessible within the scope of that trigger. Once the stored procedure or trigger completes, Adaptive Server deallocates the cursors created within it.

Figure 17-3 illustrates how cursors operate between scopes.

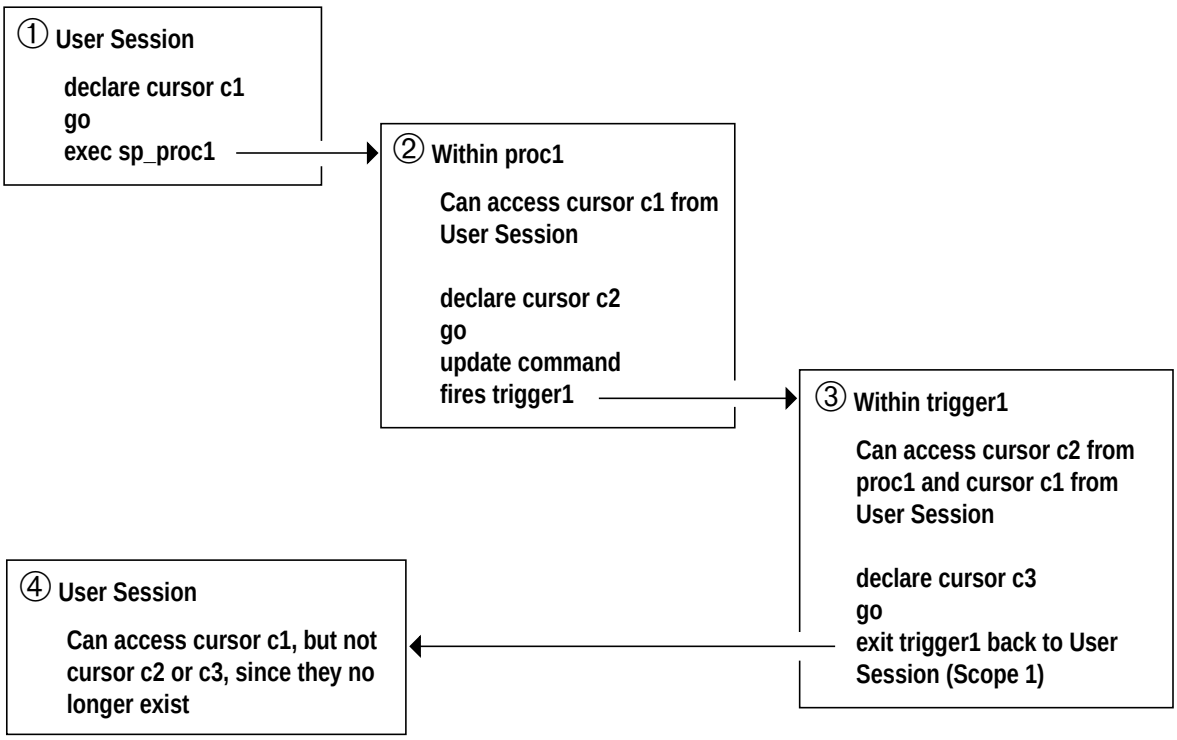


Figure 17-3: How cursors operate within scopes

A cursor name must be unique within a given scope. Adaptive 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, the following stored procedure works because only one *names_crsr* cursor is defined in its scope:

```
create procedure proc2 @flag int
as
if @flag > 0
    declare names_crsr cursor
    for select au_fname from authors
else
    declare names_crsr cursor
    for select au_lname from authors
return
```

Cursor Scans and the Cursor Result Set

The method Adaptive Server uses to create the cursor result set depends on the type of query used. If the query does not require a worktable, Adaptive Server performs a **fetch** by cursoring down the base table using the table's index keys, much like a **select** statement, except that it returns the number of rows specified by the **fetch**. After the **fetch**, Adaptive Server positions the cursor at the next valid index key, until you fetch again or close the cursor.

Some queries use worktables to generate the cursor result set. To verify whether a particular cursor uses a worktable, check the output of a **set showplan, no exec on** statement.

Be aware that when a worktable is used, the rows retrieved with a cursor **fetch** statement 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 a worktable to order the rows for the cursor result set. Adaptive Server does not lock the rows in the base table that correspond to the rows in the worktable, which permits other clients to update these base table rows. Hence, the rows returned to the client from the cursor statement are different from the base table rows. See "Cursors and Locking" on page 17-25 for more information on how locks work with cursors.

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.

Adaptive 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 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. Adaptive Server allows the declaration of cursors on tables without unique indexes, as long as you have not specified **for update** in the **declare cursor** statement. If an **update** does not change the position of the row, the cursor position does not change until the next **fetch**.

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, Adaptive Server attempts to determine whether a cursor is updatable before designating it as read-only.

You can explicitly specify whether a cursor is updatable by using the **read only** or **update** keywords in the **declare** statement. Specifying a cursor to be updatable ensures that Adaptive Server performs the positioned updates correctly. Make sure the table being updated has a unique index. If it does not, Adaptive Server rejects the **declare cursor** statement.

The following example 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 constructs:

- **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. Adaptive 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*. See "Types of Cursors" on page 17-7 for information on the types of cursors Adaptive Server supports.

Determining Which Columns Can Be Updated

If you do not specify a *column_name_list* with the **for update** clause, all the specified columns in the query are updatable. Adaptive Server attempts to use unique indexes for updatable cursors when scanning the base table. For cursors, Adaptive Server considers an index containing an IDENTITY column to be unique, even if it is not so declared.

Adaptive Server allows you to update 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*. However, when you specify a *column_name_list* with **for update**, you can update only the columns in that list.

In the following example, Adaptive 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, Adaptive 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, Adaptive Server must use a unique index defined for one or more of the columns to scan the base table. If no unique index exists, Adaptive Server returns an error message.

In most cases, you should include only columns to be updated in the *column_name_list* of the **for update** clause. If the table has only one unique index, you do not need to include its column in the **for update** *column_name_list*; Adaptive Server will find it when it performs the cursor scan. If the table has more than one unique index, you should include its column in the **for update** *column_name_list*, so that Adaptive Server can find it quickly for the cursor scan. For example, the table used in the following **declare cursor** statement has one unique index, on the column *c3*, so that column should not be included in the **for update** list:

```
declare mycursor cursor
for select c1, c2, 3
from mytable
for update of c1, c2
```

However, if *mytable* has more than one unique index, for example, on columns *c3* and *c4*, you need to specify one unique index in the **for update** clause as follows:

```
declare mycursor cursor
for select c1, c2, 3
from mytable
for update of c1, c2, c3
```

Allowing Adaptive Server to use the unique index in the cursor scan in this manner helps to prevent an update anomaly called the **Halloween problem**. Another way to avoid the Halloween problem is to create tables with the **unique auto_identity index** database option set to **on**. See the **unique auto_identity index** database option description in the *System Administration Guide* for more information.

The Halloween problem occurs when a client updates a column through a cursor, and that column defines the order in which the rows are returned from the base tables (that is, a unique indexed column). For example, if Adaptive 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. The row seems to appear twice in the result set: when the index key is updated by the client and when the updated index row moves farther down the result set.

Opening Cursors

After you declare a cursor, you must open it to **fetch**, **update**, or **delete** rows. Opening a cursor lets Adaptive Server begin to create the cursor result set by evaluating the **select** statement that defines the

cursor and makes it available for processing. The syntax for the **open** statement is:

open ***cursor_name***

Adaptive 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

A **fetch** completes the cursor result set and returns one or more rows to the client that is responsible for extracting the column data from the row. Depending on the type of query defined in the cursor, Adaptive Server creates the cursor result set either by scanning the tables directly or by scanning a worktable generated by the query type.

The **fetch** command positions the cursor before the first row of the cursor result set. If the table has a valid index, Adaptive Server positions the cursor at the first index key.

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*****]**

After generating the cursor result set, Adaptive Server moves the cursor position one or more rows down the result set. It retrieves the data from each row of the result set and stores the current position, allowing additional fetches until Adaptive Server reaches the end of the result set.

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

(1 row affected)
```

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

(1 row affected)
```

After you **fetch** all the rows, the cursor points to the last row of the result set. If you **fetch** again, Adaptive Server returns a warning through the *@@sqlstatus* variable (described under "Checking the Cursor Status" on page 17-14) 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. Close and reopen the cursor to generate the cursor result set again and start fetching from the beginning.

The **into** clause specifies that Adaptive 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
```

Adaptive 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

Adaptive 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 meanings:

Table 17-1: @@sqlstatus values

Value	Meaning
0	Indicates successful completion of the **fetch** statement.
1	Indicates that the **fetch** statement resulted in an error.

Table 17-1: @@sqlstatus values

Value	Meaning
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*. Other statements have no effect on *@@sqlstatus*.

Getting Multiple Rows with Each *fetch*

By default, the **fetch** command retrieves only one row at a time. You can use the **cursor rows** option of the **set** command to change the number of rows that are returned by **fetch**. 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 *number* can be a numeric literal with no decimal point or a local variable of type *integer*. The default setting is 1 for each cursor you declare. You can **set** the **cursor rows** option for any 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
```

After you set the number of cursor rows, each **fetch** of *authors_crsr* returns three 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

(3 rows affected)
```

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 buffers the rows sent to the client application. The client still sees a row-by-row access, but each **fetch** results in fewer calls to Adaptive Server, which improves performance.

Checking the Number of Rows Fetched

Use the *@@rowcount* global variable to monitor the number of rows of the cursor result set returned to the client up to the last fetch. This variable displays 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

---------
        5

(1 row affected)
```

Updating and Deleting Rows Using Cursors

If the cursor is updatable, use the **update** or **delete** statement to update or delete rows. Adaptive Server determines whether 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 17-10 for more information.

Updating Cursor Result Set Rows

You can use the **where current of** clause of the **update** statement to 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.

Adaptive 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 the columns in that list.

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, Adaptive Server positions the cursor before the row following the deleted row in the cursor 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, Adaptive 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 that **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

(3 rows affected)
```

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.

Closing and Deallocating Cursors

When you are finished with the result set of a cursor, you can **close** it. The syntax for closing a cursor is:

```
close cursor_name
```

Closing the cursor does not change its definition. You can open the cursor again, and Adaptive Server will create 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

(3 rows affected)
```

If you want to discard the cursor, you must deallocate it. The syntax for the **deallocate cursor** statement 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, Adaptive 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
```

The results of the query are

```
author                      au_id
-------------------------   -----------
Johnson White               172-32-1176
Marjorie Green              213-46-8915
Cheryl Carson               238-95-7766
Michael O'Leary             267-41-2394
Dick Straight               274-80-9391
Meander Smith               341-22-1782
Abraham Bennet              409-56-7008
Ann Dull                    427-17-2319
Burt Gringlesby             472-27-2349
Chastity Locksley           486-29-1786
Morningstar Greene          527-72-3246
Reginald Blotchet Halls     648-92-1872
Akiko Yokomoto              672-71-3249
Innes del Castillo          712-45-1867
Michel DeFrance             722-51-5454
Dirk Stringer               724-08-9931
Stearns MacFeather          724-80-9391
Livia Karsen                756-30-7391
Sylvia Panteley             807-91-6654
Sheryl Hunter               846-92-7186
Heather McBadden            893-72-1158
Anne Ringer                 899-46-2035
Albert Ringer               998-72-3567

(23 rows affected)
```

The following steps show how to use a cursor with the above query:

1. 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
   for update
   ```

2. Open the cursor:

   ```
   open newauthors_crsr
   ```

3. Fetch rows using the cursor:

   ```
   fetch newauthors_crsr
   ```

```
author                        au_id
-------------------------     -----------
Johnson White                 172-32-1176

(1 row affected)
```

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
go
fetch newauthors_crsr
author                        au_id
-------------------------     -----------
Marjorie Green                213-46-8915
Cheryl Carson                 238-95-7766
Michael O'Leary               267-41-2394
Dick Straight                 274-80-9391
Meander Smith                 341-22-1782

(5 rows affected)
```

Each subsequent **fetch** brings back five more rows:

```
fetch newauthors_crsr
author                        au_id
-------------------------     -----------
Abraham Bennet                409-56-7008
Ann Dull                      427-17-2319
Burt Gringlesby               472-27-2349
Chastity Locksley             486-29-1786
Morningstar Greene            527-72-3246

(5 rows affected)
```

The cursor is now positioned at author Morningstar Greene, the last row of the current **fetch**.

4. Perform the following update for Ms. Greene, who has become weary of New Age jokes about her name:

   ```
   update authors
   set au_fname = "Voilet"
   where current of newauthors_crsr
   ```

   The cursor remains at Ms. Greene's record until the next **fetch**.

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, Adaptive 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 use only one query to accomplish a task 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. (See "Cursor Scope" on page 17-7 for how cursors work within a scope.)

For example, the following stored procedure *au_sales* checks the *sales* table to see if any books by a particular author have sold well. It uses a cursor to examine each row, and then prints the information. Without the cursor, it would need several **select** statements to accomplish the same task. Note that outside stored procedures, you cannot include other statements with **declare cursor** in the same batch.

```
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
```

For example:

```
au_sales "172-32-1176"
```

```
Prolonged Data Deprivation: Four Case Studies -- Had good sales this
year.

(return status = 0)
```

For more information about stored procedures, see Chapter 14, "Using Stored Procedures." See also the *Performance and Tuning Guide* for information about how stored procedures that use cursors affect performance.

Cursors and Locking

Cursor locking methods are similar to other locking methods for Adaptive 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, Adaptive 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, Adaptive 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 Adaptive 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**.

➤ ***Note***

Adaptive Server releases an update lock when the cursor position moves off the data page. Since an application buffers rows for client cursors, the corresponding server cursor may be positioned on a different data row and page than the client cursor. In this case, a second client could update the row that represents the current cursor position of the first client, even if the first client used the **for update** option.

Any exclusive locks acquired by a cursor in a transaction are held until the end of that transaction. This also applies to shared or update locks when you use the **holdlock** keyword 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 can also continue to acquire locks as it fetches additional rows.

For more information about cursor locking in Adaptive Server, see the *Performance and Tuning Guide*.

Cursor Locking Options

These are the effects of specifying the **holdlock** or **shared** options (of the **select** statement) when you define an updatable cursor:

- If you omit both options, you can read data on the currently fetched pages only. Other users cannot update your currently fetched pages, through a cursor or otherwise. Other users can declare a cursor on the same tables you use for your cursor, but they cannot get an update lock on your currently fetched pages.
- If you specify the **shared** option, you can read data on the currently fetched pages only. Other users cannot update your currently fetched pages, through a cursor or otherwise.
- If you specify the **holdlock** option, you can read data on all pages fetched (in a current transaction) or only the pages currently fetched (if not in a transaction). Other users cannot update your currently fetched pages or pages fetched in your current transaction, through a cursor or otherwise. Other users can declare a cursor on the same tables you use for your cursor, but they cannot get an update lock on your currently fetched pages or the pages fetched in your current transaction.
- If you specify both options, you can read data on all pages fetched (in a current transaction) or only the pages currently fetched (if not in a transaction). Other users cannot update your currently fetched pages, through a cursor or otherwise.

Getting Information About Cursors

Use the system procedure **sp_cursorinfo** to find information about a cursor's 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)
```

Other methods of checking the status of a cursor are to use the *@@sqlstatus* and *@@rowcount* global variables. See "Checking the Cursor Status" on page 17-14 for information on *@@sqlstatus* and "Checking the Number of Rows Fetched" on page 17-16 for information on *@@rowcount*.

For more information about **sp_cursorinfo**, see the *Adaptive Server Reference Manual*.

Using Browse Mode in Place of Cursors

Browse mode lets you search through a table and update its values one row at a time. It is used in front-end applications using DB-Library and a host programming language. Browse mode is useful because it provides compatibility with Open Server™ applications and older Open Client libraries. However, its use in new Client-Library™ applications (release 10.0.x and later) is discouraged, because cursors provide the same functionality in a more portable and flexible manner. Also, browse mode is Sybase-specific and is not suited to heterogeneous environments.

Normally, you should use cursors to update data when you want to change table values row by row. Client-Library applications can use Client-Library cursors to implement some browse-mode features, such as updating a table while fetching rows from it. However, be aware cursors may cause locking contention in the tables being selected.

For more information on browse mode, see the **dbqual** function in the Open Client/Server™ documentation.

Browsing a Table

To browse a table in a front-end application, append the **for browse** keywords to the end of the **select** statement sent to Adaptive Server.

For example:

Start of **select** *statement in an Open Client application*

```
...
for browse
```

Completion of the Open Client application routine

A table can be browsed in a front-end application if its rows have been timestamped.

Browse Mode Restrictions

A **for browse** clause cannot be used in statements involving the **union** operator or in cursor declarations.

The use of the keyword **holdlock** is forbidden in a **select** statement that includes the **for browse** option.

The keyword **distinct** in the select statement is ignored in browse mode.

Timestamping a New Table for Browsing

When creating a new table for browsing, include a column named *timestamp* in the table definition. This column is automatically assigned the *timestamp* datatype; you do not have to specify its datatype. For example:

```
create table newtable(col1 int, timestamp,
    col3 char(7))
```

Whenever you insert or update a row, Adaptive Server timestamps it by automatically assigning a unique *varbinary* value to the *timestamp* column.

Timestamping an Existing Table

To prepare an existing table for browsing, add a column named *timestamp* with **alter table**. For example:

```
alter table oldtable add timestamp
```

A *timestamp* column with a NULL value is added to each existing row. To generate a timestamp, update each row without specifying new column values.

For example:

```
update oldtable
set col1 = col1
```

Comparing *timestamp* Values

Use the **tsequal** system function to compare timestamps when you are using browse mode in a front-end application. For example, the following statement updates a row in *publishers* that has been browsed. It compares the *timestamp* column in the browsed version of the row with the hexadecimal timestamp in the stored version. If the two timestamps are not equal, you receive an error message, and the row is not updated.

```
update publishers
set city = "Springfield"
where pub_id = "0736"
and tsequal(timestamp,0x0001000000002ea8)
```

Do not use the **tsequal** function in the **where** clause as a search argument. When you use **tsequal**, the rest of the **where** clause should match a single row uniquely. Use the **tsequal** function only in **insert** and **update** statements. If a *timestamp* column is used as a search clause, it should be compared like a regular *varbinary* column, that is, *timestamp1 = timepstamp2*.

Join Cursor Processing and Data Modifications

This section describes changes to cursor behavior beginning with version 11.9.2 that may affect applications that use cursors. It also describes cursor positioning and modification rules for cursors on data-only-locked tables.

Updates and Deletes That Can Affect the Cursor Position

Two types of deletes and updates can affect the row at the cursor position:

- **Positioned** deletes and updates, using **delete...where current of** or **update...where current of** to change the row at the cursor position
- **Searched** deletes and updates, that is, any **delete** or **update** query that changes a value in the row at the cursor position, but without including a **where current of** clause.

Cursor Positioning After a *delete* or *update* Command Without Joins

The behavior of a **delete** or **update** command to the base table for a cursor on a single-table query depends on the type of modification, the locking scheme of the base table, and whether the clustered index of the base table is affected:

- A positioned **delete** or a searched **delete** that deletes the row at the cursor location positions the cursor to the next qualifying row on the table. This is true for both allpages-locked and data-only-locked tables. A subsequent positioned **update** or **delete** via this cursor is disallowed until the next **fetch** is done to position the cursor on the next row that qualifies.
- A searched or positioned **update** that does not change the position of the row leaves the current position of the cursor unchanged. The next **fetch** returns the next qualifying row.
- A searched or positioned **update** on an allpages-locked table can change the location of the row; for example, if it updates key columns of a clustered index. The cursor does not track the row; it remains positioned just before the next row at the original location. Positioned updates are not allowed until a subsequent **fetch** returns the next row. The updated row may be visible to the cursor a second time, if the row moves to a later position in the search order.

  Data-only-locked tables have fixed row IDs, so expanding updates or updates that affect the clustered key do not move the location of the row. The cursor remains positioned on the row, and the next **fetch** returns the next qualifying row.

The cursor positioning behavior described above is unchanged from previous releases.

Effects of Updates and Deletes on Join Cursors

When a searched or positioned **delete** is issued on the row at the cursor position of a cursor that includes a join, there can be one of two results:

- Searched deletes close the cursor implicitly. The next **fetch** returns error 582:

  ```
  Cursor 'cursor_name' was closed implicitly because
  the current cursor position was deleted due to an
  update or a delete. The cursor scan position could
  not be recovered. This happens for cursors which
  reference more than one table.
  ```

- Positioned deletes fail, with error 592:

  ```
  The DELETE WHERE CURRENT OF to the cursor
  'cursor_name' failed because the cursor is on a
  join.
  ```

  The cursor remains open, positioned at the same row.

When a searched or positioned **update** is issued on the join columns of a cursor:

- Searched updates to clustered index keys on allpages-locked tables succeed, but implicitly close the cursor, so the next fetch returns error 582. Searched updates to clustered index keys do not close cursors on data-only-locked tables.
- Positioned updates for all locking schemes, and searched updates that do not change a clustered index key on an allpages-locked table succeed and do not close the cursor.

  The cursor position depends on the type of table and whether the column has a clustered index. See “Cursor Positioning After a delete or update Command Without Joins” on page 17-30 for more information.

  Join column buffering may affect the result sets returned when join columns are updated. See “Join Cursor Processing and Data Modifications” on page 17-29 for more information.

Table 17-2 shows how **delete** and **update** commands affect join cursors.

In this table, “Left open” mans that the cursor is not closed by the update. The cursor is still positioned on the row, so positioned updates can still be made, and the next **fetch** also succeeds.

Table 17-2: Effects of deletes and join column updates in join cursors

	Allpages-Locked	Data-Only-Locked
delete commands		
Positioned direct delete	Error 592	Error 592
Searched direct delete	Error 582	Error 582
Searched deferred delete	Error 582	Error 582
Searched delete affecting clustered index (deferred)	Error 582	Error 582
update commands		
Positioned direct update	Left open	Left open
Searched direct update	Left open	Left open
Searched deferred update	Left open	Left open
Searched update affecting clustered index (deferred)	Error 582	Left open

Effects of Join Column Buffering on Join Cursors

For cursors on queries that include joins, the join columns, search arguments and select list values for the outer table in the join order are buffered with the first **fetch**. If more than one row is returned from the inner table in the join order, subsequent fetches use the buffered value for the outer row. This means that the behavior of applications that update join columns and search arguments through cursors can vary, depending on the join order chosen for the query. Join column buffering affects both allpages-locked tables and data-only-locked tables.

➤ ***Note***

In version 11.5.x, join cursors on allpages-locked tables did not buffer join values.

Effects of Column Buffering During Cursor Scans

The results of cursor queries that perform joins may produce varying results, depending on the join order chosen for the query, if updates to the join column take place during the session. The following two tables are used to illustrate how join order can affect cursor results sets.

```
select * from dept

dept_id deptloc
------- -----------------
      1 Elm Street
      2 Acacia Drive
      3 Maple Lane
      4 Oak Avenue

select * from employee

dept_id empid
------- ---------
      1 172321176
      1 213468915
      2 341221782
      2 409567008
      2 427172319
      3 472272349
      3 486291786
```

Example of Join Column Buffering

These statements declare a cursor, open it, and fetch the first row:

```
declare j_curs cursor for
select d.dept_id, e.empid, d.deptloc
from dept d , employee e
where d.dept_id = e.dept_id
and d.dept_id = 2

open j_curs

fetch j_curs

dept_id     empid       deptloc
----------- ----------- --------------------
          2   341221782 Acacia Drive
```

If *dept* is chosen as the outer column in the join order, the value 2 for *dept_id* is buffered. The following **update** changes the *dept_id* of the join column in *dept*:

```
update dept set dept_id = 12
where current of  j_curs
```

This update changes the value stored in the table row, but does not alter the buffered value, hiding the result of this update to the base table from the cursor. The effect would be the same if the join column in *dept* were updated by a non-positioned update.

This **update** changes the *dept_id* of the *employee* row:

```
update employee set dept_id = 12
where current of j_curs
```

The next **fetch** on the table returns the second row in the result set:

```
fetch j_curs
dept_id     empid       deptloc
----------- ----------- --------------------
          2   409567008 Acacia Drive
```

All matching rows from *employee* can be fetched and updated.

If the join order for this cursor selects *employee* as the outer table, and the join column in *dept* is updated after the first **fetch**, the second **fetch** would find no matching rows. This is the sequence of steps:

1. The value 2 for *employee.dept_id* is buffered during the first fetch.
2. A searched or positioned update changes the value of *dept.dept_id* to 12.
3. At the second **fetch**, 2, the buffered value for *employee.dept_id* is used, and the inner table is scanned for matching values; since *dept.dept_id* has been changed to 12, the second **fetch** does not return a row.

The two remaining rows in *employee* with *dept_id* equal to 2 cannot be fetched by the cursor.

Example of Search Argument Buffering

Due to buffering of search arguments, the sensitivity of cursor results to updates of the search arguments on cursor select queries also depends on join order. The following example uses the *dept* and *employee* tables as shown page 17-33. The following cursor joins on the *dept_id* columns of the table, using a search argument on the *deptloc* column:

```
declare c cursor for
select d.dept_id, empid, deptloc
from dept d , employee e
where d.dept_id = e.dept_id
and deptloc = "Acacia Drive"
```

The first **fetch** on this table returns this row:

```
dept_id   empid       deptloc
--------- ----------- --------------
        2 41221782    Acacia Drive
```

This positioned update statement changes *employee.dept_id* value to 4 for the current row; it needs to be performed for each employee in the department located at Acacia Drive:

```
update employee set dept_id = 4
where current of c
```

This positioned update changes the *dept.deptloc* value from "Acacia Drive" to "Pine Way",

```
update dept set deptloc = "Pine Way"
where current of c
```

If the **update** to *deptloc* is issued after the first **fetch**, the result set may or may not return all of the rows that need to be changed; the results depend on the join order chosen for the query:

- If *dept* is chosen as the outer table, the *dept.dept_id* and *dept.deptloc* columns, the values "2" and "Acacia Drive", are buffered. After the first **fetch**, the **update** to *deptloc* updates the base table, but not the buffered values, and subsequent **fetch** commands return additional result set rows.
- If *employee* is chosen as the outer table, only the *dept_id* qualification on the *employee* table is buffered. The **update** to the *deptloc* column changes the base table, and when the search argument qualification "Acacia Drive" is applied to the *deptloc* table at the next **fetch**, no rows are returned, even though two rows with *employee.dept_id* of 2 remain in the table.

Note that issuing the **update** to *deptloc* after all rows from the *employee* table have been fetched would accomplish both updates, without a dependency on the join order.

Effects of Select-List Buffering

Columns from the select list of the outer table in the join order are buffered when a row from the outer table is fetched. If a searched or positioned update is performed on a column in the select list, and another row is fetched from the inner table, the buffered value from the outer table appears in the cursor result row.

Recommendations

In general, applications should attempt to avoid updating join columns or columns with search clauses and other predicates to change their value when cursor scans are in progress.

- Avoid updates to the join columns of cursors or to the search arguments of join cursors whenever possible, unless you are completely sure of the join order.
- Use a cursor query that creates a worktable, and then use searched updates and deletes on the base table. When a cursor select query requires a worktable, the cursor is always insensitive to changes to the underlying table. Cursors that include **order by**, **distinct**, **group by**, or other clauses that create a worktable cannot be updated with positioned updates.
- If you are using cursors to update both join columns, consider using searched updates instead of positioned updates. Although further fetches may return buffered values instead of values that have been changed in the data rows, use of searched updates avoids the possibility of failing to **fetch** matching rows due to the choice of join order. Note searched updates to a clustered index key in an allpages-locked table implicitly close the cursor.
- Note that the ANSI SQL 92 entry-level specification does not allow updates to join columns using cursors, so the behavior is implementation specific. Applications designed to be portable across different database software must take individual implementations into account.

18 Transactions: Maintaining Data Consistency and Recovery

A **transaction** groups a set of 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:

For more information about transactions, see the *Performance and Tuning Guide*.

How Transactions Work

Adaptive 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.

However, consider the following scenario: Lee needs to make a series of data retrievals and modifications to the *authors*, *titles*, and *titleauthors* tables. As she is doing so, Lil begins to update the *titles* table. Lil's updates could cause inconsistent results with the work that Lee is doing. To prevent this from happening, Lee can group her statements into a single transaction, which locks Lil out of the portions of the tables that Lee is working on. This allows Lee to complete her work based on accurate data. After she completes her table updates, Lil's updates can take place.

You can use the following commands to create transactions:

- **begin transaction** – marks the beginning of the transaction block. Its syntax is:

  ```
  begin {transaction | tran} [transaction_name]
  ```

where *transaction_name* is the name assigned to the transaction. It must conform to the rules for identifiers. Use transaction names only on the outermost pair of nested **begin** / **commit** or **begin** / **rollback** statements.

- **save transaction** – marks a savepoint within a transaction:

  ```
  save {transaction | tran} savepoint_name
  ```

  where *savepoint_name* is the name assigned to the savepoint. It must conform to the rules for identifiers.

- **commit** – commits the entire transaction:

  ```
  commit [transaction | tran | work]
  [transaction_name]
  ```

- **rollback** – rolls a transaction back to a savepoint or to the beginning of a transaction:

  ```
  rollback [transaction | tran | work]
  [transaction_name | savepoint_name]
  ```

For example, user Lee sets out to change the royalty split for two authors of *The Gourmet Microwave*. Since the database would be inconsistent between the two updates, they must be grouped into a transaction, as shown in the following example:

```
begin transaction royalty_change

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 percentchanged

/* After updating the royaltyper entries for
** the two authors, insert the savepoint
** percentchanged, then determine how a 10%
** increase in the book's price would affect
** the authors' royalty earnings. */
```

```
update titles
set price = price * 1.1
where title = "The Gourmet Microwave"

select (price * total_sales) * royaltyper
from titles, titleauthor
where title = "The Gourmet Microwave"
and titles.title_id = titleauthor.title_id

/* The transaction is rolled back to the savepoint
** with the rollback transaction command. */

rollback transaction percentchanged

commit transaction
```

Transactions allow Adaptive 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, Adaptive Server allows 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. See “Selecting the Transaction Mode and Isolation Level” on page 18-13 for more information.

Transactions and Consistency

In a multiuser environment, Adaptive 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, the results of the query would be ambiguous.

Adaptive 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.

Transactions and Recovery

A transaction is both a unit of work and a unit of recovery. The fact that Adaptive Server handles single-step change requests as transactions means that the database can be recovered completely in case of failure.

Adaptive Server's recovery time is measured in minutes and seconds. 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 18-34.

➤ ***Note***

Grouping large numbers of Transact-SQL commands into one long-running transaction may affect recovery time. If Adaptive Server fails before the transaction commits, recovery takes longer, because Adaptive Server must undo the transaction.

Using Transactions When Component Integration Services Is Enabled

If you are using remote database with Component Integration Services, there are a few differences in the way transactions are handled. See the *Component Integration Services User's Guide* for more information.

Using Transactions with Distributed Transaction Management (DTM) Features

If you have purchased and installed Adaptive Server DTM features, transactions that update data in multiple servers can also benefit from transactional consistency. See *Using Adaptive Server Distributed Transaction Features* for more information.

Using Transactions

The **begin transaction** and **commit transaction** commands tell Adaptive Server to process any number of individual commands as a single unit. The **rollback transaction** command 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.

Transactions give you control over transaction management. In addition to grouping SQL statements to behave as a single unit, they improve performance, since system overhead is incurred once per transaction, rather than once for each individual command.

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.

Allowing Data Definition Commands in Transactions

You can use certain data definition language commands in transactions by setting the **ddl in tran** database 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*. To check the current settings of **ddl in tran**, use **sp_helpdb**.

◆ ***WARNING!***

Be careful when using data definition commands. 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.

Avoid using data definition language commands on *tempdb* within transactions; doing so can slow performance to a halt. Always leave ddl in tran set to false in *tempdb*.

To set **ddl in tran** to true, type the following command:

```
sp_dboption database_name,"ddl in tran", true
```

Then, execute the **checkpoint** command in that database.

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 transaction only if the **ddl in tran** option to **sp_dboption** is set to true:

Table 18-1: 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 transactions.

Never use the following commands inside a transaction:

Table 18-2: 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**		

System Procedures That Are Not Allowed in Transactions

You cannot use the following system procedures within transactions:

- **sp_helpdb**, **sp_helpdevice**, **sp_helpindex**, **sp_helpjoins**, **sp_helpserver**, **sp_lookup**, and **sp_spaceused** (because they create temporary tables)
- **sp_configure**
- System procedures that change the *master* database

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 Adaptive Server if the transaction is not currently active.

Rolling Back and Saving Transactions

If you must cancel a transaction before it commits—either because of some failure or because of a change by the user—you must undo all of its completed statements or procedures.

You can cancel or roll back a transaction with the **rollback transaction** command anytime 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. You can commit only certain portions of a batch by rolling back the undesired portion to a savepoint before committing the entire batch.

You can insert a savepoint by putting a **save transaction** command in 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
    statements                  Group A
    save tran mytran
        statements              Group B
    rollback tran mytran        Rolls back group B
    statements                  Group C
commit tran                     Commits groups A and C
```

Until you issue a **commit transaction**, Adaptive Server considers all subsequent statements to be part of the transaction, unless it encounters another **begin transaction** statement. At that point, Adaptive Server considers all subsequent statements to be part of the new, nested transaction. Nested transactions are described under "Nested Transactions" on page 18-10.

rollback transaction or **save transaction** does not affect Adaptive Server and does not return an error message if the transaction is not currently active.

You can also use **save transaction** to create transactions in stored procedures or triggers in such a way that they can be rolled back without affecting batches or other procedures. For example:

```
create proc myproc as
begin tran
save tran mytran
statements
if ...
    begin
        rollback tran mytran
    /*
    ** Rolls back to savepoint.
    */
        commit tran
    /*
    ** This commit needed; rollback to a savepoint
    ** does not cancel a transaction.
    */
   end
else
commit tran
    /*
    ** Matches begin tran; either commits
    ** transaction (if not nested) or
    ** decrements nesting level.
    */
```

Unless you are rolling back to a savepoint, use transaction names only on the outermost pair of **begin** / **commit** or **begin** / **rollback** statements.

◆ ***WARNING!***

Transaction names are ignored, or can cause errors, when used in nested transaction statements. If you are using transactions in stored procedures or triggers that could be called from within other transactions, do not use transaction names.

Checking the State of Transactions

The global variable *@@transtate* keeps track of the current state of a transaction. Adaptive Server determines what state to return by keeping track of any transaction changes after a statement executes. *@@transtate* may contain the following values:

Table 18-3: @@transtate values

Value	Meaning
0	Transaction in progress. A 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.

Adaptive Server does not clear *@@transtate* after every statement. 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)
```

The 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)
```

Adaptive Server does not clear *@@transtate* after every statement. It changes *@@transtate* only in response to an action taken by a transaction. Syntax and compile errors do not affect the value of *@@transtate*.

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. Adaptive Server does not commit the

transaction until the **commit transaction** that matches the outermost **begin transaction** is issued. Normally, this transaction "nesting" occurs as stored procedures or triggers that contain **begin** / **commit** pairs call each other.

The *@@trancount* global variable 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 unless *@@trancount* equals 0.

For example, the following nested groups of statements are not committed by Adaptive 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 rolls back to the outermost **begin transaction** statement and cancels the transaction.

Example of a Transaction

This example shows how a 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 the Transaction Mode and Isolation Level

Adaptive Server provides the following options to support SQL standards-compliant transactions:

- The **transaction mode** lets you set whether transactions begin with or without an implicit **begin transaction** statement.
- The **isolation level** refers to the degree to which data can be accessed by other users during a transaction.

Set these options at the beginning of every session that requires SQL standards-compliant transactions.

The following sections describe these options in more detail.

Choosing a Transaction Mode

Adaptive Server supports the following transaction modes:

- 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**.
- The default mode, called **unchained** mode or Transact-SQL mode, requires explicit **begin transaction** statements paired with **commit transaction** or **rollback transaction** statements to complete the transaction.

You can set either mode using the **chained** option of the **set** command. However, do 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.

The SQL standards require every SQL data-retrieval and data-modification statement to occur inside a transaction, using chained mode. 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:

```
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 to **off**. The default transaction mode is unchained.

In chained transaction mode, Adaptive 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 ("9906", null, null, null)
begin transaction
delete from publishers where pub_id = "9906"
rollback transaction
```

In unchained transaction 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**.

All application programs and ad hoc user queries should know their current transaction mode. Which transaction mode you use depends on whether or not a particular query or application requires compliance to the SQL standards. Applications that use chained transactions (for example, the Embedded SQL precompiler) should set chained mode at the beginning of each session.

Transaction Modes and Nested Transactions

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 ("9907", null, null, null)
    insert into publishers
        values ("9908", 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.

Finding the Status of the Current Transaction Mode

You can check the global variable *@@tranchained* to determine Adaptive Server's current transaction mode. **select** *@@tranchained* returns 0 for unchained mode or 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 – ensures that data written by one transaction represents the actual data. It 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 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. This is the default isolation level supported by Adaptive Server.
- 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.

  Adaptive Server supports this level for data-only-locked tables. It is not supported for allpages-locked tables.
- Level 3 – ensures that data read by one transaction is valid until the end of that transaction, hence preventing **phantoms**.

Adaptive Server supports this level through the **holdlock** keyword of the **select** statement, which applies a read-lock on the specified data. 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.

You can set the isolation level for your session by 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. For example:

```
set transaction isolation level 0
```

Default Isolation Levels for Adaptive Server and SQL92

By default, the Adaptive Server 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 Adaptive Server to 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 Adaptive 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 Adaptive Server default isolation level:

```
set transaction isolation level 1
```

Dirty Reads

Applications that are not impacted by dirty reads may have 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.

Scans 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.

Because scans at isolation level 0 do not acquire any read locks, it is possible that the result set of a level 0 scan may change while the scan is in progress. If the scan position is lost due to changes in the underlying table, a unique index is required to restart the scan. In the absence of a unique index, the scan may be aborted.

By default, a unique index is required for a level 0 scan on a table that does not reside in a read-only database. You can override this requirement by forcing Adaptive Server to choose a nonunique index or a table scan, as follows:

```
select * from table_name (index table_name)
```

Activity on the underlying table may abort the scan before completion.

Repeatable Reads

A transaction performing repeatable reads locks all rows or pages read during the transaction. After one query in the transaction has read rows, no other transaction can update or delete the rows until the repeatable reads transaction completes. However, repeatable-reads transactions do not provide phantom protection by performing range locking, as serializable transactions do. Other transactions can insert values that can be read by the repeatable-reads transaction and can update rows so that they match the search criteria of the repeatable-reads transaction.

A transaction performing repeatable reads locks all rows or pages read during the transaction. After one query in the transaction has read rows, no other transaction can update or delete the rows until the repeatable reads transaction completes. However, repeatable-reads transactions do not provide phantom protection by performing range locking, as serializable transactions do. Other

transactions can insert values that can be read by the repeatable-reads transaction and can update rows so that they match the search criteria of the repeatable-reads transaction.

➤ ***Note***

Transaction isolation level 2 is only supported for data-only-locked tables. If you use transaction isolation level 2 (repeatable reads) on allpages-locked tables, isolation level 3 (serializable reads) is also enforced.

To enforce repeatable reads at a session level, use:

```
set transaction isolation level 2
```

or

```
set transaction isolation level repeatable read
```

To enforce transaction isolation level 2 from a query, use:

```
select title_id, price, advance
from titles
at isolation 2
```

or

```
select title_id, price, advance
from titles
at isolation repeatable read
```

Transaction isolation level 2 is supported only at the transaction level. You cannot use the **at isolation** clause in a **select** or **readtext** statement to set the isolation level of a query to 2. See "Changing the Isolation Level for a Query" on page 18-19.

For more information on transaction isolation levels, see the *Performance and Tuning Guide*.

Finding the Status of the Current Isolation Level

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 **at isolation** clause supports isolation levels 0, 1, and 3. It does not support isolation level 2. The **read uncommitted**, **read committed**, and **serializable** options of **at isolation** represent isolation levels as listed 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. Adaptive Server returns a syntax error if you use **at isolation**:

- 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 SQL-92 standard defines **read uncommitted**, **read committed**, and **serializable** as options for **at isolation** and **set transaction isolation level.** A Transact-SQL extension also allows you to specify 0, 1, or 3, but not 2,

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 **noholdlock** or **shared** in a query that also specifies **at isolation read uncommitted**. (If you specify **holdlock** and isolation level 0 in a query, Adaptive Server issues a warning and ignores the **at isolation** clause.) 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**. For more information about isolation levels and locking, see the *Performance and Tuning Guide*.

Isolation Level Precedences

The following describes the precedence rules as they apply to the different methods of defining isolation levels:

1. The **holdlock**, **noholdlock**, and **shared** keywords take precedence over the **at isolation** clause and **set transaction isolation level** option, except in the case of isolation level 0. For example:

   ```
   /* This query executes at isolation level 3 */
   select *
       from titles holdlock
       at isolation read committed

   create view authors_nolock
       as select * from authors noholdlock
   set transaction isolation level 3
   /* This query executes at isolation level 1 */
   select * from authors_nolock
   ```

2. The **at isolation** clause takes precedence over the **set transaction isolation level** option. For example:

   ```
   set transaction isolation level 2
   /* executes at isolation level 0 */
   select * from publishers
       at isolation read uncommitted
   ```

   You cannot use the **read uncommitted** option of **at isolation** in the same query as the **holdlock**, **noholdlock**, and **shared** keywords.

3. The **transaction isolation level 0** option of the **set** command takes precedence over the **holdlock**, **noholdlock**, and **shared** keywords. For example:

```
set transaction isolation level 0
/* executes at isolation level 0 */
select *
    from titles holdlock
```

Adaptive Server issues a warning before executing the above query.

Cursors and Isolation Levels

Adaptive Server provides three isolation levels for cursors:

- Level 0 – Adaptive Server uses no locks on base table pages that contain a row representing a current cursor position. Cursors acquire no read locks for their scans, so they do not block other applications from accessing the same data. However, cursors operating at this isolation level are not updatable, and they require a unique index on the base table to ensure the accuracy of their scans.
- Level 1 – Adaptive Server uses a shared or update lock on base table pages that contain a row representing a current cursor position. The page remains locked until the current cursor position moves off the page (as a result of **fetch** statements), or the cursor is closed. If an index is used to search the base table rows, it also applies shared or update locks to the corresponding index pages. This is the default locking behavior for Adaptive Server.
- Level 3 – Adaptive Server uses a shared or update lock on any base table pages that have been read in a transaction on behalf of the cursor. In addition, the locks are held until the transaction ends, as opposed to being released when the data page is no longer needed. The **holdlock** keyword applies this locking level to the base tables, as specified by the query on the tables or views.

Isolation level 2 is not supported for cursors.

Besides using **holdlock** for isolation level 3, you can use **set transaction isolation level** to specify any of the four isolation levels for your session. When you use **set transaction isolation level,** any cursor you open uses the specified isolation level, unless the transaction isolation level is set at 2. In this case, the cursor uses isolation level 3. You can also use the **select** statement's **at isolation** clause to specify isolation level 0, 1, or 3 for a specific 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**), Adaptive 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.

Adaptive Server determines a cursor's isolation level when you open the cursor (not when you declare it), 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.

Adaptive Server compiles the cursor's query when you declare it. This compilation process is different for isolation level 0 as compared to isolation levels 1 or 3. If you declare a **language** or **client** cursor in a transaction with isolation level 1 or 3, opening it in a transaction at isolation level 0 causes an error.

For example:

```
set transaction isolation level 1

declare publishers_crsr cursor
    for select *
    from publishers

open publishers_crsr     /* no error */

fetch publishers_crsr

close publishers_crsr

set transaction isolation level 0

open publishers_crsr     /* error */
```

Stored Procedures and Isolation Levels

The Sybase system procedures always operate at isolation level 1, regardless of the isolation level of the transaction or session. 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, Adaptive Server sets the trigger's isolation level to 1 before executing its first statement.

Compliance with SQL Standards

To get transactions that comply with SQL standards, you must **set** the **chained** and **transaction isolation level 3** options at the beginning of every application that changes the mode and isolation level for subsequent transactions. If your application uses cursors, you must also set the **close on endtran** option. Each of these options is described later.

Using the *lock table* Command to Improve Performance

The **lock table** command allows you to explicitly request that before a table is accessed, a table lock be put on it for the duration of a transaction. This is use useful when an immediate table lock may reduce the overhead of acquiring a large number of row or page locks and save overall locking time. Examples of such cases are:

- A table will be scanned more than once in the same transaction, and each scan may need to acquire many page or row locks.
- It is known that a scan will exceed a table's lock-promotion threshold and will therefore attempt to escalate to a table lock in any event.

  If a table lock is not explicity requested, a scan acquires page or row locks until it reaches the table's lock promotion threshold (see system procedure **sp_setpglockpromote**), at which point it tries to acquire a table lock.

Syntax of the *lock table* command

The syntax of the **lock table** command is:

```
lock table table_name in {share | exclusive } mode
      [ wait [no_of_seconds] | nowait ]
```

The **wait** / **nowait** option allows you to specify how long the command waits to acquire a table lock if it is blocked (see "Using the wait / nowait Options of the lock table Command" on page 18-24).

The following considerations apply to the use of the **lock table** command:

- The **lock table** command can only be issued within a transaction.
- You cannot use **lock table** on system tables.
- You can first use **lock table** to lock a table in **share** mode and then use it to upgrade the lock to **exclusive** mode.
- You can use separate **lock table** commands to lock multiple tables within the same transaction.
- Once a table lock is obtained, there is no difference between a table locked with the **lock table** command and a table locked through lock promotion without the **lock table** command.

Using the *wait/nowait* Options of the *lock table* Command

Use the **wait** option in the **lock table** command to specify the length of time the command waits to acquire a table lock. For example, the following command, inside a transaction, sets a wait period of 2 seconds for acquiring a table lock on the *titles* table:

```
lock table titles in share mode wait 2
```

If the wait time expires before a table lock is acquired, the transaction proceeds, and row or page locking is used, exactly as it would have been without the lock table command, and the following informational message (error number 12207) is generated:

```
Could not acquire a lock within the specified wait
period. COMMAND level wait...
```

For an example of error handling during transactions, see **lock table** in the *Adaptive Server Reference Manual*.

➤ ***Note***

If you use **lock table...wait** without specifying *no_of_seconds*, the command waits indefinitely for a lock.

You can set time limits on waiting for a lock at the session level and the system level. The wait period set with the **lock table** command overrides both of these, as described in the following section.

The **nowait** option is equivalent to the **wait** option with a 0-second wait: **lock table** either obtains a table lock immediately or generates the informational message given above. If the lock is not acquired, the

transaction proceeds as it would have without the **lock table** command.

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. Adaptive 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**.

Figure 18-1 demonstrates how nested transaction statements work within stored procedures:

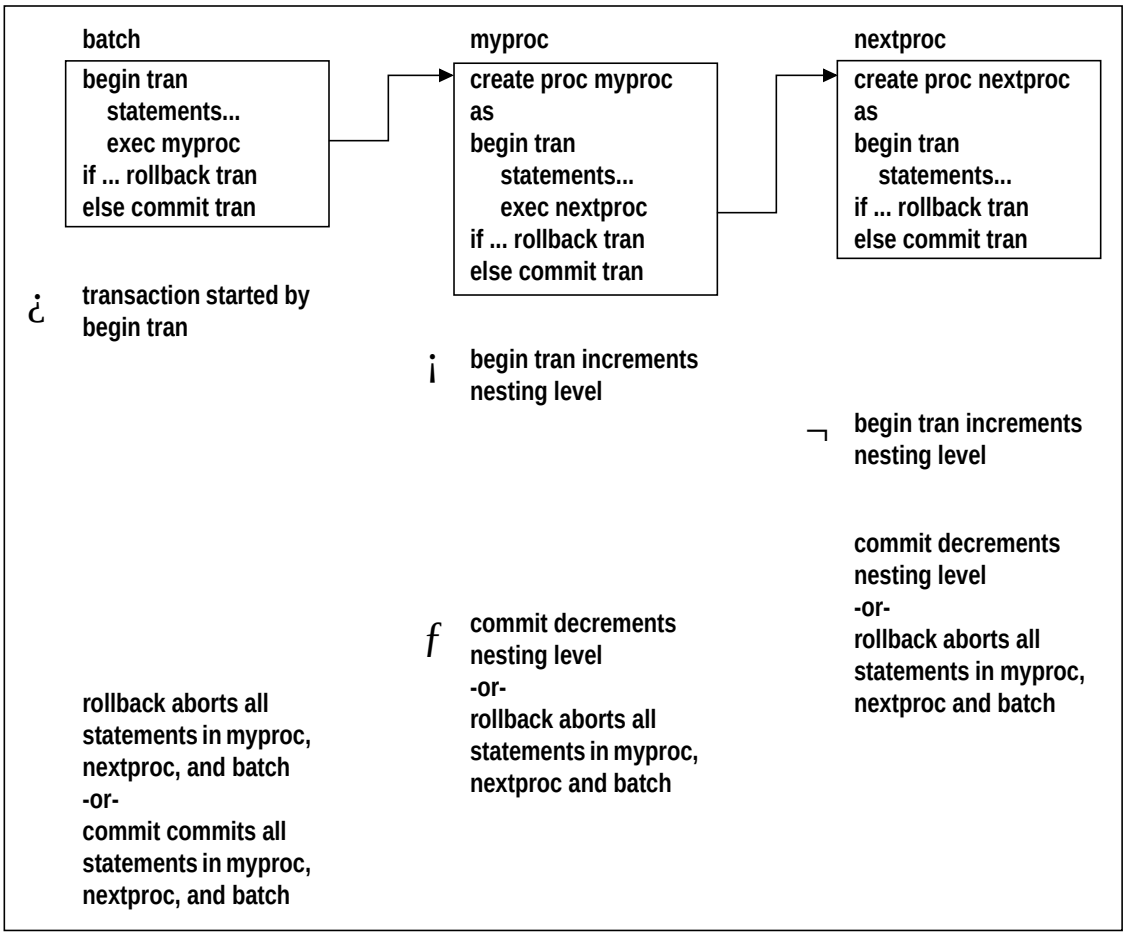


Figure 18-1: Nesting transaction statements

rollback transaction statements in stored procedures do not affect subsequent statements in the procedure or batch that originally called the procedure. Adaptive Server executes subsequent statements in the stored procedure or batch. However, **rollback transaction** statements in triggers abort the batch so that subsequent statements are not executed.

➤ *Note*

rollback statements in triggers: 1) roll back the transaction, 2) complete subsequent statements in the trigger, and 3) abort the batch so that subsequent statements in the batch 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. Adaptive 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 under "Transaction Modes and Stored Procedures" on page 18-30. Triggers are not affected by the current transaction mode, since they are called as part of a data modification statement.

Errors and Transaction Rollbacks

Errors that affect data integrity can affect the state of implicit or explicit transactions:

- Errors with severity levels of 19 or greater

  Since these errors terminate the user connection to the server, any errors of level 19 or greater that occur while a user transaction is in progress abort the transaction and roll back all statements to the outermost **begin transaction**. Adaptive Server always rolls back any uncommitted transactions at the end of a session.

- Errors in data modification commands that affect data integrity
  - Arithmetic overflow and divide-by-zero errors (effects on transactions can be changed with the **set arithabort arith_overflow** command)
  - Permissions violations
  - Rules violations
  - Duplicate key violations

The following table summarizes how **rollback** affects Adaptive Server processing in several different contexts (such as within a transaction, stored procedure, or trigger):

Table 18-4: How rollback affects processing

Context	Effects of *rollback*
Transaction only	All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, **rollback** affects all of those batches. Any commands issued after the rollback are executed.
Stored procedure only	None.
Stored procedure in a transaction	All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, **rollback** affects all those batches. Any commands issued after the rollback are executed. Stored procedure produces error message 266: "Transaction count after EXECUTE indicates that a COMMIT or ROLLBACK TRAN is missing."
Trigger only	Trigger completes, but trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger in a transaction	Trigger completes, but trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, **rollback** affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger	Inner trigger completes, but all trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger in a transaction	Inner trigger completes, but all trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, **rollback** affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.

In stored procedures and triggers, the number of **begin transaction** statements must match the number of **commit** statements. A procedure or trigger that contains unpaired **begin** / **commit** statements produces a warning message when it is executed. This also applies to

stored procedures that use chained mode: The first statement that implicitly begins a transaction must have a matching **commit**.

With duplicate key errors and rules violations, the trigger completes (unless there is also a **return** statement), and statements such as **print**, **raiserror**, or remote procedure calls are performed. Then, the trigger and the rest of the transaction are rolled back, and the rest of the batch is aborted. Note that remote procedure calls executed from inside a normal SQL transaction (not using the DB-Library two-phase commit) are not rolled back by a **rollback** statement.

The following table summarizes how a rollback caused by a duplicate key error or a rules violation affects Adaptive Server processing in several different contexts:

Table 18-5: Rollbacks caused by duplicate key errors/rules violations

Context	Effects of Duplicate Key or Rules Error Rollback
Transaction only	Current command is aborted. Previous commands are not rolled back, and subsequent commands are executed.
Stored procedure only	Same as above.
Stored procedure in a transaction	Same as above.
Trigger only	Trigger completes, but trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger in a transaction	Trigger completes, but trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, the rollback affects all of those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger	Inner trigger completes, but all trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger in a transaction	Inner trigger completes, but all trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, the rollback affects all of those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.

Table 18-5: Rollbacks caused by duplicate key errors/rules violations (continued)

Context	Effects of Duplicate Key or Rules Error Rollback
Trigger with rollback followed by an error in the transaction	Trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, the rollback affects all of those batches. Trigger continues and gets duplicate key or rules error. Normally, the trigger rolls back effects and continues, but trigger effects are not rolled back in this case. After the trigger completes, any remaining commands in the batch are not executed. Processing resumes at the next batch.

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 ("9996", 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 Adaptive 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 mode. The exceptions to that rule are Sybase system procedures (except for **sp_procxmode**), that can be invoked by sessions using any transaction mode. (For information about **sp_proxmode**, see “Setting Transaction Modes for Stored Procedures” on page 18-31.) If no transaction is active when you execute a system procedure, Adaptive Server turns off chained mode for the duration of the procedure. Before returning, it resets the mode its original setting.

Adaptive 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 that use one mode to invoke transactions that use the other mode. A stored procedure tagged as "chained" is not executable in sessions using unchained transaction mode, and vice versa.

Triggers are executable in any transaction mode. Since they are always called as part of a data modification statement, either they are part of a chained transaction (if the session uses chained mode) or they maintain their current transaction mode.

◆ ***WARNING!***

When using transaction modes, be aware of the effects each setting can have on your applications.

Setting Transaction Modes for Stored Procedures

Use **sp_procxmode to** display or change the transaction mode of stored procedures. For example, to change the transaction mode for the stored procedure *byroyalty* to "chained", enter:

```
sp_procxmode byroyalty, "chained"
```

sp_procxmode "anymode" lets stored procedures run under either chained or unchained 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                   Any Mode
discount_proc               Unchained
history_proc                Unchained
insert_sales_proc           Unchained
insert_salesdetail_proc     Unchained
storeid_proc                Unchained
storename_proc              Unchained
title_proc                  Unchained
titleid_proc                Unchained

(9 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, Adaptive Server does not change a cursor's state (open or closed) when a transaction ends through a **commit** or **rollback**. 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, Adaptive Server provides the **close on endtran** option of the **set** command. In addition, if you set chained mode to **on**, Adaptive Server starts a transaction when you open a cursor and 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 to **on**, the cursor's state changes from open to closed after the **commit**. This allows the cursor to be reopened.

➤ ***Note***

Since client application buffer rows are returned through cursors, and allow users to scroll within those buffers, those client applications should not scroll backward after a transaction aborts. The rows in a client cache may become invalid because of a transaction rollback (unknown to the client) that is enforced by the **close on endtran** option or the chained mode.

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 you use 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 may also continue to acquire locks as it fetches additional rows.

The following rules define the behavior of updates through a cursor with regard to transactions:

- If an update occurs within an explicit transaction, the update is considered part of the transaction. If the transaction commits, any updates included with the transaction also commit. If the transaction aborts, any updates included with the transaction are rolled back. Updates through the same cursor that occurred outside the aborted transaction are not affected.
- If updates through a cursor occur within an explicit (and client-specified) transaction, Adaptive Server does not commit them when the cursor is closed. It commits or rolls back pending updates only when the transaction associated with that cursor ends.
- A transaction commit or abort has no effect on SQL cursor statements that do not manipulate result rows, such as **declare cursor**, **open cursor**, **close cursor**, **set cursor rows**, and **deallocate cursor.** For example, if the client opens a cursor within a transaction, and the transaction aborts, the cursor remains open after the abort (unless the **close on endtran** is set or the chained mode is used).

Issues to Consider When Using Transactions

You should consider the following issues when using transactions in your applications:

- A **rollback** statement, without a transaction or savepoint name, always rolls back statements to the outermost **begin transaction** (explicit or implicit) statement and cancels the transaction. If there is no current transaction when you issue **rollback**, the statement has no effect.

  In triggers or stored procedures, **rollback** statements, without transaction or savepoint names, roll back all statements to the outermost **begin transaction** (explicit or implicit).
- **rollback** does not produce any messages to the user. If warnings are needed, use **raiserror** or **print** statements.
- Grouping a large number of Transact-SQL commands into one long-running transaction may affect recovery time. If Adaptive Server fails during a long transaction, recovery time increases, since Adaptive Server must first undo the entire transaction.
- You can refer to as many as 16 databases within a transaction. However, Adaptive Server may refer to internal databases to

process a transaction, so the number of databases you can actually refer to may be lower. Adaptive Server displays an error message if you exceed the open databases limit.

- A remote procedure call (RPC) may be executed independently from any transaction in which it is included. In a standard transaction (one that does not use Open Client DB-Library / C two-phase commit or Adaptive Server Distributed Transaction Management features), commands executed via an RPC by a remote server are not rolled back with **rollback** and do not depend on **commit** to be executed.
- Transactions cannot span more than one connection between a client application and a server. For example, a DB-Library / C application cannot group SQL statements in a transaction across multiple open DBPROCESSes.

Backup and Recovery of Transactions

Every change to a database, whether it is the result of a single **update** statement or a grouped set of SQL statements, is 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 a user decision to cancel the 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 Adaptive Server fails, undoing the changes may require as much time as the transactions have been running. Such cases include transactions that do not contain a **commit transaction** or **rollback transaction** to match a **begin transaction**. This prevents Adaptive Server from writing any changes and increases recovery time.

Adaptive 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 the **ss_oper** role 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, Adaptive Server handles all aspects of the recovery process. Adaptive Server also 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 about backup and recovery, see the *Adaptive Server Reference Manual* and the *System Administration Guide*.

19 Locking Commands and Options

Adaptive Server provides a wide range of commands and options for providing locking. This chapter reviews the types of locking available and how they are invoked through Transact SQL. It includes the following sections:

Setting a Time Limit on Waiting for Locks

Adaptive Server allows you to specify a lock wait period that determines how long a command will wait to acquire locks:

- You can specify a time limit on waiting to obtain a table lock with the **wait** / **nowait** option of the **lock table** command.
- During a session, you can use the **set lock** command to specify a lock wait period for all subsequent commands issued during the session
- Within a stored procedure, you can use the **set lock** command to specify a lock wait period for all subsequent commands issued within the stored procedure
- You can use the **lock wait period** option of **sp_configure** to set a Server-wide lock wait period.

wait/*nowait* Option of the *lock table* Command

Within a transaction, the **lock table** command allows you to request a table lock on a table without waiting for the command to acquire enough row-level or page-level locks to escalate to a table lock.

The **lock table** command contains a **wait** / **nowait** option that allows you to specify the lenth of time the command will wait until operations in other transactions relinquish any locks they have on the target table.

The syntax for the lock table command is:

```
lock table table_name in {share | exclusive } mode
      [ wait [no_of_seconds] | nowait ]
```

The following command, inside a transaction, sets a wait period of 2 seconds for acquiring a table lock on the *titles* table:

```
lock table titles in share mode wait 2
```

If the wait time expires before a table lock is acquired, the transaction proceeds, and row or page locking is used, exactly as it would have been without the **lock table** command, and the following informational message (error number 12207) is generated:

```
Could not acquire a lock within the specified wait
period. COMMAND level wait...
```

For a code example of handling this error message during a transaction, see **lock table** in the *Reference Manual*.

➤ ***Note***

If you use **lock table...wait** without specifying *no_of_seconds*, the command waits indefinitely for a lock.

You can set time limits on waiting for a lock at the session level and the system level, as described in the following sections. The wait period set with the **lock table** command overrides both of these

The **nowait** option is equivalent to the **wait** option with a 0-second wait: **lock table** either obtains a table lock immediately or generates the informational message given above. If the lock is not acquired, the transaction proceeds as it would have without the **lock table** command.

You can use the new **set lock** command at either the session level or within a stored procedure to control the length of time a task will wait to acquire locks.

A System Administrator can use the new **sp_configure** option, **lock wait period**, to set a server-wide time limit on acquiring locks.

Setting a Session-Level Lock-Wait Limit

You can use the **set lock wait** command to control the length of time that a command in a session or in a stored procedure will wait to acquire locks. The syntax is:

```
set lock { wait no_of_seconds | nowait }
```

The *no_of_seconds* parameter is entered as an integer. Thus, the following example sets a session-level time limit of 5 seconds on waiting for locks:

```
set lock wait 5
```

With one exception, if the **set lock wait** period expires before a command acquires a lock, the command fails, the transaction containing it is rolled back, and the following error message is generated:

```
Msg 12205, Level 17, State 2:
Server 'sagan', Line 1:
Could not acquire a lock within the specified wait
period. SESSION level wait period=300 seconds,
spid=12, lock type=shared page, dbid=9,
objid=2080010441, pageno=92300, rowno=0. Aborting
the transaction.
```

The exception to this occurs when the **lock table** command in a transaction sets a longer wait period than **set lock wait.** In this case, the transaction uses the **lock table** wait period before timing out, as described in the preceding section.

The **set lock nowait** option is equivalent to the **set lock wait** option with a 0-second wait. If a command other than the **lock table** command is not able to obtain a requested lock immediately, the command fails, its transaction is rolled back, and the preceding error message is generated.

If both a server-wide lock-wait limit and a session-level lock-wait limit are set, the session-level limit takes precedence. If no session-level wait period is set with **set lock wait**, the server-level wait period is used.

➤ ***Note***

Stored procedures do not use a wait period set at either the server level or the session level. The wait period for commands in a stored procedure is unbounded, unless you explicitly specify a time limit by using the **set lock wait** command inside the stored procedure.

Setting a Server-Wide lock-wait Limit

A System Administrator can configure a server-wide lock-wait limit with the configuration parameter **lock wait period**. The syntax is:

```
sp_configure "lock wait period" [, no_of_seconds]
```

If the lock-wait period expires before a command acquires a lock, unless there is an overriding **set lock wait** or **lock table** wait period, the command fails, the transaction containing it is rolled back, and the following error message is generated:

```
Msg 12205, Level 17, State 2:
Server 'wiz', Line 1:
Could not acquire a lock within the specified wait
period. SERVER level wait period=300 seconds,
spid=12, lock type=shared page, dbid=9,
objid=2080010441, pageno=92300, rowno=0. Aborting
the transaction.
```

A time limit entered through **set lock wait** or **lock table wait** overrides a server-level lock-wait period. Thus, for example, if the server-level wait period is 5 seconds and the session-level wait period is 10 seconds, an **update** command waits 10 seconds to acquire a lock before failing and aborting its transaction.

The default server-level lock-wait period is effectively "wait forever." To restore the default after setting a time-limited wait, you can use **sp_configure** to set the value of **lock wait period** as follows:

```
sp_configure "lock wait period", 0, "default"
```

Information on the Number of lock-wait Timeouts

The **sp_sysmon** system procedure reports on the number of times tasks waiting for locks did not acquire the lock within the specified period. See the *Performance and Tuning Guide* for more information.

Readpast Locking for Queue Processing

Readpast locking is an option available for the **select** and **readtext** commands and the data modification commands **update**, **delete**, and **writetext**. It instructs a command to silently skip all incompatible locks it encounters, without blocking, terminating, or generating a message. It is primarily used when the rows of a table constitute a queue. In such a case, a number of tasks may access the table to process the queued rows, which could, for example, represent queued customers or customer orders. A given task is not concerned with processing a specific member of the queue, but with processing any available members of the queue that meet its selection criteria.

Readpast Syntax

The syntax for using readpast locking is:

```
{ select | update | delete } ...
       from tablename [ holdlock | noholdlock ]
           [ readpast ] [ shared ]
           ...
readtext [[database.]owner.]table_name.column_name
   text_pointer offset size
   [holdlock | noholdlock]  [shared] [readpast]
       ...
writetext [[database.]owner.]table_name.column_name
   text_pointer [readpast] [with log] data
```

Incompatible Locks During Readpast Queries

For **select** and **readtext** commands, incompatible locks are exclusive locks. Therefore, **select** and **readpast** commands can access any rows or pages on which shared or update locks are held.

For **delete**, **update** and **writetext** commands, any type of page or row lock is incompatible, so that:

- All rows with shared, update, or exclusive row locks are skipped in datarows-locked tables, and
- All pages with shared, update, or exclusive locks are skipped in datapages-locked tables.

All commands specifying **readpast** block if there is an exclusive table lock, except **select** commands executed at transaction isolation level 0.

Allpages-Locked Tables and Readpast Queries

If the **readpast** option is specified for an allpages-locked table, the **readpast** option is ignored. The command operates at the isolation level specified for the command or session:

- If the isolation level is 0, dirty reads are performed, and the command returns values from locked rows, and does not block.
- If the isolation level is 1 or 3, the command blocks when pages with incompatible locks must be read.

Effects of Isolation Levels *select* Queries with Readpast

Readpast locking is designed to be used at transaction isolation level 1 or 2.

Session-Level Transaction Isolation Levels and Readpast

For data-only-locked tables, the effects of **readpast** on a table in a **select** command are shown in Table 19-1.

Table 19-1: Session-level isolation level and the use of readpast

Session Isolation Level	Effects
0, **read uncommitted** (dirty reads)	**readpast** is ignored, and rows containing uncommitted transactions are returned to the user. A warning message is printed.
1, **read committed**	Rows or pages with incompatible locks are skipped;, no locks are held on the rows or pages read
2, **repeatable read**	Rows or pages with incompatible locks skipped; shared locks are held on all rows or pages that are read until the end of the statement or transaction.
3, **serializable**	**readpast** is ignored, and the command executes at level 3. The command blocks on any rows or pages with incompatible locks.

Query-Level Isolation Levels and Readpast

If **select** commands that specify **readpast** also include any of the following clauses, the commands fail and display error messages:

- The **at isolation** clause, specifying **0** or **read uncommitted**
- The **at isolation** clause, specifying **3** or **serializable**
- The **holdlock** keyword on the same table

If a **select** query that specifies **readpast** also specifies **at isolation 2** or **at isolation repeatable read**, shared locks are held on the **readpast** table or tables until the statement or transaction completes.

readtext commands that include **readpast** and that specify **at isolation read uncommitted** automatically run at isolation level 0 after issuing a warning message.

Data Modification Commands with *readpast* and Isolation Levels

If the transaction isolation level for a session is 0, the **delete**, **update**, and **writetext** commands that use **readpast** do not issue warning messages.

- For datapages-locked tables, these commands modify all rows on all pages that are not locked with incompatible locks.
- For datarows-locked tables, they affect all rows that are not locked with incompatible locks.

If the transaction isolation level for a session is 3 (serializable reads), the **delete**, **update**, and **writetext** commands that use **readpast** automatically block when they encounter a row or page with an incompatible lock.

At transaction isolation level 2 (serializable reads), the **delete**, **update**, and **writetext** commands:

- Modify all rows on all pages that are not locked with incompatible locks.
- For datarows-locked tables, they affect all rows that are not locked with incompatible locks.

text and *image* Columns and Readpast

If a **select** command with the **readpast** option encounters a text column that has an incompatible lock on it, readpast locking retrieves the row, but returns the text column with a value of **null**. No distinction is made, in this case, between a text column containing a null value and a null value returned because the column is locked.

If an **update** command with the **readpast** option applies to two or more text columns, and the first text column checked has an incompatible lock on it, readpast locking skips the row. If the column does not have an incompatible lock, the command acquires a lock and modifies the column. Then, if any subsequent text column in the row has an incompatible lock on it, the command blocks until it can obtain a lock and modify the column.

A **delete** command with the **readpast** option skips the row if any of the text columns in the row have an incompatible lock.

Readpast-Locking Examples

The following examples illustrate readpast locking.

To skip all rows that have exclusive locks on them:

```
select * from titles readpast
```

To update only rows that are not locked by another session:

```
update titles
    set price = price * 1.1
    from titles readpast
```

To use readpast locking on the *titles* table but not on the *authors* or *titleauthor* table:

```
select *
    from titles readpast, authors, titleauthor
    where titles.title_id = titleauthor.title_id
    and authors.au_id = titleauthor.au_id
```

To delete only rows that are not locked in the *stores* table, but to allow the scan to block on the *authors* table:

```
delete stores from stores readpast, authors
where stores.city = authors.city
```

A The *pubs2* Database

This appendix describes the sample database *pubs2*. This database contains the tables *publishers, authors, titles, titleauthor, au_pix, salesdetail, sales, stores, discounts, roysched,* and *blurbs.*

The *pubs2* database also contains primary and foreign keys, rules, defaults, views, triggers, and stored procedures.

A diagram of the *pubs2* database appears in Figure A-1 on page A-27.

For information about installing *pubs2*, see the configuration guide for your platform.

Tables in the *pubs2* Database

The following tables describe each *pubs2* table. Each column header specifies the column name, its datatype (including any user-defined datatypes), and its null or not null status. The column header also specifies any defaults, rules, triggers, and indexes that affect the column.

publishers Table

publishers is defined as follows:

```
create table publishers
(pub_id char(4) not null,
pub_name varchar(40) not null,
city varchar(20) null,
state char(2) null)
```

Its primary key is *pub_id*:

```
sp_primarykey publishers, pub_id
```

Its *pub_idrule* rule is defined as:

```
create rule pub_idrule
as @pub_id in
("1389", "0736", "0877", "1622", "1756")
or @pub_id like "99[0-9][0-9]"
```

Table A-1 lists the contents of *publishers*:

Table A-1: publishers table

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

authors Table

authors is defined as follows:

```
create table authors
(au_id id not null,
au_lname varchar(40) not null,
au_fname varchar(20) not null,
phone char(12) not null,
address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode char(10) null)
```

Its primary key is *au_id*:

```
sp_primarykey authors, au_id
```

Its nonclustered index for the *au_lname* and *au_fname* columns is defined as:

```
create nonclustered index aunmind
on authors (au_lname, au_fname)
```

The *phone* column has the following default:

```
create default phonedflt as "UNKNOWN"

sp_bindefault phonedft, "authors.phone"
```

The following view uses *authors*:

```
create view titleview
as
select title, au_ord, au_lname,
price, total_sales, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

Table A-2 lists the contents of *authors*:

Table A-2: authors table

au_id	au_-lname	au_ fname	phone	address	city	state	coun-try	post-alcode
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	USA	94025
213-46-8915	Green	Marjorie	510 986-7020	309 63rd St. #411	Oakland	CA	USA	94618
238-95-7766	Carson	Cheryl	510 548-7723	589 Darwin Ln.	Berkeley	CA	USA	94705
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14	San Jose	CA	USA	95128
274-80-9391	Straight	Dick	510 834-2919	5420 College Av.	Oakland	CA	USA	94609
341-22-1782	Smith	Meander	913 843-0462	10 Mississippi Dr.	Lawrence	KS	USA	66044
409-56-7008	Bennet	Abraham	510 658-9932	6223 Bateman St.	Berkeley	CA	USA	94705
427-17-2319	Dull	Ann	415 836-7128	3410 Blonde St.	Palo Alto	CA	USA	94301
472-27-2349	Gringlesby	Burt	707 938-6445	PO Box 792	Covelo	CA	USA	95428
486-29-1786	Locksley	Chastity	415 585-4620	18 Broadway Av.	San Francisco	CA	USA	94130
527-72-3246	Greene	Morningstar	615 297-2723	22 Graybar House Rd.	Nashville	TN	USA	37215
648-92-1872	Blotchet-Halls	Reginald	503 745-6402	55 Hillsdale Bl.	Corvallis	OR	USA	97330
672-71-3249	Yokomoto	Akiko	415 935-4228	3 Silver Ct.	Walnut Creek	CA	USA	94595
712-45-1867	del Castillo	Innes	615 996-8275	2286 Cram Pl. #86	Ann Arbor	MI	USA	48105
722-51-5454	DeFrance	Michel	219 547-9982	3 Balding Pl.	Gary	IN	USA	46403
724-08-9931	Stringer	Dirk	510 843-2991	5420 Telegraph Av.	Oakland	CA	USA	94609
724-80-9391	MacFeather	Stearns	510 354-7128	44 Upland Hts.	Oakland	CA	USA	94612
756-30-7391	Karsen	Livia	510 534-9219	5720 McAuley St.	Oakland	CA	USA	94609
807-91-6654	Panteley	Sylvia	301 946-8853	1956 Arlington Pl.	Rockville	MD	USA	20853

Table A-2: authors table (continued)

au_id	au_-lname	au_ fname	phone	address	city	state	coun-try	post-alcode
846-92-7186	Hunter	Sheryl	415 836-7128	3410 Blonde St.	Palo Alto	CA	USA	94301
893-72-1158	McBadden	Heather	707 448-4982	301 Putnam	Vacaville	CA	USA	95688
899-46-2035	Ringer	Anne	801 826-0752	67 Seventh Av.	Salt Lake City	UT	USA	84152
998-72-3567	Ringer	Albert	801 826-0752	67 Seventh Av.	Salt Lake City	UT	USA	84152

titles Table

titles is defined as follows:

```
create table titles
(title_id tid not null,
title varchar(80) not null,
type char(12) not null,
pub_id char(4) null,
price money null,
advance money null,
total_sales int null,
notes varchar(200) null,
pubdate datetime not null,
contract bit not null)
```

Its primary key is *title_id*:

```
sp_primarykey titles, title_id
```

Its *pub_id* column is a foreign key to the *publishers* table:

```
sp_foreignkey titles, publishers, pub_id
```

Its nonclustered index for the *title* column is defined as:

```
create nonclustered index titleind
on titles (title)
```

Its *title_idrule* is defined as:

```
create rule title_idrule
as
@title_id like "BU[0-9][0-9][0-9][0-9]" or
@title_id like "[MT]C[0-9][0-9][0-9][0-9]" or
@title_id like "P[SC][0-9][0-9][0-9][0-9]" or
@title_id like "[A-Z][A-Z]xxxx" or
@title_id like "[A-Z][A-Z]yyyy"
```

The *type* column has the following default:

```
create default typedflt as "UNDECIDED"
sp_bindefault typedflt, "titles.type"
```

The *pubdate* column has this default:

```
create default datedflt as getdate()
sp_bindefault datedflt, "titles.pubdate"
```

titles uses the following trigger:

```
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
```

The following view uses *titles*:

```
create view titleview
as
select title, au_ord, au_lname,
price, total_sales, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

Table A-3 lists the contents of *titles:*

Table A-3: titles table

title-_id	title	type	pub-_id	price	ad-vance	total_-sales	notes	pub-date	con-tract
BU1032	The Busy Executive's Database Guide	business	1389	19.99	5000.00	4095	An overview of available database systems with emphasis on common business applica-tions. Illustrated.	Jun 12, 1986	1

Table A-3: titles table (continued)

title-_id	title	type	pub-_id	price	ad-vance	total_-sales	notes	pub-date	con-tract
BU1111	Cooking with Computers: Surrepti-tious Balance Sheets	business	1389	11.95	5000.00	3876	Helpful hints on how to use your electronic resources to the best advantage.	Jun 9, 1988	1
BU2075	You Can Combat Computer Stress!	business	0736	2.99	10125.00	18722	The latest medical and psycho-logical techniques for living with the electronic office. Easy-to-under-stand explana-tions.	Jun 30, 1985	1
BU7832	Straight Talk About Computers	business	1389	19.99	5000.00	4095	Annotated analysis of what computers can do for you: a no-hype guide for the critical user.	Jun 22, 1987	1
MC2222	Silicon Valley Gastro-nomic Treats	mod_-cook	0877	19.99	0.00	2032	Favorite recipes for quick, easy, and elegant meals; tried and tested by people who never have time to eat, let alone cook.	Jun 9, 1989	1

Table A-3: titles table (continued)

title-_id	title	type	pub-_id	price	ad-vance	total_-sales	notes	pub-date	con-tract
MC3021	The Gourmet Microwave	mod_-cook	0877	2.99	15000.00	22246	Traditional French gourmet recipes adapted for modern micro-wave cooking.	Jun 18, 1985	1
PC1035	But Is It User Friendly?	popular_comp	1389	22.95	7000.00	8780	A survey of software for the naive user, focusing on the "friendli-ness" of each.	Jun 30, 1986	1
MC3026	The Psychology of Computer Cooking	UNDE-CIDED	0877	NULL	NULL	NULL	NULL	Jul 24, 1991	0
PC8888	Secrets of Silicon Valley	popular_comp	1389	20.00	8000.00	4095	Muckrak-ing reporting by two courag-eous women on the world's largest computer hardware and software manufact-urers.	Jun 12, 1987	1
PC9999	Net Etiquette	popular_comp	1389	NULL	NULL	NULL	A must-read for computer conferenc-ing debu-tantes!	Jul 24, 1996	0

Table A-3: titles table (continued)

title-_id	title	type	pub-_id	price	ad-vance	total_-sales	notes	pub-date	con-tract
PS1372	Computer Phobic and Non-Phobic Individ-uals: Behavior Variations	psycho-logy	0877	21.59	7000.00	375	A must for the specialist, this book examines the difference between those who hate and fear computers and those who think they are swell.	Oct 21,1990	1
PS2091	Is Anger the Enemy?	psycho-logy	0736	10.95	2275.00	2045	Carefully researched study of the effects of strong emotions on the body. Metabolic charts included.	Jun 15, 1989	1
PS2106	Life Without Fear	psycho-logy	0736	7.00	6000.00	111	New exercise, meditat-ion, and nutritional techniques that can reduce the shock of daily inter-actions. Popular audience. Sample menus included. Exercise video available separately.	Oct 5, 1990	1

Table A-3: titles table (continued)

title-_id	title	type	pub-_id	price	ad-vance	total_-sales	notes	pub-date	con-tract
PS3333	Prolonged Data Deprivation: Four Case Studies	psychology	0736	19.99	2000.00	4072	What happens when the data runs dry? Searching evaluations of information-shortage effects on heavy users.	Jun 12, 1988	1
PS7777	Emotional Security: A New Algorithm	psychology	0736	7.99	4000.00	3336	Protecting yourself and your loved ones from undue emotional stress in the modern world. Use of computer and nutritional aids emphasized.	Jun 12, 1988	1
TC3218	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	trad_cook	0877	20.95	7000.00	375	Profusely illustrated in color, this makes a wonderful gift book for a cuisine-oriented friend.	Oct 21, 1990	1

Table A-3: titles table (continued)

title-_id	title	type	pub-_id	price	ad-vance	total_-sales	notes	pub-date	con-tract
TC4203	Fifty Years in Bucking-ham Palace Kitchens	trad_cook	0877	11.95	4000.00	15096	More anecdotes from the Queen's favorite cook describing life among English royalty. Recipes, tech-niques, tender vignettes.	Jun 12, 1985	1
TC7777	Sushi, Anyone?	trad_cook	0877	14.99	8000.00	4095	Detailed instruc-tions on improving your position in life by learning how to make authentic Japanese sushi in your spare time. 5-10% increase in number of friends per recipe reported from beta test.	Jun 12, 1987	1

titleauthor Table

titleauthor is defined as follows:

```
create table titleauthor
(au_id id not null,
title_id tid not null,
au_ord tinyint null,
royaltyper int null)
```

Its primary keys are *au_id* and *title_id*:

```
sp_primarykey titleauthor, au_id, title_id
```

Its *title_id* and *au_id* columns are foreign keys to *titles* and *authors*:

```
sp_foreignkey titleauthor, titles, title_id
sp_foreignkey titleauthor, authors, au_id
```

Its nonclustered index for the *au_id* column is defined as:

```
create nonclustered index auidind
on titleauthor(au_id)
```

Its nonclustered index for the *title_id* column is defined as:

```
create nonclustered index titleidind
on titleauthor(title_id)
```

The following view uses *titleauthor*:

```
create view titleview
as
select title, au_ord, au_lname,
price, total_sales, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

The following procedure uses *titleauthor:*

```
create procedure byroyalty @percentage int
as
select au_id from titleauthor
where titleauthor.royaltyper = @percentage
```

Table A-4 lists the contents of *titleauthor*:

Table A-4: titleauthor table

au_id	title_id	au_ord	royaltyper
172-32-1176	PS3333	1	100
213-46-8915	BU1032	2	40
213-46-8915	BU2075	1	100
238-95-7766	PC1035	1	100
267-41-2394	BU1111	2	40
267-41-2394	TC7777	2	30
274-80-9391	BU7832	1	100
409-56-7008	BU1032	1	60
427-17-2319	PC8888	1	50

Table A-4: titleauthor table (continued)

au_id	title_id	au_ord	royaltyper
472-27-2349	TC7777	3	30
486-29-1786	PC9999	1	100
486-29-1786	PS7777	1	100
648-92-1872	TC4203	1	100
672-71-3249	TC7777	1	40
712-45-1867	MC2222	1	100
722-51-5454	MC3021	1	75
724-80-9391	BU1111	1	60
724-80-9391	PS1372	2	25
756-30-7391	PS1372	1	75
807-91-6654	TC3218	1	100
846-92-7186	PC8888	2	50
899-46-2035	MC3021	2	25
899-46-2035	PS2091	2	50
998-72-3567	PS2091	1	50
998-72-3567	PS2106	1	100

salesdetail Table

salesdetail is defined as follows:

```
create table salesdetail
(stor_id char(4) not null,
ord_num numeric(6,0),
title_id tid not null,
qty smallint not null,
discount float not null)
```

Its primary keys are *stor_id* and *ord_num*:

```
sp_primarykey salesdetail, stor_id, ord_num
```

Its *title_id*, *stor_id*, and *ord_num* columns are foreign keys to *titles* and *sales*:

```
sp_foreignkey salesdetail, titles, title_id
sp_foreignkey salesdetail, sales, stor_id, ord_num
```

Its nonclustered index for the *title_id* column is defined as:

```
create nonclustered index titleidind
on salesdetail (title_id)
```

Its nonclustered index for the *stor_id* column is defined as:

```
create nonclustered index salesdetailind
on salesdetail (stor_id)
```

Its *title_idrule* rule is defined as:

```
create rule title_idrule
as
@title_id like "BU[0-9][0-9][0-9][0-9]" or
@title_id like "[MT]C[0-9][0-9][0-9][0-9]" or
@title_id like "P[SC][0-9][0-9][0-9][0-9]" or
@title_id like "[A-Z][A-Z]xxxx" or
@title_id like "[A-Z][A-Z]yyyy"
```

salesdetail uses the following trigger:

```
create trigger totalsales_trig on salesdetail
    for insert, update, delete
as
/* Save processing:  return if there are no rows affected */
if @@rowcount = 0
    begin
       return
end
/* add all the new values */
/* use isnull:  a null value in the titles table means
**              "no sales yet" not "sales unknown"
*/
update titles
    set total_sales = isnull(total_sales, 0) + (select sum(qty)
    from inserted
    where titles.title_id = inserted.title_id)
    where title_id in (select title_id from inserted)
/* remove all values being deleted or updated */
update titles
    set total_sales = isnull(total_sales, 0) - (select sum(qty)
    from deleted
    where titles.title_id = deleted.title_id)
    where title_id in (select title_id from deleted)
```

Table A-5 lists the contents of *salesdetail*:

Table A-5: salesdetail table

stor_id	ord_num	title_id	qty	discount
7896	234518	TC3218	75	40.000000

Table A-5: salesdetail table (continued)

stor_id	ord_num	title_id	qty	discount
7896	234518	TC7777	75	40.000000
7131	Asoap432	TC3218	50	40.000000
7131	Asoap432	TC7777	80	40.000000
5023	XS-135-DER-432-8J2	TC3218	85	40.000000
8042	91-A-7	PS3333	90	45.000000
8042	91-A-7	TC3218	40	45.000000
8042	91-A-7	PS2106	30	45.000000
8042	91-V-7	PS2106	50	45.000000
8042	55-V-7	PS2106	31	45.000000
8042	91-A-7	MC3021	69	45.000000
5023	BS-345-DSE-860-1F2	PC1035	1000	46.700000
5023	AX-532-FED-452-2Z7	BU2075	500	46.700000
5023	AX-532-FED-452-2Z7	BU1032	200	46.700000
5023	AX-532-FED-452-2Z7	BU7832	150	46.700000
5023	AX-532-FED-452-2Z7	PS7777	125	46.700000
5023	NF-123-ADS-642-9G3	TC7777	1000	46.700000
5023	NF-123-ADS-642-9G3	BU1032	1000	46.700000
5023	NF-123-ADS-642-9G3	PC1035	750	46.700000
7131	Fsoap867	BU1032	200	46.700000
7066	BA52498	BU7832	100	46.700000
7066	BA71224	PS7777	200	46.700000
7066	BA71224	PC1035	300	46.700000
7066	BA71224	TC7777	350	46.700000
5023	ZD-123-DFG-752-9G8	PS2091	1000	46.700000
7067	NB-3.142	PS2091	200	46.700000
7067	NB-3.142	PS7777	250	46.700000
7067	NB-3.142	PS3333	345	46.700000
7067	NB-3.142	BU7832	360	46.700000
5023	XS-135-DER-432-8J2	PS2091	845	46.700000
5023	XS-135-DER-432-8J2	PS7777	581	46.700000
5023	ZZ-999-ZZZ-999-0A0	PS1372	375	46.700000
7067	NB-3.142	BU1111	175	46.700000
5023	XS-135-DER-432-8J2	BU7832	885	46.700000
5023	ZD-123-DFG-752-9G8	BU7832	900	46.700000
5023	AX-532-FED-452-2Z7	TC4203	550	46.700000
7131	Fsoap867	TC4203	350	46.700000
7896	234518	TC4203	275	46.700000
7066	BA71224	TC4203	500	46.700000
7067	NB-3.142	TC4203	512	46.700000
7131	Fsoap867	MC3021	400	46.700000
5023	AX-532-FED-452-2Z7	PC8888	105	46.700000
5023	NF-123-ADS-642-9G3	PC8888	300	46.700000

Table A-5: salesdetail table (continued)

stor_id	ord_num	title_id	qty	discount
7066	BA71224	PC8888	350	46.700000
7067	NB-3.142	PC8888	335	46.700000
7131	Asoap432	BU1111	500	46.700000
7896	234518	BU1111	340	46.700000
5023	AX-532-FED-452-2Z7	BU1111	370	46.700000
5023	ZD-123-DFG-752-9G8	PS3333	750	46.700000
8042	13-J-9	BU7832	300	51.700000
8042	13-E-7	BU2075	150	51.700000
8042	13-E-7	BU1032	300	51.700000
8042	13-E-7	PC1035	400	51.700000
8042	91-A-7	PS7777	180	51.700000
8042	13-J-9	TC4203	250	51.700000
8042	13-E-7	TC4203	226	51.700000
8042	13-E-7	MC3021	400	51.700000
8042	91-V-7	BU1111	390	51.700000
5023	AB-872-DEF-732-2Z1	MC3021	5000	50.000000
5023	NF-123-ADS-642-9G3	PC8888	2000	50.000000
5023	NF-123-ADS-642-9G3	BU2075	2000	50.000000
5023	GH-542-NAD-713-9F9	PC1035	2000	50.000000
5023	ZA-000-ASD-324-4D1	PC1035	2000	50.000000
5023	ZA-000-ASD-324-4D1	PS7777	1500	50.000000
5023	ZD-123-DFG-752-9G8	BU2075	3000	50.000000
5023	ZD-123-DFG-752-9G8	TC7777	1500	50.000000
5023	ZS-645-CAT-415-1B2	BU2075	3000	50.000000
5023	ZS-645-CAT-415-1B2	BU2075	3000	50.000000
5023	XS-135-DER-432-8J2	PS3333	2687	50.000000
5023	XS-135-DER-432-8J2	TC7777	1090	50.000000
5023	XS-135-DER-432-8J2	PC1035	2138	50.000000
5023	ZZ-999-ZZZ-999-0A0	MC2222	2032	50.000000
5023	ZZ-999-ZZZ-999-0A0	BU1111	1001	50.000000
5023	ZA-000-ASD-324-4D1	BU1111	1100	50.000000
5023	NF-123-ADS-642-9G3	BU7832	1400	50.000000
5023	BS-345-DSE-860-1F2	TC4203	2700	50.000000
5023	GH-542-NAD-713-9F9	TC4203	2500	50.000000
5023	NF-123-ADS-642-9G3	TC4203	3500	50.000000
5023	BS-345-DSE-860-1F2	MC3021	4500	50.000000
5023	AX-532-FED-452-2Z7	MC3021	1600	50.000000
5023	NF-123-ADS-642-9G3	MC3021	2550	50.000000
5023	ZA-000-ASD-324-4D1	MC3021	3000	50.000000
5023	ZS-645-CAT-415-1B2	MC3021	3200	50.000000
5023	BS-345-DSE-860-1F2	BU2075	2200	50.000000
5023	GH-542-NAD-713-9F9	BU1032	1500	50.000000

Table A-5: salesdetail table (continued)

stor_id	ord_num	title_id	qty	discount
5023	ZZ-999-ZZZ-999-0A0	PC8888	1005	50.000000
7896	124152	BU2075	42	50.500000
7896	124152	PC1035	25	50.500000
7131	Asoap132	BU2075	35	50.500000
7067	NB-1.142	PC1035	34	50.500000
7067	NB-1.142	TC4203	53	50.500000
8042	12-F-9	BU2075	30	55.500000
8042	12-F-9	BU1032	94	55.500000
7066	BA27618	BU2075	200	57.200000
7896	124152	TC4203	350	57.200000
7066	BA27618	TC4203	230	57.200000
7066	BA27618	MC3021	200	57.200000
7131	Asoap132	MC3021	137	57.200000
7067	NB-1.142	MC3021	270	57.200000
7067	NB-1.142	BU2075	230	57.200000
7131	Asoap132	BU1032	345	57.200000
7067	NB-1.142	BU1032	136	57.200000
8042	12-F-9	TC4203	300	62.200000
8042	12-F-9	MC3021	270	62.200000
8042	12-F-9	PC1035	133	62.200000
5023	AB-123-DEF-425-1Z3	TC4203	2500	60.500000
5023	AB-123-DEF-425-1Z3	BU2075	4000	60.500000
6380	342157	BU2075	200	57.200000
6380	342157	MC3021	250	57.200000
6380	356921	PS3333	200	46.700000
6380	356921	PS7777	500	46.700000
6380	356921	TC3218	125	46.700000
6380	234518	BU2075	135	46.700000
6380	234518	BU1032	320	46.700000
6380	234518	TC4203	300	46.700000
6380	234518	MC3021	400	46.700000

sales Table

sales is defined as follows:

```
create table sales
(stor_id char(4) not null,
ord_num varchar(20) not null,
date datetime not null)
```

Its primary keys are *stor_id* and *ord_num*:

```
sp_primarykey sales, stor_id, ord_num
```

Its *stor_id* column is a foreign key to *stores*:

```
sp_foreignkey sales, stores, stor_id
```

Table A-6 lists the contents of *sales*:

Table A-6: sales table

stor_id	ord_num	date
5023	AB-123-DEF-425-1Z3	Oct 31 1995
5023	AB-872-DEF-732-2Z1	Nov 6 1995
5023	AX-532-FED-452-2Z7	Dec 1 1996
5023	BS-345-DSE-860-1F2	Dec 12 1996
5023	GH-542-NAD-713-9F9	Feb 15 1997
5023	NF-123-ADS-642-9G3	Mar 18 1997
5023	XS-135-DER-432-8J2	Mar 21 1997
5023	ZA-000-ASD-324-4D1	Jul 27 1994
5023	ZD-123-DFG-752-9G8	Mar 21 1997
5023	ZS-645-CAT-415-1B2	Mar 21 1997
5023	ZZ-999-ZZZ-999-0A0	Mar 21 1997
6380	342157	Dec 13 1994
6380	356921	Feb 17 1995
7066	BA27618	Oct 12 1996
7066	BA52498	Oct 27 1995
7066	BA71224	Aug 5 1996
7067	NB-1.142	Jan 2 1997
7067	NB-3.142	Jun 13 1995
7131	Asoap132	Nov 16 1996
7131	Asoap432	Dec 20 1995
7131	Fsoap867	Sep 8 1996
7896	124152	Aug 14 1996
7896	234518	Feb 14 1997
8042	12-F-9	Jul 13 1996
8042	13-E-7	May 23 1995
8042	13-J-9	Jan 13 1997
8042	55-V-7	Feb 20 1997

Table A-6: sales table (continued)

stor_id	ord_num	date
8042	91-A-7	Mar 20 1997
8042	91-V-7	Mar 20 1997

stores Table

stores is defined as follows:

```
create table stores
(stor_id char(4) not null,
stor_name varchar(40) not null,
stor_address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode char(10) null,
payterms varchar(12) null)
```

Its primary key is *stor_id*:

```
sp_primarykey stores, stor_id
```

Table A-7 lists the contents of *stores*:

Table A-7: stores table

stor-_id	stor_-name	stor_-address	city	state	country	postal-code	pay-terms
7066	Barnum's	567 Pasadena Ave.	Tustin	CA	USA	92789	Net 30
7067	News & Brews	577 First St.	Los Gatos	CA	USA	96745	Net 30
7131	Doc-U-Mat: Quality Laundry and Books	24-A Avrogado Way	Remulade	WA	USA	98014	Net 60
8042	Bookbeat	679 Carson St.	Portland	OR	USA	89076	Net 30
6380	Eric the Read Books	788 Catamaugus Ave.	Seattle	WA	USA	98056	Net 60
7896	Fricative Bookshop	89 Madison St.	Fremont	CA	USA	90019	Net 60
5023	Thoreau Reading Discount Chain	20435 Walden Expressway	Concord	MA	USA	01776	Net 60

roysched Table

roysched is defined as follows:

```
create table roysched
title_id tid not null,
lorange int null,
hirange int null,
royalty int null)
```

Its primary key is *title_id*:

```
sp_primarykey roysched, title_id
```

Its *title_id* column is a foreign key to *titles*:

```
sp_foreignkey roysched, titles, title_id
```

Its nonclustered index for the *title_id* column is defined as:

```
create nonclustered index titleidind
on roysched (title_id)
```

Table A-8 lists the contents of *roysched*:

Table A-8: roysched table

title_id	lorange	hirange	royalty
BU1032	0	5000	10
BU1032	5001	50000	12
PC1035	0	2000	10
PC1035	2001	3000	12
PC1035	3001	4000	14
PC1035	4001	10000	16
PC1035	10001	50000	18
BU2075	0	1000	10
BU2075	1001	3000	12
BU2075	3001	5000	14
BU2075	5001	7000	16
BU2075	7001	10000	18
BU2075	10001	12000	20
BU2075	12001	14000	22
BU2075	14001	50000	24
PS2091	0	1000	10
PS2091	1001	5000	12
PS2091	5001	10000	14

Table A-8: roysched table (continued)

title_id	lorange	hirange	royalty
PS2091	10001	50000	16
PS2106	0	2000	10
PS2106	2001	5000	12
PS2106	5001	10000	14
PS2106	10001	50000	16
MC3021	0	1000	10
MC3021	1001	2000	12
MC3021	2001	4000	14
MC3021	4001	6000	16
MC3021	6001	8000	18
MC3021	8001	10000	20
MC3021	10001	12000	22
MC3021	12001	50000	24
TC3218	0	2000	10
TC3218	2001	4000	12
TC3218	4001	6000	14
TC3218	6001	8000	16
TC3218	8001	10000	18
TC3218	10001	12000	20
TC3218	12001	14000	22
TC3218	14001	50000	24
PC8888	0	5000	10
PC8888	5001	10000	12
PC8888	10001	15000	14
PC8888	15001	50000	16
PS7777	0	5000	10
PS7777	5001	50000	12
PS3333	0	5000	10
PS3333	5001	10000	12
PS3333	10001	15000	14
PS3333	15001	50000	16
BU1111	0	4000	10
BU1111	4001	8000	12
BU1111	8001	10000	14

Table A-8: roysched table (continued)

title_id	lorange	hirange	royalty
BU1111	12001	16000	16
BU1111	16001	20000	18
BU1111	20001	24000	20
BU1111	24001	28000	22
BU1111	28001	50000	24
MC2222	0	2000	10
MC2222	2001	4000	12
MC2222	4001	8000	14
MC2222	8001	12000	16
MC2222	8001	12000	16
MC2222	12001	20000	18
MC2222	20001	50000	20
TC7777	0	5000	10
TC7777	5001	15000	12
TC7777	15001	50000	14
TC4203	0	2000	10
TC4203	2001	8000	12
TC4203	8001	16000	14
TC4203	16001	24000	16
TC4203	24001	32000	18
TC4203	32001	40000	20
TC4203	40001	50000	22
BU7832	0	5000	10
BU7832	5001	10000	12
BU7832	10001	15000	14
BU7832	15001	20000	16
BU7832	20001	25000	18
BU7832	25001	30000	20
BU7832	30001	35000	22
BU7832	35001	50000	24
PS1372	0	10000	10
PS1372	10001	20000	12
PS1372	20001	30000	14
PS1372	30001	40000	16

Table A-8: roysched table (continued)

title_id	lorange	hirange	royalty
PS1372	40001	50000	18

discounts Table

discounts is defined as follows:

```
create table discounts
(discounttype varchar(40) not null,
stor_id char(4) null,
lowqty smallint null,
highqty smallint null,
discount float not null)
```

Its primary keys are *discounttype* and *stor_id*:

```
sp_primarykey discounts, discounttype, stor_id
```

Its *stor_id* is a foreign key to *stores*:

```
sp_foreignkey discounts, stores, stor_id
```

Table A-9 lists the contents of *discounts*:

Table A-9: discounts table

discounttype	stor_id	lowqty	highqty	discount
Initial Customer	8042	NULL	NULL	10.5
Volume Discount	NULL	100	1001	6.7
Huge Volume Discount	NULL	1001	NULL	10
Customer Discount	8042	NULL	NULL	5

blurbs Table

blurbs is defined as follows:

```
create table blurbs
(au_id  id not null,
copy text null)
```

Its primary key is *au_id*:

```
sp_primarykey blurbs, au_id
```

Its *au_id* column is a foreign key to *authors*:

```
sp_foreignkey blurbs, authors, au_id
```

Table A-10 lists the contents of *blurbs*:

Table A-10: blurbs table

au_id	copy
486-29-1786	If Chastity Locksley didn't exist, this troubled world would have created her! Not only did she master the mystic secrets of inner strength to conquer adversity when she encountered it in life, but also, after "reinventing herself," as she says, by writing "Emotional Security: A New Algorithm" following the devastating loss of her cat, Old Algorithm, she founded Publish or Perish, the page-by-page, day-by-day, write-yourself-to-wellness encounter workshops franchise empire, the better to share her inspiring discoveries with us all. Her "Net Etiquette," a brilliant social treatise in its own right and a fabulous pun, is the only civilized alternative to the gross etiquette often practiced on the public networks.
648-92-1872	A chef's chef and a raconteur's raconteur, Reginald Blotchet-Halls calls London his second home. "Th' palace kitchen's me first 'ome, act'lly!" Blotchet-Halls' astounding ability to delight our palates with palace delights is matched only by his equal skill in satisfying our perpetual hunger for delicious back-stairs gossip by serving up tidbits and entrees literally fit for a king!
998-72-3567	Albert Ringer was born in a trunk to circus parents, but another kind of circus trunk played a more important role in his life years later. He grew up as an itinerant wrestler and roustabout in the reknowned Ringer Brothers and Betty and Bernie's Circus. Once known in the literary world only as Anne Ringer's wrestling brother, he became a writer while recuperating from a near-fatal injury received during a charity benefit bout with a gorilla. "Slingshotting" himself from the ring ropes, Albert flew over the gorilla's head and would have landed head first on the concrete. He was saved from certain death by Nana, an elephant he had befriended as a child, who caught him in her trunk. Nana held him so tightly that three ribs cracked and he turned blue from lack of oxygen. "I was delirious. I had an out-of-body experience! My whole life passed before me eyes. I promised myself 'If I get through this, I'll use my remaining time to share what I learned out there.' I owe it all to Nana!"
899-46-2035	Anne Ringer ran away from the circus as a child. A university creative writing professor and her family took Anne in and raised her as one of their own. In this warm and television-less setting she learned to appreciate the great classics of literature. The stream of aspiring and accomplished writers that flowed constantly through the house confirmed her repudiation of the circus family she'd been born into: "Barbarians!" The steadily growing recognition of her literary work was, to her, vindication. When her brother's brush with death brought them together after many years, she took advantage of life's crazy chance thing and broke the wall of anger that she had constructed to separate them. Together they wrote, "Is Anger the Enemy?" an even greater blockbuster than her other collaborative work, with Michel DeFrance, "The Gourmet Microwave."

Table A-10: blurbs table (continued)

au_id	copy
672-71-3249	They asked me to write about myself and my book, so here goes: I started a restaurant called "de Gustibus" with two of my friends. We named it that because you really can't discuss taste. We're very popular with young business types because we're young business types ourselves. Whenever we tried to go out to eat in a group we always got into these long tiresome negotiations: "I just ate Italian," or "I ate Greek yesterday," or "I NEVER eat anything that's not organic!" Inefficient. Not what business needs today. So, it came to us that we needed a restaurant we could all go to every day and not eat the same thing twice in a row maybe for a year! We thought, "Hey, why make people choose one kind of restaurant over another, when what they really want is a different kind of food?" At de Gustibus you can eat Italian, Chinese, Japanese, Greek, Russian, Tasmanian, Iranian, and on and on all at the same time. You never have to choose. You can even mix and match! We just pooled our recipes, opened the doors, and never looked back. We're a big hit, what can I say? My recipes in "Sushi, Anyone?" are used at de Gustibus. They satisfy crowds for us every day. They will work for you, too. Period!
409-56-7008	Bennet was the classic too-busy executive. After discovering computer databases, he now has the time to run several successful businesses and sit on three major corporate boards. Bennet also donates time to community service organizations. Miraculously, he also finds time to write and market executive-oriented, in-depth computer hardware and software reviews. "I'm hyperkinetic, so being dynamic and fast-moving is a piece of cake. But being organized isn't easy for me or for anyone I know. There's just one word for that: 'databases!' Databases can cure you or kill you. If you get the right one, you can be like me. If you get the wrong one, watch out. Read my book!"

au_pix Table

au_pix is defined as follows:

```
create table au_pix
(au_id char(11) not null,
pic image null,
format_type char(11) null,
bytesize int null,
pixwidth_hor char(14) null,
pixwidth_vert char(14) null)
```

Its primary key is *au_id*:

```
sp_primarykey au_pix, au_id
```

Its *au_id* column is a foreign key to authors:

```
sp_foreignkey au_pix, authors, au_id
```

Table A-11 lists the contents of *au_pix*:

Table A-11: au_pix table

au_id	pic	format_- type	bytesize	pixwidth_- hor	pixwidth_- vert
409-56-7008	0x0000...	PICT	30220	626	635
486-29-1786	0x59a6...	Sunraster	27931	647	640
648-92-1872	0x59a6...	Sunraster	36974	647	640
672-71-3249	0x000a...	PICT	13487	654	639
899-46-2035	0x4949...	TIF	52023	648	641
998-72-3567	0x4949...	TIF	52336	653	637

The *pic* column contains binary data, which is not reproduced in its entirety in this table. The pictures represented by this data are shown on the next page. Since the *image* data (six pictures, two each in PICT, TIF, and Sunraster file formats) is quite large, you should run the **installpix2** script **only** if you want to use or test the *image* datatype. The *image* data is supplied to show how Sybase stores *image* data. Sybase does not supply any tools for displaying *image* data: you must use the appropriate screen graphics tools in order to display the images once you have extracted them from the database.

Author Portraits from the *au_pix* Table

Akiko Yokomoto 672-71-3249

Chastity Locksley 486-29-1786

Anne Ringer 899-46-2035

Albert Ringer 998-72-3567

Bennet Abraham 409-56-7008

Reginald Blotchet-Halls 648-92-1872

Diagram of the *pubs2* Database

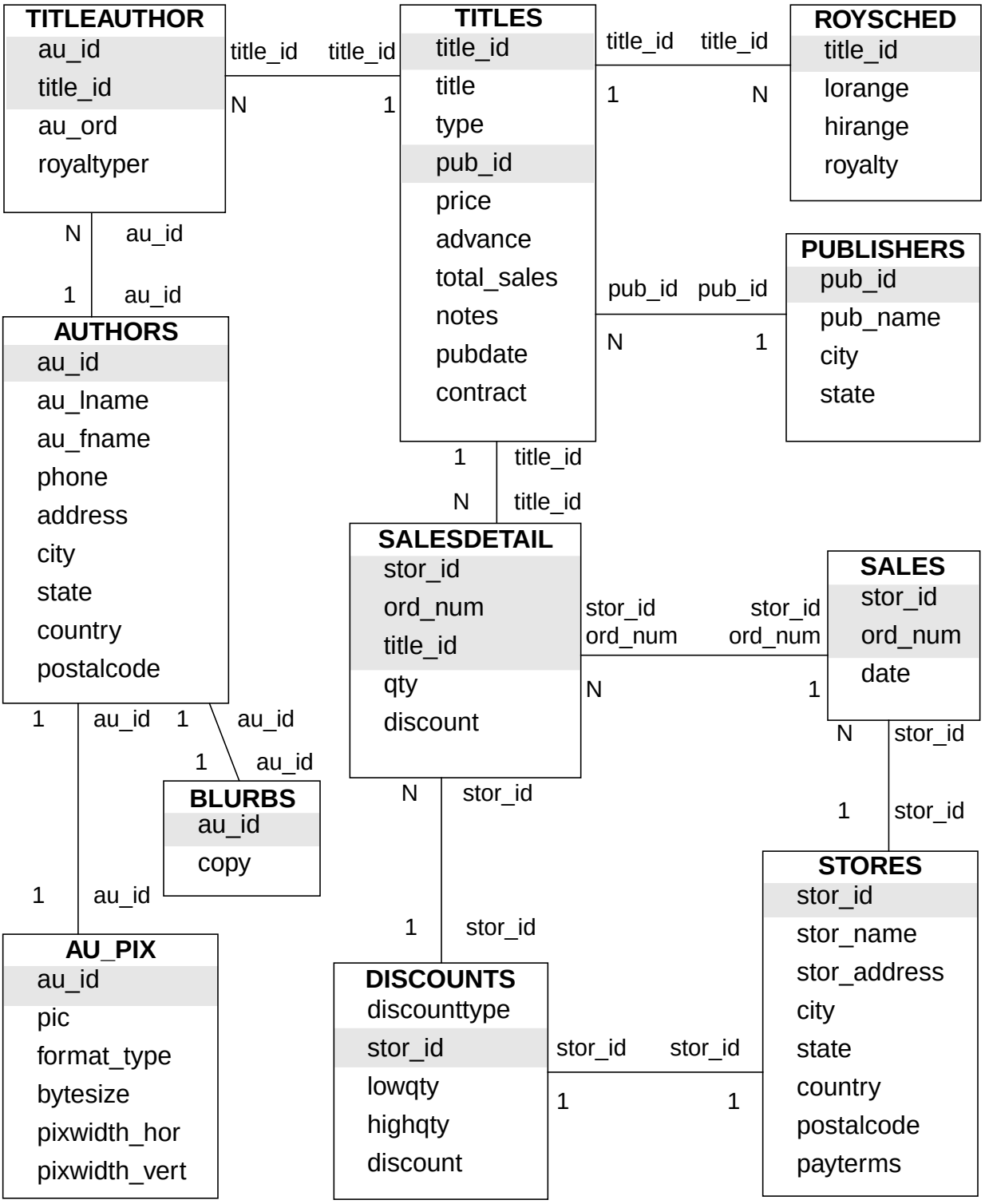


Figure A-1: Diagram of the pubs2 database

B The *pubs3* Database

This chapter describes the sample database *pubs3*. This database contains the tables *publishers, authors, titles, titleauthor, salesdetail, sales, stores, store_employees, discounts, roysched,* and *blurbs.*

The *pubs3* database also contains primary and foreign keys, rules, defaults, views, triggers and stored procedures.

A diagram of the *pubs3* database appears in Figure B-1 on page B-29.

For information about installing *pubs3,* see the configuration guide for your platform.

Tables in the *pubs3* Database

The following tables describe each *pubs3* table. Each column header specifies the column name, its datatype (including any user-defined datatypes), its null or not null status, and how it uses referential integrity. The column header also specifies any defaults, rules, triggers, and indexes that affect the column are also specified in the column header.

publishers Table

publishers is defined as follows:

```
create table publishers
(pub_id char(4) not null,
pub_name varchar(40) not null,
city varchar(20) null,
state char(2) null,
unique nonclustered (pub_id))
```

Its *pub_idrule* rule is defined as:

```
create rule pub_idrule
as @pub_id in
("1389", "0736", "0877", "1622", "1756")
or @pub_id like "99[0-9][0-9]"
```

Table B-1 lists the contents of *publishers*:

Table B-1: publishers table

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

authors Table

authors is defined as follows:

```
create table authors
(au_id id not null,
au_lname varchar(40) not null,
au_fname varchar(20) not null,
phone char(12) not null,
address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode char(10) null,
unique nonclustered (au_id))
```

Its nonclustered index for the *au_lname* and *au_fname* columns is defined as:

```
create nonclustered index aunmind
on authors (au_lname, au_fname)
```

The *phone* column has the following default:

```
create default phonedflt as "UNKNOWN"

sp_bindefault phonedft, "authors.phone"
```

The following view uses *authors*:

```
create view titleview
as
select title, au_ord, au_lname,
price, num_sold, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

Table B-2 lists the contents of *authors*:

Table B-2: authors table

au_id	au_-lname	au_ fname	phone	address	city	state	coun-try	post-alcode
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	USA	94025
213-46-8915	Green	Marjorie	510 986-7020	309 63rd St. #411	Oakland	CA	USA	94618
238-95-7766	Carson	Cheryl	510 548-7723	589 Darwin Ln.	Berkeley	CA	USA	94705
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14	San Jose	CA	USA	95128
274-80-9391	Straight	Dick	510 834-2919	5420 College Av.	Oakland	CA	USA	94609
341-22-1782	Smith	Meander	913 843-0462	10 Mississippi Dr.	Lawrence	KS	USA	66044
409-56-7008	Bennet	Abraham	510 658-9932	6223 Bateman St.	Berkeley	CA	USA	94705
427-17-2319	Dull	Ann	415 836-7128	3410 Blonde St.	Palo Alto	CA	USA	94301
472-27-2349	Gringlesby	Burt	707 938-6445	PO Box 792	Covelo	CA	USA	95428
486-29-1786	Locksley	Chastity	415 585-4620	18 Broadway Av.	San Francisco	CA	USA	94130
527-72-3246	Greene	Morningstar	615 297-2723	22 Graybar House Rd.	Nashville	TN	USA	37215
648-92-1872	Blotchet-Halls	Reginald	503 745-6402	55 Hillsdale Bl.	Corvallis	OR	USA	97330
672-71-3249	Yokomoto	Akiko	415 935-4228	3 Silver Ct.	Walnut Creek	CA	USA	94595
712-45-1867	del Castillo	Innes	615 996-8275	2286 Cram Pl. #86	Ann Arbor	MI	USA	48105
722-51-5454	DeFrance	Michel	219 547-9982	3 Balding Pl.	Gary	IN	USA	46403
724-08-9931	Stringer	Dirk	510 843-2991	5420 Telegraph Av.	Oakland	CA	USA	94609
724-80-9391	MacFeather	Stearns	510 354-7128	44 Upland Hts.	Oakland	CA	USA	94612
756-30-7391	Karsen	Livia	510 534-9219	5720 McAuley St.	Oakland	CA	USA	94609
807-91-6654	Panteley	Sylvia	301 946-8853	1956 Arlington Pl.	Rockville	MD	USA	20853

Table B-2: authors table (continued)

au_id	au_-lname	au_fname	phone	address	city	state	coun-try	post-alcode
846-92-7186	Hunter	Sheryl	415 836-7128	3410 Blonde St.	Palo Alto	CA	USA	94301
893-72-1158	McBadden	Heather	707 448-4982	301 Putnam	Vacaville	CA	USA	95688
899-46-2035	Ringer	Anne	801 826-0752	67 Seventh Av.	Salt Lake City	UT	USA	84152
998-72-3567	Ringer	Albert	801 826-0752	67 Seventh Av.	Salt Lake City	UT	USA	84152

titles Table

titles is defined as follows:

```
create table titles
(title_id tid not null,
title varchar(80) not null,
type char(12) not null,
pub_id char(4) null
    references publishers(pub_id),
price money null,
advance numeric(12,2) null,
num_sold int null,
notes varchar(200) null,
pubdate datetime not null,
contract bit not null,
unique nonclustered (title_id))
```

Its nonclustered index for the *title* column is defined as:

```
create nonclustered index titleind
on titles (title)
```

Its *title_idrule* is defined as:

```
create rule title_idrule
as
@title_id like "BU[0-9][0-9][0-9][0-9]" or
@title_id like "[MT]C[0-9][0-9][0-9][0-9]" or
@title_id like "P[SC][0-9][0-9][0-9][0-9]" or
@title_id like "[A-Z][A-Z]xxxx" or
@title_id like "[A-Z][A-Z]yyyy"
```

The *type* column has the following default:

```
create default typedflt as "UNDECIDED"

sp_bindefault typedflt, "titles.type"
```

The *pubdate* column has this default:

```
create default datedflt as getdate()

sp_bindefault datedflt, "titles.pubdate"
```

titles uses the following trigger:

```
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
```

The following view uses *titles*:

```
create view titleview
as
select title, au_ord, au_lname,
price, num_sold, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

Table B-3 lists the contents of *titles:*

Table B-3: titles table

title-_id	title	type	pub-_id	price	ad-vance	num_-sold	notes	pub-date	con-tract
BU1032	The Busy Executive's Database Guide	business	1389	19.99	5000.00	4095	An overview of available database systems with emphasis on common business applica-tions. Illustrated.	Jun 12, 1986	1

Table B-3: titles table (continued)

title-_id	title	type	pub-_id	price	ad-vance	num_-sold	notes	pub-date	con-tract
BU1111	Cooking with Computers: Surrepti-tious Balance Sheets	business	1389	11.95	5000.00	3876	Helpful hints on how to use your electronic resources to the best advantage.	Jun 9, 1988	1
BU2075	You Can Combat Computer Stress!	business	0736	2.99	10125.00	18722	The latest medical and psycho-logical techniques for living with the electronic office. Easy-to-under-stand explana-tions.	Jun 30, 1985	1
BU7832	Straight Talk About Computers	business	1389	19.99	5000.00	4095	Annotated analysis of what computers can do for you: a no-hype guide for the critical user.	Jun 22, 1987	1
MC2222	Silicon Valley Gastro-nomic Treats	mod_-cook	0877	19.99	0.00	2032	Favorite recipes for quick, easy, and elegant meals; tried and tested by people who never have time to eat, let alone cook.	Jun 9, 1989	1

Table B-3: titles table (continued)

title-_id	title	type	pub-_id	price	ad-vance	num_-sold	notes	pub-date	con-tract
MC3021	The Gourmet Microwave	mod_-cook	0877	2.99	15000.00	22246	Traditional French gourmet recipes adapted for modern micro-wave cooking.	Jun 18, 1985	1
PC1035	But Is It User Friendly?	popular_comp	1389	22.95	7000.00	8780	A survey of software for the naive user, focusing on the “friendli-ness” of each.	Jun 30, 1986	1
MC3026	The Psychology of Computer Cooking	UNDE-CIDED	0877	NULL	NULL	NULL	NULL	Jul 24, 1991	0
PC8888	Secrets of Silicon Valley	popular_comp	1389	20.00	8000.00	4095	Muck-raking reporting by two coura-geous women on the world’s largest computer hardware and software manufact-urers.	Jun 12, 1987	1
PC9999	Net Etiquette	popular_comp	1389	NULL	NULL	NULL	A must-read for computer conferenc-ing debu-tantes!	Jul 24, 1996	0

Table B-3: titles table (continued)

title-_id	title	type	pub-_id	price	ad-vance	num_-sold	notes	pub-date	con-tract
PS1372	Computer Phobic and Non-Phobic Indivi-duals: Behavior Variations	psycho-logy	0877	21.59	7000.00	375	A must for the specialist, this book examines the difference between those who hate and fear computers and those who think they are swell.	Oct 21,1990	1
PS2091	Is Anger the Enemy?	psycho-logy	0736	10.95	2275.00	2045	Carefully researched study of the effects of strong emotions on the body. Metabolic charts included.	Jun 15, 1989	1
PS2106	Life Without Fear	psycho-logy	0736	7.00	6000.00	111	New exercise, medita-tion, and nutritional techniques that can reduce the shock of daily inter-actions. Popular audience. Sample menus included. Exercise video available separately.	Oct 5, 1990	1

Table B-3: titles table (continued)

title-_id	title	type	pub-_id	price	ad-vance	num_-sold	notes	pub-date	con-tract
PS3333	Prolonged Data Deprivation: Four Case Studies	psycho-logy	0736	19.99	2000.00	4072	What happens when the data runs dry? Searching evaluations of information-shortage effects on heavy users.	Jun 12, 1988	1
PS7777	Emotional Security: A New Algorithm	psycho-logy	0736	7.99	4000.00	3336	Protecting yourself and your loved ones from undue emotional stress in the modern world. Use of computer and nutritional aids empha-sized.	Jun 12, 1988	1
TC3218	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterra-nean	trad_cook	0877	20.95	7000.00	375	Profusely illustrated in color, this makes a wonderful gift book for a cuisine-oriented friend.	Oct 21, 1990	1

Table B-3: titles table (continued)

title-_id	title	type	pub-_id	price	ad-vance	num_-sold	notes	pub-date	con-tract
TC4203	Fifty Years in Bucking-ham Palace Kitchens	trad_cook	0877	11.95	4000.00	15096	More anecdotes from the Queen's favorite cook describing life among English royalty. Recipes, techni-ques, tender vignettes.	Jun 12, 1985	1
TC7777	Sushi, Anyone?	trad_cook	0877	14.99	8000.00	4095	Detailed instruc-tions on improving your position in life by learning how to make authentic Japanese sushi in your spare time. 5-10% increase in number of friends per recipe reported from beta test.	Jun 12, 1987	1

titleauthor Table

titleauthor is defined as follows:

```
create table titleauthor
(au_id id not null
    references authors(au_id),
title_id tid not null
    references titles(title_id),
au_ord tinyint null,
royaltyper int null)
```

Its nonclustered index for the *au_id* column is defined as:

```
create nonclustered index auidind
on titleauthor(au_id)
```

Its nonclustered index for the *title_id* column is defined as:

```
create nonclustered index titleidind
on titleauthor(title_id)
```

The following view uses *titleauthor*:

```
create view titleview
as
select title, au_ord, au_lname,
price, num_sold, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

The following procedure uses *titleauthor:*

```
create procedure byroyalty @percentage int
as
select au_id from titleauthor
where titleauthor.royaltyper = @percentage
```

Table B-4 lists the contents of *titleauthor*:

Table B-4: titleauthor table

au_id	title_id	au_ord	royaltyper
172-32-1176	PS3333	1	100
213-46-8915	BU1032	2	40
213-46-8915	BU2075	1	100
238-95-7766	PC1035	1	100
267-41-2394	BU1111	2	40
267-41-2394	TC7777	2	30
274-80-9391	BU7832	1	100
409-56-7008	BU1032	1	60
427-17-2319	PC8888	1	50
472-27-2349	TC7777	3	30
486-29-1786	PC9999	1	100
486-29-1786	PS7777	1	100
648-92-1872	TC4203	1	100

Table B-4: titleauthor table (continued)

au_id	title_id	au_ord	royaltyper
672-71-3249	TC7777	1	40
712-45-1867	MC2222	1	100
722-51-5454	MC3021	1	75
724-80-9391	BU1111	1	60
724-80-9391	PS1372	2	25
756-30-7391	PS1372	1	75
807-91-6654	TC3218	1	100
846-92-7186	PC8888	2	50
899-46-2035	MC3021	2	25
899-46-2035	PS2091	2	50
998-72-3567	PS2091	1	50
998-72-3567	PS2106	1	100

salesdetail Table

salesdetail is defined as follows:

```
create table salesdetail
(stor_id char(4) not null
    references sales(stor_id),
ord_num numeric(6,0)
    references sales(ord_num),
title_id tid not null
    references titles(title_id),
qty smallint not null,
discount float not null)
```

Its nonclustered index for the *title_id* column is defined as:

```
create nonclustered index titleidind
on salesdetail (title_id)
```

Its nonclustered index for the *stor_id* column is defined as:

```
create nonclustered index salesdetailind
on salesdetail (stor_id)
```

Its *title_idrule* rule is defined as:

```
create rule title_idrule
as
@title_id like "BU[0-9][0-9][0-9][0-9]" or
@title_id like "[MT]C[0-9][0-9][0-9][0-9]" or
@title_id like "P[SC][0-9][0-9][0-9][0-9]" or
@title_id like "[A-Z][A-Z]xxxx" or
@title_id like "[A-Z][A-Z]yyyy"
```

salesdetail uses the following trigger:

```
create trigger totalsales_trig on salesdetail
    for insert, update, delete
as
/* Save processing:  return if there are no rows affected */
if @@rowcount = 0
    begin
       return
end
/* add all the new values */
/* use isnull:  a null value in the titles table means
**              "no sales yet" not "sales unknown"
*/
update titles
    set num_sold = isnull(num_sold, 0) + (select sum(qty)
    from inserted
    where titles.title_id = inserted.title_id)
    where title_id in (select title_id from inserted)
/* remove all values being deleted or updated */
update titles
    set num_sold = isnull(num_sold, 0) - (select sum(qty)
    from deleted
    where titles.title_id = deleted.title_id)
    where title_id in (select title_id from deleted)
```

Table B-5 lists the contents of *salesdetail*:

Table B-5: salesdetail table

stor_id	ord_num	title_id	qty	discount
7896	100014	TC3218	75	40.000000
7896	100014	TC7777	75	40.000000
7131	100017	TC3218	50	40.000000
7131	100017	TC7777	80	40.000000
5023	100020	TC3218	85	40.000000
8042	100016	PS3333	90	45.000000
8042	100016	TC3218	40	45.000000
8042	100016	PS2106	30	45.000000
8042	100023	PS2106	50	45.000000
8042	100015	PS2106	31	45.000000

Table B-5: salesdetail table (continued)

stor_id	ord_num	title_id	qty	discount
8042	100016	MC3021	69	45.000000
5023	100009	PC1035	1000	46.700000
5023	100007	BU2075	500	46.700000
5023	100007	BU1032	200	46.700000
5023	100007	BU7832	150	46.700000
5023	100007	PS7777	125	46.700000
5023	100018	TC7777	1000	46.700000
5023	100018	BU1032	1000	46.700000
5023	100018	PC1035	750	46.700000
7131	100004	BU1032	200	46.700000
7066	100012	BU7832	100	46.700000
7066	100021	PS7777	200	46.700000
7066	100021	PC1035	300	46.700000
7066	100021	TC7777	350	46.700000
5023	100025	PS2091	1000	46.700000
7067	100019	PS2091	200	46.700000
7067	100019	PS7777	250	46.700000
7067	100019	PS3333	345	46.700000
7067	100019	BU7832	360	46.700000
5023	100020	PS2091	845	46.700000
5023	100020	PS7777	581	46.700000
5023	100027	PS1372	375	46.700000
7067	100019	BU1111	175	46.700000
5023	100020	BU7832	885	46.700000
5023	100025	BU7832	900	46.700000
5023	100007	TC4203	550	46.700000
7131	100004	TC4203	350	46.700000
7896	100014	TC4203	275	46.700000
7066	100021	TC4203	500	46.700000
7067	100019	TC4203	512	46.700000
7131	100004	MC3021	400	46.700000
5023	100007	PC8888	105	46.700000
5023	100018	PC8888	300	46.700000
7066	100021	PC8888	350	46.700000
7067	100019	PC8888	335	46.700000
7131	100017	BU1111	500	46.700000
7896	100014	BU1111	340	46.700000
5023	100007	BU1111	370	46.700000
5023	100025	PS3333	750	46.700000
8042	100014	BU7832	300	51.700000
8042	100013	BU2075	150	51.700000
8042	100013	BU1032	300	51.700000

Table B-5: salesdetail table (continued)

stor_id	ord_num	title_id	qty	discount
8042	100013	PC1035	400	51.700000
8042	100016	PS7777	180	51.700000
8042	100014	TC4203	250	51.700000
8042	100013	TC4203	226	51.700000
8042	100013	MC3021	400	51.700000
8042	100023	BU1111	390	51.700000
5023	100003	MC3021	5000	50.000000
5023	100018	PC8888	2000	50.000000
5023	100018	BU2075	2000	50.000000
5023	100010	PC1035	2000	50.000000
5023	100022	PC1035	2000	50.000000
5023	100022	PS7777	1500	50.000000
5023	100025	BU2075	3000	50.000000
5023	100025	TC7777	1500	50.000000
5023	100026	BU2075	3000	50.000000
5023	100026	BU2075	3000	50.000000
5023	100020	PS3333	2687	50.000000
5023	100020	TC7777	1090	50.000000
5023	100020	PC1035	2138	50.000000
5023	100027	MC2222	2032	50.000000
5023	100027	BU1111	1001	50.000000
5023	100022	BU1111	1100	50.000000
5023	100018	BU7832	1400	50.000000
5023	100009	TC4203	2700	50.000000
5023	100010	TC4203	2500	50.000000
5023	100018	TC4203	3500	50.000000
5023	100009	MC3021	4500	50.000000
5023	100007	MC3021	1600	50.000000
5023	100018	MC3021	2550	50.000000
5023	100022	MC3021	3000	50.000000
5023	100026	MC3021	3200	50.000000
5023	100009	BU2075	2200	50.000000
5023	100010	BU1032	1500	50.000000
5023	100027	PC8888	1005	50.000000
7896	100013	BU2075	42	50.500000
7896	100013	PC1035	25	50.500000
7131	100005	BU2075	35	50.500000
7067	100011	PC1035	34	50.500000
7067	100011	TC4203	53	50.500000
8042	100015	BU2075	30	55.500000
8042	100015	BU1032	94	55.500000
7066	100001	BU2075	200	57.200000

Table B-5: salesdetail table (continued)

stor_id	ord_num	title_id	qty	discount
7896	100013	TC4203	350	57.200000
7066	100001	TC4203	230	57.200000
7066	100001	MC3021	200	57.200000
7131	100005	MC3021	137	57.200000
7067	100011	MC3021	270	57.200000
7067	100011	BU2075	230	57.200000
7131	100005	BU1032	345	57.200000
7067	100011	BU1032	136	57.200000
8042	100015	TC4203	300	62.200000
8042	100015	MC3021	270	62.200000
8042	100015	PC1035	133	62.200000
5023	100002	TC4203	2500	60.500000
5023	100002	BU2075	4000	60.500000
6380	100028	BU2075	200	57.200000
6380	100028	MC3021	250	57.200000
6380	100029	PS3333	200	46.700000
6380	100029	PS7777	500	46.700000
6380	100029	TC3218	125	46.700000
6380	100014	BU2075	135	46.700000
6380	100014	BU1032	320	46.700000
6380	100014	TC4203	300	46.700000
6380	100014	MC3021	400	46.700000

sales Table

sales is defined as follows:

```
create table sales
(stor_id char(4) not null
    references stores(stor_id),
ord_num numeric(6,0) identity,
date datetime not null,
unique nonclustered (ord_num))
```

Table B-6 lists the contents of *sales*:

Table B-6: sales table

stor_id	ord_num	date
5023	100002	Oct 31 1995
5023	100003	Nov 6 1995
5023	100007	Dec 1 1996

Table B-6: sales table (continued)

stor_id	ord_num	date
5023	100009	Dec 12 1996
5023	100010	Feb 15 1997
5023	100018	Mar 18 1997
5023	100020	Mar 21 1997
5023	100022	Jul 27 1994
5023	100025	Mar 21 1997
5023	100026	Mar 21 1997
5023	100027	Mar 21 1997
6380	100028	Dec 13 1994
6380	100029	Feb 17 1995
7066	100001	Oct 12 1996
7066	100012	Oct 27 1995
7066	100021	Aug 5 1996
7067	100008	Jan 2 1997
7067	100024	Jun 13 1995
7131	100006	Nov 16 1996
7131	100011	Dec 20 1995
7131	100019	Sep 8 1996
7896	100005	Aug 14 1996
7896	100017	Feb 14 1997
8042	100004	Jul 13 1996
8042	100013	May 23 1995
8042	100014	Jan 13 1997
8042	100015	Feb 20 1997
8042	100016	Mar 20 1997
8042	100023	Mar 20 1997

stores Table

stores is defined as follows:

```
create table stores
(stor_id char(4) not null,
stor_name varchar(40) not null,
stor_address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode char(10) null,
payterms varchar(12) null,
unique nonclustered (stor_id))
```

Table B-7 lists the contents of *stores*:

Table B-7: stores table

stor-_id	stor_-name	stor_-address	city	state	country	postal-code	pay-terms
7066	Barnum's	567 Pasadena Ave.	Tustin	CA	USA	92789	Net 30
7067	News & Brews	577 First St.	Los Gatos	CA	USA	96745	Net 30
7131	Doc-U-Mat: Quality Laundry and Books	24-A Avrogado Way	Remulade	WA	USA	98014	Net 60
8042	Bookbeat	679 Carson St.	Portland	OR	USA	89076	Net 30
6380	Eric the Read Books	788 Catamaugus Ave.	Seattle	WA	USA	98056	Net 60
7896	Fricative Bookshop	89 Madison St.	Fremont	CA	USA	90019	Net 60
5023	Thoreau Reading Discount Chain	20435 Walden Expressway	Concord	MA	USA	01776	Net 60

store_employees Table

store_employees is defined as follows:

```
create table store_employees
(stor_id char(4) null
    references stores(stor_id),
emp_id id not null,
mgr_id id null
    references store_employees(emp_id),
emp_lname varchar(40) not null,
emp_fname varchar(20) not null,
phone char(12) null,
address varchar(40) null,
city varchar(20) null,
state char(2) null,
country varchar(12) null,
postalcode varchar(10) null,
unique nonclustered (emp_id))
```

Table B-8 lists the contents of *store_employees*:

Table B-8: store_employees table

stor-_id	emp-_id	mgr-_id	emp_-lname	emp_-fname	phone	address	city	state	coun-try	post-al-code
7066	415-92-7301	415-92-7301	Chichikov	P.	619 336-8978	20 Alfalfa Ave.	Twenty-nine Palms	CA	USA	92277
7066	202-93-0009	415-92-7301	Spindle-shanks	Daisy	619 336-8900	14 Wild Cat Way	Twenty-nine Palms	CA	USA	92277
7066	444-89-2342	415-92-7301	Ratsbane	Ian	619 344-92321	1827 Road-runner Dr.	Twenty-nine Palms	CA	USA	92278
7066	332-78-2382	415-92-7301	Bayless	Chuck	619 777-4356	7 Maynard Dr.	Twenty-nine Palms	CA	USA	92277
7066	898-00-2383	415-92-7301	Pumpion	Tammy	619 222-3434	10089 Lazy Joe Ave.	Twenty-nine Palms	CA	USA	92277
7066	222-90-3737	415-92-7301	Barnum	Arthur	619 333-3232	522 Lupine Ave.	Twenty-nine Palms	CA	USA	92277
7066	939-32-1212	415-92-7301	Ronald	Bill	619 333-4550	69 Coyote Ct.	Twenty-nine Palms	CA	USA	92277

Table B-8: store_employees table (continued)

stor-_id	emp-_id	mgr-_id	emp_-lname	emp_-fname	phone	address	city	state	coun-try	post-al-code
7067	343-38-9494	343-38-9494	St. Augustine	Fiona	408 222-8383	17 Call of the Wild Rd.	Los Gatos	CA	USA	95030
7067	222-90-3483	343-38-9494	Sebastian	Basil	408 334-8928	29 Dharma Rd.	Los Gatos	CA	USA	95030
7067	121-00-3823	343-38-9494	March-main	Albin	408 343-8934	1001 Top of the Hill Ct.	Los Gatos	CA	USA	95030
7067	232-29-8938	343-38-9494	Barlow	Hilary	408 232-3332	2 Topping Way	Los Gatos	CA	USA	95032
7067	232-89-9393	343-38-9494	Dieuberi	Gustav	408 675-8938	30 Mount Madonna Rd.	Los Gatos	CA	USA	95030
7131	565-67-3920	565-67-3920	Lavator	Marie	206 328-2838	14 Peter Grubb Rd. SE	Renton	WA	USA	98058
7131	848-34-4838	565-67-3920	Sope	Inez	206 438-3434	1238 Index Ave. SE	Renton	WA	USA	98058
7131	232-38-2232	565-67-3920	Tubb	Wally	206 323-2828	180 Moses Lane S	Renton	WA	USA	98058
7131	111-90-3283	565-67-3920	Basin	Chalker	206 323-7777	192 Naches Ave. N	Renton	WA	USA	98055
7131	323-32-4444	565-67-3920	Waters	Johnny	206 323-9020	1 Pelly Ave. N	Renton	WA	USA	98055
7131	238-32-0079	238-32-0079	Scrubb	Billy	206 444-3232	90 Vashion St. NE	Renton	WA	USA	98059
7131	349-00-9393	565-67-3920	Tuwel	Terry	206 323-3233	66 Sunset Blvd. SW	Renton	WA	USA	98055
8042	723-99-9329	723-99-9329	Crook-shank	Flannery	503 323-9090	SW 222 Laber Ct.	Port-land	OR	USA	97236
8042	343-32-0034	723-99-9329	Sitwell	Daven-port	503 333-9494	14 SE Krause Lane	Port-land	OR	USA	97236
8042	877-12-9393	723-99-9329	Bartholo-mew	Curan	503 333-9332	18 SE Knee St.	Port-land	OR	USA	97266
8042	323-99-0009	723-99-9329	Yaddo	Mary	503 333-4844	NE Klickitat St.	Port-land	OR	USA	97212

Table B-8: store_employees table (continued)

stor-_id	emp-_id	mgr-_id	emp_-lname	emp_-fname	phone	address	city	state	coun-try	post-al-code
8042	434-94-3203	723-99-9329	Peacock	Mavis	503 494-3413	4978 SW Huns Rd.	Port-land	OR	USA	97223
8042	232-94-5885	723-99-9329	Baldwin	Dorothy	503 333-9494	9 N. Image Canoe Ave.	Port-land	OR	USA	97217
8042	323-09-5872	723-99-9329	Corbit	P.	503 335-3491	89 NW Industry St.	Port-land	OR	USA	97210
8042	323-54-3434	723-99-9329	Pilkinton	Faith	503 333-4531	18 SW Inez St.	Port-land	OR	USA	97224
8042	331-90-8484	723-99-9329	Botteghe	Obscura	503 889-2832	555 SW Keerins Ct.	Port-land	OR	USA	97223
8042	884-32-8282	723-99-9329	Martineau	Tristan	503 323-5954	346 N. Leade Rd.	Port-land	OR	USA	97203
8042	112-90-3829	723-99-9329	Laughton	Karla	503 323-5929	900 N. Magpie St.	Port-land	OR	USA	97202
8042	434-23-9292	723-99-9329	MacLeish	Cochran	503 433-9493	1722 SE Ogden Ct.	Port-land	OR	USA	97202
6380	433-01-3922	433-01-3922	Horgran	Eric	206 323-4983	18 Thud Dr. S.	Seattle	WA	USA	98198
6380	232-30-9999	433-01-3922	Farnsdale	Finwick	206 434-4949	2 Bellevue Ave.	Seattle	WA	USA	98118
6380	333-90-3828	433-01-3922	Wolfe	Wanda	206 323-3333	2 S. Boze St. #15	Seattle	WA	USA	98108
6380	322-09-2122	433-01-3922	Begude	Nina	206 333-9023	1297 W. Cay Way	Seattle	WA	USA	98108
6380	342-78-2832	433-01-3922	Howe	Saphron	206 323-4933	9827 Des Moines Dr.	Seattle	WA	USA	98108
6380	882-00-3234	433-01-3922	Salidin	Uma	206 390-8328	13 Echo Lake Place	Seattle	WA	USA	98126
6380	321-89-8832	433-01-3922	Marsicano	Evan	206 322-9392	7 Dancer Ave. SW'	Seattle	WA	USA	98126
6380	762-32-5555	433-01-3922	Boulignini	Gus	206 323-9396	308 S. Fidalgo St.	Seattle	WA	USA	98108
7896	433-01-3921	433-01-3921	Malraison	Marcel	206 889-9983	2298 Berry Dr.	Fremont	CA	USA	94555
7896	322-32-8382	433-01-3921	West	Diana	206 898-3233	20-B Knute St.	Fremont	CA	USA	94536

Table B-8: store_employees table (continued)

stor-_id	emp-_id	mgr-_id	emp_-lname	emp_-fname	phone	address	city	state	coun-try	post-al-code
7896	328-74-8748	433-01-3921	Cazalis	Mari	206 326-6785	18 May Dr. #7	Fremont	CA	USA	94555
7896	325-87-6766	433-01-3921	Zorronius	Gaius	206 363-8988	1967 Claude Dr.	Fremont	CA	USA	94555
7896	323-67-5646	433-01-3921	Podding	Marvin	206 438-7858	242 Ram Ct. #123	Fremont	CA	USA	94539
7896	673-03-6728	433-01-3921	Phelan	Micky	206 323-7676	1212 Spinn Ct.	Fremont	CA	USA	94539
7896	326-77-6482	433-01-3921	Bokanow-ski	Peter	206 432-4832	13 Sparto Dr.	Fremont	CA	USA	94539
7896	653-48-2975	433-01-3921	Poodals	F.W.	206 387-6863	739 Sendak Dr.	Fremont	CA	USA	94538
7896	320-37-4682	433-01-3921	Scrubbs	Randy	206 323-3874	18 Washo Dr. #A	Fremont	CA	USA	94539
7896	323-67-4762	433-01-3921	Mullins	M.	206 382-7862	1717 Zircon Ter.	Fremont	CA	USA	94555
7896	326-76-3863	433-01-3921	Clarac	Sebastia n	206 387-6732	5622 Zin St. #78	Fremont	CA	USA	94539
5023	534-92-2910	534-92-2910	Armand	Mary	510 323-5478	1912 Badger Ln.	Con-cord	CA	USA	94521
5023	342-38-2782	534-92-2910	Blanzat	Gomez	510 382-3828	4229 Tiffany Pl.	Con-cord	CA	USA	94518
5023	322-39-3222	534-92-2910	Winate	Patrick	510 322-5873	16 Mahoo Ln.	Con-cord	CA	USA	94521
5023	932-20-4389	534-92-2910	Burke	Aldous	510 433-4829	12 Ponder Dr. #4	Con-cord	CA	USA	94521
5023	322-43-2828	534-92-2910	Rough-head	Jonathan	510 323-4782	2 Athos Ct.	Con-cord	CA	USA	94521
5023	832-32-3289	534-92-2910	Johnson	Alexis	510 323-3824	927 Aramis Ct.	Con-cord	CA	USA	94520
5023	432-32-4444	534-92-2910	Blitzstein	Jack	510 323-3874	2 Porthos Ct.	Con-cord	CA	USA	94519
5023	543-43-8438	534-92-2910	Darby	Andy	510 422-3232	222 Juniper Ct.	Con-cord	CA	USA	94519
5023	423-42-9933	534-92-2910	Askanzi	Robert	510 422-3232	222 Juniper Ct.	Con-cord	CA	USA	94519

Table B-8: store_employees table (continued)

stor-_id	emp-_id	mgr-_id	emp_-lname	emp_-fname	phone	address	city	state	coun-try	post-al-code
5023	838-28-1982	534-92-2910	Zavatta	Virginia	510 423-4932	629 Wind Way	Con-cord	CA	USA	94521
5023	433-43-3838	534-92-2910	Fife	Ben	510 322-8427	782 Sapling Ct.	Con-cord	CA	USA	94519

roysched Table

roysched is defined as follows:

```
create table roysched
title_id tid not null
    references titles(title_id),
lorange int null,
hirange int null,
royalty int null)
```

Its nonclustered index for the *title_id* column is defined as:

```
create nonclustered index titleidind
on roysched (title_id)
```

Table B-9 lists the contents of *roysched*:

Table B-9: roysched table

title_id	lorange	hirange	royalty
BU1032	0	5000	10
BU1032	5001	50000	12
PC1035	0	2000	10
PC1035	2001	3000	12
PC1035	3001	4000	14
PC1035	4001	10000	16
PC1035	10001	50000	18
BU2075	0	1000	10
BU2075	1001	3000	12
BU2075	3001	5000	14
BU2075	5001	7000	16
BU2075	7001	10000	18
BU2075	10001	12000	20

Table B-9: roysched table (continued)

title_id	lorange	hirange	royalty
BU2075	12001	14000	22
BU2075	14001	50000	24
PS2091	0	1000	10
PS2091	1001	5000	12
PS2091	5001	10000	14
PS2091	10001	50000	16
PS2106	0	2000	10
PS2106	2001	5000	12
PS2106	5001	10000	14
PS2106	10001	50000	16
MC3021	0	1000	10
MC3021	1001	2000	12
MC3021	2001	4000	14
MC3021	4001	6000	16
MC3021	6001	8000	18
MC3021	8001	10000	20
MC3021	10001	12000	22
MC3021	12001	50000	24
TC3218	0	2000	10
TC3218	2001	4000	12
TC3218	4001	6000	14
TC3218	6001	8000	16
TC3218	8001	10000	18
TC3218	10001	12000	20
TC3218	12001	14000	22
TC3218	14001	50000	24
PC8888	0	5000	10
PC8888	5001	10000	12
PC8888	10001	15000	14
PC8888	15001	50000	16
PS7777	0	5000	10
PS7777	5001	50000	12
PS3333	0	5000	10
PS3333	5001	10000	12

Table B-9: roysched table (continued)

title_id	lorange	hirange	royalty
PS3333	10001	15000	14
PS3333	15001	50000	16
BU1111	0	4000	10
BU1111	4001	8000	12
BU1111	8001	10000	14
BU1111	12001	16000	16
BU1111	16001	20000	18
BU1111	20001	24000	20
BU1111	24001	28000	22
BU1111	28001	50000	24
MC2222	0	2000	10
MC2222	2001	4000	12
MC2222	4001	8000	14
MC2222	8001	12000	16
MC2222	8001	12000	16
MC2222	12001	20000	18
MC2222	20001	50000	20
TC7777	0	5000	10
TC7777	5001	15000	12
TC7777	15001	50000	14
TC4203	0	2000	10
TC4203	2001	8000	12
TC4203	8001	16000	14
TC4203	16001	24000	16
TC4203	24001	32000	18
TC4203	32001	40000	20
TC4203	40001	50000	22
BU7832	0	5000	10
BU7832	5001	10000	12
BU7832	10001	15000	14
BU7832	15001	20000	16
BU7832	20001	25000	18
BU7832	25001	30000	20
BU7832	30001	35000	22

Table B-9: roysched table (continued)

title_id	lorange	hirange	royalty
BU7832	35001	50000	24
PS1372	0	10000	10
PS1372	10001	20000	12
PS1372	20001	30000	14
PS1372	30001	40000	16
PS1372	40001	50000	18

discounts Table

discounts is defined as follows:

```
create table discounts
(discounttype varchar(40) not null,
stor_id char(4) null
    references stores(stor_id),
lowqty smallint null,
highqty smallint null,
discount float not null)
```

Table B-10 lists the contents of *discounts*:

Table B-10: discounts table

discounttype	stor_id	lowqty	highqty	discount
Initial Customer	8042	NULL	NULL	10.5
Volume Discount	NULL	100	1001	6.7
Huge Volume Discount	NULL	1001	NULL	10
Customer Discount	8042	NULL	NULL	5

blurbs Table

blurbs is defined as follows:

```
create table blurbs
(au_id  id not null
    references authors(au_id),
copy text null)
```

Table B-11 lists the contents of *blurbs*:

Table B-11: blurbs table

au_id	copy
486-29-1786	If Chastity Locksley didn't exist, this troubled world would have created her! Not only did she master the mystic secrets of inner strength to conquer adversity when she encountered it in life, but also, after "reinventing herself," as she says, by writing "Emotional Security: A New Algorithm" following the devastating loss of her cat, Old Algorithm, she founded Publish or Perish, the page-by-page, day-by-day, write-yourself-to-wellness encounter workshops franchise empire, the better to share her inspiring discoveries with us all. Her "Net Etiquette," a brilliant social treatise in its own right and a fabulous pun, is the only civilized alternative to the gross etiquette often practiced on the public networks.
648-92-1872	A chef's chef and a raconteur's raconteur, Reginald Blotchet-Halls calls London his second home. "Th' palace kitchen's me first 'ome, act'lly!" Blotchet-Halls' astounding ability to delight our palates with palace delights is matched only by his equal skill in satisfying our perpetual hunger for delicious back-stairs gossip by serving up tidbits and entrees literally fit for a king!
998-72-3567	Albert Ringer was born in a trunk to circus parents, but another kind of circus trunk played a more important role in his life years later. He grew up as an itinerant wrestler and roustabout in the reknowned Ringer Brothers and Betty and Bernie's Circus. Once known in the literary world only as Anne Ringer's wrestling brother, he became a writer while recuperating from a near-fatal injury received during a charity benefit bout with a gorilla. "Slingshotting" himself from the ring ropes, Albert flew over the gorilla's head and would have landed head first on the concrete. He was saved from certain death by Nana, an elephant he had befriended as a child, who caught him in her trunk. Nana held him so tightly that three ribs cracked and he turned blue from lack of oxygen. "I was delirious. I had an out-of-body experience! My whole life passed before me eyes. I promised myself 'If I get through this, I'll use my remaining time to share what I learned out there.' I owe it all to Nana!"
899-46-2035	Anne Ringer ran away from the circus as a child. A university creative writing professor and her family took Anne in and raised her as one of their own. In this warm and television-less setting she learned to appreciate the great classics of literature. The stream of aspiring and accomplished writers that flowed constantly through the house confirmed her repudiation of the circus family she'd been born into: "Barbarians!" The steadily growing recognition of her literary work was, to her, vindication. When her brother's brush with death brought them together after many years, she took advantage of life's crazy chance thing and broke the wall of anger that she had constructed to separate them. Together they wrote, "Is Anger the Enemy?" an even greater blockbuster than her other collaborative work, with Michel DeFrance, "The Gourmet Microwave."

Table B-11: blurbs table (continued)

au_id	copy
672-71-3249	They asked me to write about myself and my book, so here goes: I started a restaurant called “de Gustibus” with two of my friends. We named it that because you really can’t discuss taste. We’re very popular with young business types because we’re young business types ourselves. Whenever we tried to go out to eat in a group we always got into these long tiresome negotiations: “I just ate Italian,” or “I ate Greek yesterday,” or “I NEVER eat anything that’s not organic!” Inefficient. Not what business needs today. So, it came to us that we needed a restaurant we could all go to every day and not eat the same thing twice in a row maybe for a year! We thought, “Hey, why make people choose one kind of restaurant over another, when what they really want is a different kind of food?” At de Gustibus you can eat Italian, Chinese, Japanese, Greek, Russian, Tasmanian, Iranian, and on and on all at the same time. You never have to choose. You can even mix and match! We just pooled our recipes, opened the doors, and never looked back. We’re a big hit, what can I say? My recipes in “Sushi, Anyone?” are used at de Gustibus. They satisfy crowds for us every day. They will work for you, too. Period!
409-56-7008	Bennet was the classic too-busy executive. After discovering computer databases, he now has the time to run several successful businesses and sit on three major corporate boards. Bennet also donates time to community service organizations. Miraculously, he also finds time to write and market executive-oriented, in-depth computer hardware and software reviews. “I’m hyperkinetic, so being dynamic and fast-moving is a piece of cake. But being organized isn’t easy for me or for anyone I know. There’s just one word for that: ‘databases!’ Databases can cure you or kill you. If you get the right one, you can be like me. If you get the wrong one, watch out. Read my book!”

Diagram of the *pubs3* Database

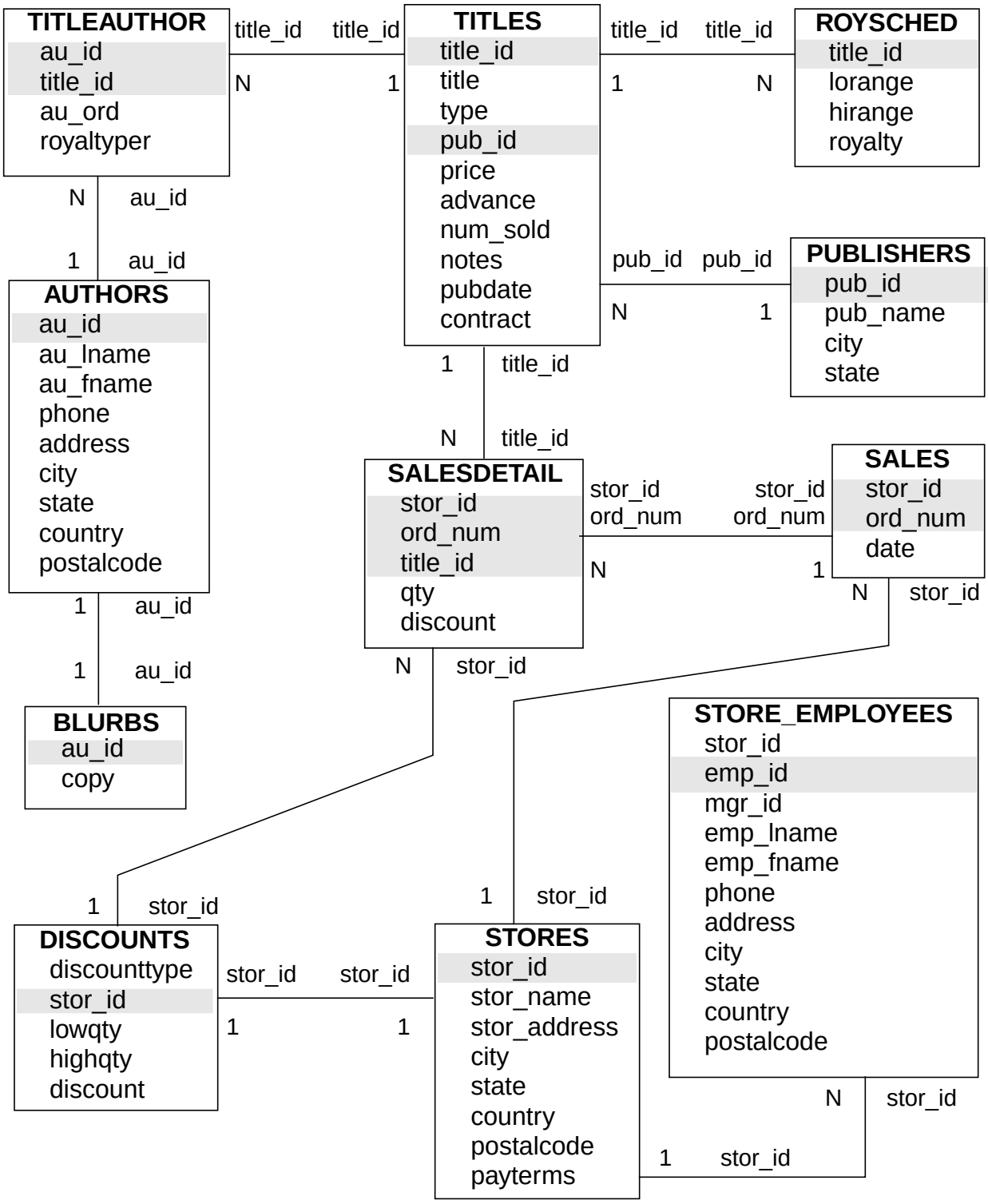


Figure B-1: Diagram of the pubs3 database

Index

Note: Page numbers in **bold** are primary references.

Symbols

Numerics

A

B

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

V

W

X

Y

Z