SYBASE®

Sybase® Adaptive Server™ Enterprise Reference Manual
Volume 1: Commands and Functions

Adaptive Server™

Document ID: 32401-01-1150

September 1997

Copyright Information

Copyright © 1989–1997 by Sybase, Inc. All rights reserved.

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

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. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

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 ServerConnect, Open Solutions, Optima++, PB-Gen, PC APT-Execute, PC DB-Net, PC Net Library, Power++, Power AMC, PowerBuilt, PowerBuilt with PowerBuilder, PowerDesigner, Power J, PowerScript, PowerSite, PowerSocket, Powersoft Portfolio, Power Through Knowledge, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, Quickstart Datamart, Replication Agent, Replication Driver, Replication Server Manager, Report-Execute, Report Workbench, Resource Manager, RW-DisplayLib, RW-Library, SAFE, SDF, Secure SQL Server, Secure SQL Toolset, Security Guardian, SKILS, smart.partners, smart.parts, smart.script, SQL Anywhere, SQL Central, SQL Code Checker, SQL Edit, SQL Edit / TPU, SQL Modeler, SQL Remote, SQL Server, SQL Server / CFT, SQL Server / DBM, SQL Server Manager, SQL Server SNMP SubAgent, SQL Station, SQL Toolset, Sybase Client / Server Interfaces, Sybase Development Framework, Sybase Gateways, Sybase IQ, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase Synergy Program, Sybase Virtual Server Architecture, Sybase User Workbench, SybaseWare, SyBooks, System 10, System 11, the System XI logo, SystemTools, Tabular Data Stream, The Architecture for Change, The Enterprise Client / Server Company, The Model for Client / Server Solutions, The Online Information Center, Translation Toolkit, Turning Imagination Into Reality, Unibom, Unilib, Uninull, Unisep, Unistring, Viewer, Visual Components, VisualSpeller, WarehouseArchitect, WarehouseNow, Warehouse WORKS, Watcom, Watcom SQL, Watcom SQL Server, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, and XA-Server are trademarks of Sybase, Inc. 6 / 97

All other company and product names used herein may be trademarks or registered trademarks of their respective companies.

Table of Contents

List of Figures

List of Tables

About This Book

The *Adaptive Server Reference Manual* is a three-volume guide to Sybase® Adaptive Server™ Enterprise and the Transact-SQL® language. This volume, Volume 1, includes information about Transact-SQL commands and built-in functions. Volume 2 contains information about system procedures, catalog stored procedures, extended stored procedures, and **dbcc** stored procedures. Volume 3 contains information about datatypes, the system tables, expressions and identifiers, SQLSTATE errors, reserved words, and an index for all three volumes.

Audience

The *Adaptive Server Reference Manual* is intended as a reference tool for Transact-SQL users of all levels. It provides basic syntax and usage information for every command, function, system procedure, catalog stored procedure, and extended stored procedure.

How to Use This Book

This manual consists of the following chapters:

- Chapter 1, "Transact-SQL Commands," contains reference information for every Transact-SQL command. Particularly complex commands, such as **select**, are divided into subsections. For example, there are reference pages on the **compute** clause and on the **group by** and **having** clauses of the **select** command.
- Chapter 2, "Transact-SQL Functions," contains reference information for the Adaptive Server aggregate functions, datatype conversion functions, date functions, mathematical functions, row aggregate functions, string functions, system functions, and text and image functions.

Chapter 6, "**dbcc Stored Procedures,"Related 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 SyBooks™-on-the-Web.

- The Adaptive Server installation documentation for your platform – describes installation and upgrade procedures for all Adaptive Server and related Sybase products.
- The Adaptive Server configuration documentation for your platform – describes configuring a server, creating network connections, configuring for optional functionality, such as auditing, installing most optional system databases, and performing operating system administration tasks.
- *What's New in Adaptive Server Enterprise Release 11.5?* – describes the new features in Adaptive Server release 11.5, the system changes added to support those features, and the changes that may affect your existing applications.
- *Navigating the Documentation for Adaptive Server* – an electronic interface for using Adaptive Server. This online document provides links to the concepts and syntax in the documentation that are relevant to each task.
- *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 and user and system databases, and specifying character conversion, international language, and sort order settings.
- *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.

- *Security Administration Guide* – explains how to use the security features provided by Adaptive Server to control user access to data. This manual includes information about how to add users to Adaptive Server, administer both system and user-defined roles, grant database access to users, and manage remote Adaptive Servers.
- *Security Features User's Guide* – provides instructions and guidelines for using the security options provided in Adaptive Server from the perspective of the non-administrative user.
- *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 for Adaptive Server Enterprise and OmniConnect* – explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.
- *Adaptive Server Glossary* – defines technical terms used in the Adaptive Server documentation.
- *Master Index for Adaptive Server Publications* – combines the indexes of the *Adaptive Server Reference Manual, Component Integration Services User's Guide, Performance and Tuning Guide, Security Administration Guide, Security Features User's Guide, System Administration Guide,* and *Transact-SQL User's Guide*.

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 Used in This Manual

Formatting SQL Statements

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

SQL Syntax Conventions

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

Table 1: Syntax statement conventions

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

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

```
sp_dropdevice [device_name]
```

or, for a command with more options:

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

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

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

```
select * from publishers
```

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

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

(3 rows affected)
```

Case

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.

Note that 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 "Changing the Default Character Set, Sort Order, or Language" in Chapter 13, "Configuring Character Sets, Sort Orders, and Languages" in the *System Administration Guide* for more information.

Obligatory Options {You Must Choose At Least One}

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

```
{die_on_your_feet | live_on_your_knees |
live_on_your_feet}
```

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

```
{cash, check, credit}
```

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

- **One Item in Square Brackets:** You don't have to choose it.

  `[anchovies]`

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

  `[beans | rice | sweet_potatoes]`

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

  `[extra_cheese, avocados, sour_cream]`

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

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

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

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

Expressions

Adaptive Server syntax statements use several different 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

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

Usage	Definition
integer_expr	Any integer expression, or an expression that implicitly converts to an integer value
numeric_expr	Any numeric expression that returns a single value
char_expr	Any expression that returns a single character-type value
binary_expression	An expression that returns a single *binary* or *varbinary* value

If You Need Help

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.

Commands

1 Transact-SQL Commands

This chapter describes commands, clauses, and other elements used to construct a Transact-SQL statement. Table 1-1 lists the elements discussed in this chapter and describes the function of each element.

Table 1-1: Transact-SQL commands

Command	Description
alter database	Increases the amount of space allocated to a database.
alter role	Defines mutually exclusive relationships between roles and adds, drops, and changes passwords for roles.
alter table	Adds new columns; adds, changes, or drops constraints, changes constraints; partitions or unpartitions an existing table.
begin...end	Encloses a series of SQL statements so that control-of-flow language, such as **if...else**, can affect the performance of the whole group.
begin transaction	Marks the starting point of a user-defined transaction.
break	Causes an exit from a **while** loop. **break** is often activated by an **if** test.
case	Allows SQL expressions to be written for conditional values. **case** expressions can be used anywhere a value expression can be used.
checkpoint	Writes all **dirty** pages (pages that have been updated since they were last written) to the database device.
close	Deactivates a cursor.
commit	Marks the ending point of a user-defined transaction.
compute Clause	Generates summary values that appear as additional rows in the query results.
connect to...disconnect	Specifies the server to which a passthrough connection is required.
continue	Causes the **while** loop to restart. **continue** is often activated by an **if** test.
create database	Creates a new database.
create default	Specifies a value to insert in a column (or in all columns of a user-defined datatype) if no value is explicitly supplied at insert time.

Table 1-1: Transact-SQL commands (continued)

Command	Description
create existing table	Confirms that the current remote table information matches the information that is stored in *column_list*, and verifies the existence of the underlying object.
create index	Creates an index on one or more columns in a table.
create procedure	Creates a stored procedure that can take one or more user-supplied parameters.
create role	Creates a user-defined role.
create rule	Specifies the domain of acceptable values for a particular column or for any column of a user-defined datatype.
create schema	Creates a new collection of tables, views and permissions for a database user.
create table	Creates new tables and optional integrity constraints.
create trigger	Creates a trigger, a type of stored procedure often used for enforcing integrity constraints. A trigger executes automatically when a user attempts a specified data modification statement on a specified table.
create view	Creates a view, which is an alternative way of looking at the data in one or more tables.
dbcc	Database Consistency Checker (**dbcc**) checks the logical and physical consistency of a database. Use **dbcc** regularly as a periodic check or if you suspect any damage.
deallocate cursor	Makes a cursor inaccessible and releases all memory resources committed to that cursor.
declare	Declares the name and type of local variables for a batch or procedure.
declare cursor	Defines a cursor.
delete	Removes rows from a table.
disk init	Makes a physical device or file usable by Adaptive Server.
disk mirror	Creates a software mirror that immediately takes over when the primary device fails.
disk refit	Rebuilds the *master* database's *sysusages* and *sysdatabases* system tables from information contained in *sysdevices*. Use **disk refit** after **disk reinit** as part of the procedure to restore the *master* database.

Table 1-1: Transact-SQL commands (continued)

Command	Description
disk refit	Rebuilds the *master* database's *sysdevices* system table. Use **disk reinit** as part of the procedure to restore the *master* database.
disk remirror	Reenables disk mirroring after it is stopped by failure of a mirrored device or temporarily disabled by the **disk unmirror** command.
disk unmirror	Disables either the original device or its mirror, allowing hardware maintenance or the changing of a hardware device.
drop database	Removes one or more databases from a Adaptive Server.
drop default	Removes a user-defined default.
drop index	Removes an index from a table in the current database.
drop procedure	Removes user-defined stored procedures.
drop role	Drops a user-defined role.
drop rule	Removes a user-defined rule.
drop table	Removes a table definition and all of its data, indexes, triggers, and permission specifications from the database.
drop trigger	Removes a trigger.
drop view	Removes one or more views from the current database.
dump database	Makes a backup copy of the entire database, including the transaction log, in a form that can be read in with **load database**. Dumps and loads are performed through Backup Server.
dump transaction	Makes a copy of a transaction log and removes the inactive portion.
execute	Runs a system procedure or a user-defined stored procedure.
fetch	Returns a row or a set of rows from a cursor result set.
goto Label	Branches to a user-defined label.
grant	Assigns permissions to users or to user-defined roles.
group by and having Clauses	Used in **select** statements to divide a table into groups and to return only groups that match conditions in the having clause.
if...else	Imposes conditions on the execution of a SQL statement.
insert	Adds new rows to a table or view.

Table 1-1: Transact-SQL commands (continued)

Command	Description
kill	Kills a process.
load database	Loads a backup copy of a user database, including its transaction log.
load transaction	Loads a backup copy of the transaction log.
online database	Marks a database available for public use after a normal load sequence and, if needed, upgrades a loaded database and transaction log dumps to the current version of Adaptive Server.
open	Opens a cursor for processing.
order by Clause	Returns query results in the specified column(s) in sorted order.
prepare transaction	Used by DB-Library™ in a two-phase commit application to see if a server is prepared to commit a transaction.
print	Prints a user-defined message on the user’s screen.
raiserror	Prints a user-defined error message on the user’s screen and sets a system flag to record that an error condition has occurred.
readtext	Reads *text* and *image* values, starting from a specified offset and reading a specified number of bytes or characters.
reconfigure	The **reconfigure** command currently has no effect; it is included to allow existing scripts to run without modification. In previous releases, **reconfigure** was required after the **sp_configure** system procedure to implement new configuration parameter settings.
return	Exits from a batch or procedure unconditionally, optionally providing a return status. Statements following **return** are not executed.
revoke	Revokes permissions or roles from users or roles.
rollback	Rolls a user-defined transaction back to the last savepoint inside the transaction or to the beginning of the transaction.
rollback trigger	Rolls back the work done in a trigger, including the update that caused the trigger to fire, and issues an optional **raiserror** statement.
save transaction	Sets a savepoint within a transaction.
select	Retrieves rows from database objects.

Table 1-1: Transact-SQL commands (continued)

Command	Description
set	Sets Adaptive Server query-processing options for the duration of the user’s work session. Can be used to set some options inside a trigger or stored procedure. Can also be used to activate or deactivate a role in the current session.
setuser	Allows a Database Owner to impersonate another user.
shutdown	Shuts down Adaptive Server or a Backup Server^{TM}. This command can be issued only by a System Administrator.
truncate table	Removes all rows from a table.
union Operator	Returns a single result set that combines the results of two or more queries. Duplicate rows are eliminated from the result set unless the **all** keyword is specified.
update	Changes data in existing rows, either by adding data or by modifying existing data.
update all statistics	Updates all statistics information for a given table.
update partition statistics	Updates information about the number of pages in each partition for a partitioned table.
update statistics	Updates information about the distribution of key values in specified indexes.
use	Specifies the database with which you want to work.
waitfor	Specifies a specific time, a time interval, or an event for the execution of a statement block, stored procedure, or transaction.
where Clause	Sets the search conditions in a **select**, **insert**, **update**, or **delete** statement.
while	Sets a condition for the repeated execution of a statement or statement block. The statement(s) execute repeatedly, as long as the specified condition is true.
writetext	Permits non-logged, interactive updating of an existing *text* or *image* column.

alter database

Function

Increases the amount of space allocated to a database.

Syntax

```
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]
```

Keywords and Options

database_name – is the name of the database. The database name can be a literal, a variable, or a stored procedure parameter.

on – indicates a size and / or location for the database extension. If you have your log and data on separate device fragments, use this clause for the data device and the **log on** clause for the log device.

default – indicates that **alter database** can put the database extension on any default database device(s) (as shown by **sp_helpdevice**). To specify a size for the database extension without specifying the exact location, use this command:

```
on default = size
```

To change a database device's status to default, use the system procedure **sp_diskdefault**.

database_device – is the name of the database device on which to locate the database extension. A database can occupy more than one database device with different amounts of space on each. Add database devices to Adaptive Server with **disk init**.

size – is the amount of space, in megabytes, to allocate to the database extension. The minimum extension is 1MB (512 2K pages). The default value is 1MB.

log on – indicates that you want to specify additional space for the database's transaction logs. The **log on** clause uses the same defaults as the **on** clause.

with override – forces Adaptive Server to accept your device specifications, even if they mix data and transaction logs on the same device, thereby endangering up-to-the-minute recoverability for your database. If you attempt to mix log and data on the same device without using this clause, the **alter database** command fails. If you mix log and data, and use **with override**, you are warned, but the command succeeds.

for load – is used only after **create database for load**, when you must re-create the space allocations and segment usage of the database being loaded from a dump.

Examples

1.
```
alter database mydb
```
Adds 1MB to the database *mydb* on a default database device.

2.
```
alter database pubs2
on newdata = 3
```
Adds 3MB to the space allocated for the *pubs2* database on the database device named *newdata*.

3.
```
alter database production
on userdata1 = 10
log on logdev = 2
```
Adds 10MB of space for data on *userdata1* and 2MB for the log on *logdev*.

Comments

Restrictions

- You must be using the *master* database, or executing a stored procedure in the *master* database, to use **alter database**.
- If Adaptive Server cannot allocate the requested space, it comes as close as possible per device and prints a message telling how much space has been allocated on each database device.
- You can expand the *master* database only on the master device. An attempt to use **alter database** to expand the *master* database to any other database device results in an error message. Here is an example of the correct statement for modifying the *master* database on the master device:

```
alter database master on master = 1
```

- The maximum number of device fragments for any database is 128. Each time you allocate space on a database device with **create database** or **alter database**, that allocation represents a device fragment, and the allocation is entered as a row in *sysusages*.
- If you use **alter database** on a database that is in the process of being dumped, the **alter database** command cannot complete until the dump finishes. Adaptive Server locks the in-memory map of database space use during a dump. If you issue an **alter database** command while this in-memory map is locked, Adaptive Server updates the map from the disk after the dump completes. If you interrupt **alter database**, Adaptive Server instructs you to run **sp_dbremap**. If you fail to run **sp_dbremap**, the space you added does not become available to Adaptive Server until the next reboot.
- You can use **alter database on** *database_device* on an offline database.

Backing Up *master* After Allocating More Space

- Back up the *master* database with the **dump database** command after each use of **alter database**. This makes recovery easier and safer in case *master* becomes damaged.
- If you use **alter database** and fail to back up *master*, you may be able to recover the changes with **disk refit**.

Placing the Log on a Separate Device

- To increase the amount of storage space allocated for the transaction log when you have used the **log on** extension to **create database**, give the name of the log's device in the **log on** clause when you issue the **alter database** command.
- If you did not use the **log on** extension of **create database** to place your logs on a separate device, you may not be able to recover fully in case of a hard disk crash. In this case, you can extend your logs by using **alter database** with the **log on** clause and then using **sp_logdevice**.

Getting Help on Space Usage

- To see the names, sizes, and usage of device fragments already in use by a database, execute **sp_helpdb** *dbname*.
- To see how much space the current database is using, execute **sp_spaceused**.

The *system* and *default* Segments

- The *system* and *default* segments are mapped to each new database device included in the **on** clause of an **alter database** command. To unmap these segments, use **sp_dropsegment**.
- When you use **alter database** (without **override)** to extend a database on a device already in use by that database, the segments mapped to that device are also extended. If you use the **override** clause, all device fragments named in the **on** clause become system / default segments, and all device fragments named in the **log on** clause become log segments.

Using *alter database* to Awaken Sleeping Processes

- If user processes are suspended because they have reached a last-chance threshold on a log segment, use **alter database** to add space to the log segment. The processes awaken when the amount of free space exceeds the last-chance threshold.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

alter database permission defaults to the Database Owner. System Administrators can also alter databases.

See Also

Commands	**create database**, **disk init**, **drop database**, **load database**
System procedures	**sp_addsegment**, **sp_dropsegment**, **sp_helpdb**, **sp_helpsegment**, **sp_logdevice**, **sp_renamedb**, **sp_spaceused**

alter role

Function

Defines mutually exclusive relationships between roles and adds, drops, and changes passwords for roles.

Syntax

```
alter role role1 { add | drop } exclusive {
   membership | activation } role2

alter role role_name { add passwd "password"| drop
   passwd }
```

Keywords and Options

role1 – is either role in a mutually exclusive relationship.

add – adds a role in a mutually exclusive relationship; adds a password to a role.

drop – drops a role in a mutually exclusive relationship; drops a password from a role.

exclusive – makes both named roles mutually exclusive.

membership – does not allow you to grant users both roles at the same time.

activation – allows you to grant a user both roles at the same time, but does not allow the user to activate both roles at the same time.

role2 – is the other role in a mutually exclusive relationship.

role_name – is the name of the role for which you want to add, drop, or change a password.

passwd – adds a password to a role.

password – is the password to add to a role.

Examples

1.
   ```
   alter role intern_role add exclusive membership
   specialist_role
   ```

   Defines two roles as mutually exclusive.

2. **`alter role specialist_role add exclusive membership intern_role`**
   `alter role intern_role add exclusive activation surgeon_role`

   Defines roles as mutually exclusive at the membership level and at the activation level.

3. **`alter role doctor_role add passwd "physician"`**

   Adds a password to an existing role.

4. **`alter role doctor_role drop passwd`**

   Drops a password from an existing role.

Comments

- The **alter role** command defines mutually exclusive relationships between roles and adds, drops, and changes passwords for roles.
- For more information on altering roles, see Chapter 4, "Administering Roles," in the *Security Administration Guide*.

Mutually Exclusive Roles

- You need not specify the roles in a mutually exclusive relationship or role hierarchy in any particular order.
- You can use mutual exclusivity with role hierarchy to impose constraints on user-defined roles.
- Mutually exclusive membership is a stronger restriction than mutually exclusive activation. If you define two roles as mutually exclusive at membership, they are implicitly mutually exclusive at activation.
- If you define two roles as mutually exclusive at membership, defining them as mutually exclusive at activation has no effect on the membership definitions. Mutual exclusivity at activation is added and dropped independently of mutual exclusivity at membership.
- You cannot define two roles as having mutually exclusive after granting both roles to users or roles. Revoke either granted role from existing grantees before attempting to define the roles as mutually exclusive on the membership level.
- If two roles are defined as mutually exclusive at activation, the System Security Officer can assign both roles to the same user, but the user cannot activate both roles at the same time.

- If the System Security Officer defines two roles as mutually exclusive at activation, and users have already activated both roles or, by default, have set both roles to activate at login, Adaptive Server makes the roles mutually exclusive, but issues a warning message naming specific users with conflicting roles. The users' activated roles do not change.

Changing Passwords for Roles

- To change the password for a role, first drop the existing password, and then add the new password, as follows:

```
alter role doctor_role drop passwd
alter role doctor_role add passwd "physician"
```

  Passwords must be at least 6 characters in length and must conform to the rules for identifiers. You cannot use variables for passwords.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Only a System Security Officer can execute **alter role**.

See Also

Commands	**create role**, **drop role**, **grant**, **revoke**, **set**
Functions	**mut_excl_roles**, **proc_role**, **role_contain**, **role_id**, **role_name**
System Procedures	**sp_activeroles**, **sp_displaylogin**, **sp_displayroles**, **sp_modifylogin**

alter table

Function

Adds new columns; adds, changes, or drops constraints; partitions or unpartitions an existing table.

Syntax

```
alter table [database.[owner].]table_name
   {add column_name datatype
       [default {constant_expression | user | null}]
       {[{identity | null}]
       | [[constraint constraint_name]
         {{unique | primary key}
           [clustered | nonclustered]
           [with {fillfactor | max_rows_per_page} = x]
                    [on segment_name]
           | references [[database.]owner.]ref_table
              [(ref_column)]
           | check (search_condition)}]}...
       {[, next_column]}...

   | add {[constraint constraint_name]
       {unique | primary key}
         [clustered | nonclustered]
         (column_name [{, column_name}...])
         [with {fillfactor | max_rows_per_page} = x]
              [on segment_name]
      | foreign key (column_name [{, column_name}...])
           references [[database.]owner.]ref_table
               [(ref_column [{, ref_column}...])]
       | check (search_condition)}

   | drop constraint constraint_name

   | replace column_name
       default {constant_expression | user | null}

   | partition number_of_partitions

   | unpartition}
```

Keywords and Options

table_name – is the name of the table to change. Specify the database name if the table is in another database, and specify the owner's name if more than one table of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

add – specifies the name of the column or constraint to add to the table.

If Component Integration Services is enabled, you cannot use **add** for remote servers.

column_name – is the name of a column in that table.

datatype – is any system datatype except *bit* or any user-defined datatype except those based on *bit*.

default – specifies a default value for a column. If you specify a default and the user does not provide a value for this column when inserting data, Adaptive Server inserts this value. The default can be a *constant_expression*, **user** (to insert the name of the user who is inserting the data), or **null** (to insert the null value).

Adaptive Server generates a name for the default in the form of *tabname_colname_objid,* where *tabname* is the first 10 characters of the table name, *colname* is the first 5 characters of the column name, and *objid* is the object ID number for the default. Setting the default to **null** drops the default.

If Component Integration Services is enabled, you cannot use **default** for remote servers.

constant_expression – is a constant expression to use as a default value for a column. It cannot include the name of any columns or other database objects, but can include built-in functions. This default value must be compatible with the datatype of the column.

user – specifies that Adaptive Server should insert the user name as the default if the user does not supply a value. The datatype of the column must be either *char*(30), *varchar*(30), or a type that Adaptive Server implicitly converts to *char*; however, if the datatype is not *char*(30) or *varchar*(30), truncation may occur.

null – specifies that Adaptive Server should insert the null value as the default if the user does not supply a value and no default exists. The column must allow null values.

identity – indicates that the column has the IDENTITY property. Each table in a database can have one IDENTITY column of type *numeric* and scale zero. IDENTITY columns are not updatable and do not allow nulls.

IDENTITY columns store sequential numbers, such as invoice numbers or employee numbers, automatically generated by Adaptive Server. The value of the IDENTITY column uniquely identifies each row in a table.

If Component Integration Services is enabled, you cannot use **identity** for remote servers.

constraint – introduces the name of an integrity constraint.

If Component Integration Services is enabled, you cannot use **constraint** for remote servers.

constraint_name – is the name of the constraint. It must conform to the rules for identifiers and be unique in the database. If you do not specify the name for a table-level constraint, Adaptive Server generates a name in the form of *tabname_colname_objectid*, where *tabname* is the first 10 characters of the table name, *colname* is the first 5 characters of the column name, and *objectid* is the object ID number for the constraint. If you do not specify the name for a unique or primary key constraint, Adaptive Server generates a name in the format *tabname_colname_tabindid*, where *tabindid* is a string concatenation of the table ID and index ID.

Constraints do not apply to the data that already exists in the table at the time the constraint is added.

unique – constrains the values in the indicated column or columns so that no two rows can have the same non-null value. This constraint creates a unique index that can be dropped only if the constraint is dropped. You cannot use this option along with the **null** option described above.

primary key – constrains the values in the indicated column or columns so that no two rows can have the same value and so that the value cannot be NULL. This constraint creates a unique index that can be dropped only if the constraint is dropped.

clustered | **nonclustered** – specifies that the index created by a **unique** or **primary key** constraint is a clustered or nonclustered index. **clustered** is the default (unless a clustered index already exists for the table) for primary key constraints; **nonclustered** is the default for unique constraints. There can be only one clustered index per table. See **create index** for more information.

fillfactor – specifies how full to make each page when Adaptive Server creates a new index on existing data. The **fillfactor** percentage is relevant only when the index is created. As the data changes, the pages are not maintained at any particular level of fullness.

The default for **fillfactor** is 0; this is used when you do not include **with fillfactor** in the **create index** statement (unless the value has been changed with **sp_configure**). When specifying a **fillfactor**, use a value between 1 and 100.

A **fillfactor** of 0 creates clustered indexes with completely full pages and nonclustered indexes with completely full leaf pages. It leaves a comfortable amount of space within the index B-tree in both clustered and nonclustered indexes. There is seldom a reason to change the **fillfactor**.

If the **fillfactor** is set to 100, Adaptive Server creates both clustered and nonclustered indexes with each page 100 percent full. A **fillfactor** of 100 makes sense only for read-only tables—tables to which no additional data will ever be added.

fillfactor values smaller than 100 (except 0, which is a special case) cause Adaptive Server to create new indexes with pages that are not completely full. A **fillfactor** of 10 might be a reasonable choice if you are creating an index on a table that will eventually hold a great deal more data, but small **fillfactor** values cause each index (or index and data) to take more storage space.

◆ ***WARNING!***

Creating a clustered index with a fillfactor affects the amount of storage space your data occupies, since Adaptive Server redistributes the data as it creates the clustered index.

max_rows_per_page – limits the number of rows on data pages and the leaf level pages of indexes. Unlike **fillfactor**, the **max_rows_per_page** value is maintained until it is changed with **sp_chgattribute**.

If you do not specify a value for **max_rows_per_page**, Adaptive Server uses a value of 0 when creating the index. When specifying **max_rows_per_page** for data pages, use a value between 0 and 256. The maximum number of rows per page for nonclustered indexes depends on the size of the index key; Adaptive Server returns an error message if the specified value is too high.

For indexes created by constraints, a **max_rows_per_page** setting of 0 creates clustered indexes with full pages and nonclustered indexes with full leaf pages. A setting of 0 leaves a comfortable amount of space within the index B-tree in both clustered and nonclustered indexes.

If **max_rows_per_page** is set to 1, Adaptive Server creates both clustered and nonclustered leaf index pages with one row per page at the leaf level. You can use this to reduce lock contention on frequently accessed data.

Low **max_rows_per_page** values cause Adaptive Server to create new indexes with pages that are not completely full, uses more storage space, and may cause more page splits.

◆ ***WARNING!***

Creating a clustered index with max_rows_per_page can affect the amount of storage space your data occupies, since Adaptive Server redistributes the data as it creates the clustered index.

on *segment_name* – specifies that the index is to be created on the named segment. Before the **on** *segment_name* option can be used, the device must be initialized with **disk init**, and the segment must be added to the database with the **sp_addsegment** system procedure. See your System Administrator or use **sp_helpsegment** for a list of the segment names available in your database.

If you specify **clustered** and use the **on** *segment_name* option, the entire table migrates to the segment you specify, since the leaf level of the index contains the actual data pages.

references – specifies a column list for a referential integrity constraint. You can specify only one column value for a column-constraint. By including this constraint with a table that references another table, any data inserted into the **referencing** table must already exist in the **referenced** table.

To use this constraint, you must have **references** permission on the referenced table. The specified columns in the referenced table must be constrained by a unique index (created by either a **unique** constraint or a **create index** statement). If no columns are specified, there must be a **primary key** constraint on the appropriate columns in the referenced table. Also, the datatypes of the referencing table columns must exactly match the datatype of the referenced table columns.

If Component Integration Services is enabled, you cannot use **references** for remote servers.

foreign key – specifies that the listed column(s) are foreign keys in this table whose matching primary keys are the columns listed in the **references** clause.

ref_table – is the name of the table that contains the referenced columns. You can reference tables in another database. Constraints can reference up to 192 user tables and internally generated worktables. Use the system procedure **sp_helpconstraint** to check a table's referential constraints.

ref_column – is the name of the column or columns in the referenced table.

check – specifies a *search_condition* constraint that Adaptive Server enforces for all the rows in the table.

If Component Integration Services is enabled, you cannot use **check** for remote servers.

search_condition – is a boolean expression that defines the **check** constraint on the column values. These constraints can include:

- A list of constant expressions introduced with **in**.
- A set of conditions, which may contain wildcard characters, introduced with **like**.

An expression can include arithmetic operations and Transact-SQL functions. The *search_condition* cannot contain subqueries, aggregate functions, parameters, or host variables.

next_column – includes additional column definitions (separated by commas) using the same syntax described for a column definition.

drop *constraint constraint_name* – specifies the name of a constraint to drop from the table.

If Component Integration Services is enabled, you cannot use **drop** for remote servers.

replace – specifies the column whose default value you want to change with the new value specified by a following **default** clause.

If Component Integration Services is enabled, you cannot use **replace** for remote servers.

partition *number_of_partitions* – creates multiple database page chains for the table. Adaptive Server can perform concurrent insertion operations into the last page of each chain. *number_of_partitions* must be a positive integer greater than or equal to 2. Each partition requires an additional control page; lack of disk space can limit the number of partitions you can create in a table. Lack of memory can limit the number of partitioned tables you can access.

If Component Integration Services is enabled, you cannot use **partition** for remote servers.

unpartition – creates a single page chain for the table by concatenating subsequent page chains with the first one.

If Component Integration Services is enabled, you cannot use **unpartition** for remote servers.

Examples

1.
```
alter table publishers
add manager_name varchar(40) null
```
Adds a column to a table. For each existing row in the table, Adaptive Server assigns a NULL column value.

2.
```
alter table sales_daily
add ord_num numeric(5,0) identity
```
Adds an IDENTITY column to a table. For each existing row in the table, Adaptive Server assigns a unique, sequential column value. Note that the IDENTITY column has type *numeric* and a scale of zero. The precision determines the maximum value (10^5 **-** 1, or 99,999) that can be inserted into the column.

3.
```
alter table authors
add constraint au_identification
primary key (au_id, au_lname, au_fname)
```
Adds a primary key constraint to the *authors* table. If there is an existing primary key or unique constraint on the table, the existing constraint must be dropped first (see example 4).

4.
```
alter table titles
drop constraint au_identification
```
Drops the *au_identification* constraint.

5. **`alter table authors`**
   `replace phone default null`

   Changes the default constraint on the *phone* column in the *authors* table to insert the value NULL if the user does not enter a value.

6. **`alter table titleauthor partition 5`**

   Creates four new page chains for the *titleauthor* table. After the table is partitioned, existing data remains in the first partition. New rows, however, are inserted into all five partitions.

7. **`alter table titleauthor unpartition`**
   `alter table titleauthor partition 6`

   Concatenates all page chains of the *titleauthor* table and then repartitions it with six partitions.

Comments

- A column added with **alter table** is not visible to stored procedures that **select** * from that table. To reference the new column, drop the procedures and re-create them.

Restrictions

- You cannot add a column of datatype *bit* to an existing table.
- The number of columns in a table cannot exceed 250. The maximum number of bytes per row is 1962.
- You cannot delete columns from a table. Use **select into** to create a new table with a different column structure or sort order.

◆ ***WARNING!***

Do not alter the system tables.

- You cannot partition a system table or a table that is already partitioned.
- You cannot issue the **alter table command** with a **partition** or **unpartition** clause within a user-defined transaction.

Getting Information About Tables

- For information about a table and its columns, use **sp_help**.
- To rename a table, execute the system procedure **sp_rename** (do not rename the system tables).

- For information about integrity constraints (**unique**, **primary key, references**, and **check**) or the **default** clause, see **create table** in this chapter.

Using Cross-Database Referential Integrity Constraints

- When you create a cross-database constraint, Adaptive Server stores the following information in the *sysreferences* system table of each database:

Table 1-2: Information stored about referential integrity constraints

Information Stored in *sysreferences*	**Columns with Information About the Referenced Table**	**Columns with Information About the Referencing Table**
Key column IDs	*refkey1* through *refkey16*	*fokey1* through *fokey16*
Table ID	*reftabid*	*tableid*
Database name	*pmrydbname*	*frgndbname*

- When you drop a referencing table or its database, Adaptive Server removes the foreign key information from the referenced database.
- Because the referencing table depends on information from the referenced table, Adaptive Server does not allow you to:
  - Drop the referenced table,
  - Drop the external database that contains the referenced table, or
  - Rename either database with **sp_renamedb**.

  You must first remove the cross-database constraint with **alter table.**
- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases could cause database corruption.

- The *sysreferences* system table stores the **name**—not the ID number—of the external database. Adaptive Server cannot guarantee referential integrity if you use **load database** to change the database name or to load it onto a different server.

◆ ***WARNING!***

Before dumping a database in order to load it with a different name or move it to another Adaptive Server, use alter table to drop all external referential integrity constraints.

Changing Defaults

- You can create column defaults in two ways: by declaring the default as a column constraint in the **create table** or **alter table** statement or by creating the default using the **create default** statement and binding it to a column using **sp_bindefault**.
- You cannot replace a user-defined default bound to the column with **sp_bindefault**. Unbind the default with **sp_unbindefault** first.
- If you declare a default column value with **create table** or **alter table**, you cannot bind a default to that column with **sp_bindefault**. Drop the default by altering it to NULL, and then bind the user-defined default. Changing the default to NULL unbinds the default and deletes it from the *sysobjects* table.

Partitioning Tables for Improved Insert Performance

- Partitioning a table with the **partition** clause of the **alter table** command creates additional page chains, making multiple last pages available at any given time for concurrent insert operations. This improves insert performance by reducing page contention and, if the segment containing the table is spread over multiple physical devices, by reducing I/O contention while the server flushes data from cache to disk.
- When you partition a table, Adaptive Server allocates a control page for each partition, including the first partition. The existing page chain becomes part of the first partition. Adaptive Server creates a first page for each subsequent partition. Since each partition has its own control page, partitioned tables require slightly more disk space than unpartitioned tables.
- You can partition both empty tables and those that contain data. Partitioning a table does **not** move data; existing data remains

where it was originally stored, in the first partition. For best performance, partition a table **before** inserting data.

- You cannot partition a system table or a table that is already partitioned. You can partition a table that contains *text* and *image* columns; however, partitioning has no effect on the way Adaptive Server stores the *text* and *image* columns.
- Once you have partitioned a table, you cannot use the **truncate table** command or the **sp_placeobject** system procedure on it.
- To change the number of partitions in a table, first use the **unpartition** clause of **alter table** to concatenate all existing page chains, and then use the **partition** clause of **alter table** to repartition the table.
- If you unpartition a table, recompile the query plans of any dependent procedures. Unpartitioning does not automatically recompile procedures.
- When you unpartition a table with the **unpartition** clause of the **alter table** command, Adaptive Server deallocates all control pages, including that of the first partition, and concatenates the page chains. The resulting single page chain contains no empty pages, with the possible exception of the first page. Unpartitioning a table does **not** move data.

Adding IDENTITY Columns

- When adding an IDENTITY column to a table, make sure the column precision is large enough to accommodate the number of existing rows. If the number of rows exceeds $10^{\text{PRECISION}} - 1$, Adaptive Server prints an error message and does not add the column.
- When adding an IDENTITY column to a table, Adaptive Server:
  - Locks the table until all the IDENTITY column values have been generated. If a table contains a large number of rows, this process may be time-consuming.
  - Assigns each existing row a unique, sequential IDENTITY column value, beginning with the value 1.
  - Logs each insert operation into the table. Use **dump transaction** to clear the database's transaction log before adding an IDENTITY column to a table with a large number of rows.
- Each time you insert a row into the table, Adaptive Server generates an IDENTITY column value that is one higher than the

last value. This value takes precedence over any defaults declared for the column in the **alter table** statement or bound to it with **sp_bindefault**.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Transact-SQL extension.	See Chapter 7, "System and User-Defined Datatypes" for datatype compliance information.

Permissions

alter table permission defaults to the table owner; it cannot be transferred except to the Database Owner, who can impersonate the table owner by running the **setuser** command. A System Administrator can also alter user tables.

See Also

Commands	**create table**, **dbcc**, **drop database**, **insert**
System procedures	**sp_help**, **sp_helpartition**, **sp_rename**

begin...end

Function

Encloses a series of SQL statements so that control-of-flow language, such as **if...else**, can affect the performance of the whole group.

Syntax

```
begin
   statement block
end
```

Keywords and Options

statement block – is a series of statements enclosed by **begin** and **end**.

Examples

1.
```
if (select avg(price) from titles) < $15
begin
  update titles
  set price = price * $2
  select title, price
  from titles
  where price > $28
end
```

   Without **begin** and **end**, the **if** condition would cause execution of only one SQL statement.

2.
```
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
else
  print "Deletion successful--no sales for this
      title."
```

   Without **begin** and **end**, the **print** statement would not execute.

Comments

- **begin...end** blocks can nest within other **begin...end** blocks.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

begin...end permission defaults to all users. No permission is required to use it.

See Also

Commands	**if...else**

begin transaction

Function

Marks the starting point of a user-defined transaction.

Syntax

```
begin tran[saction] [transaction_name]
```

Keywords and Options

transaction_name – is the name assigned to this transaction. Transaction names must conform to the rules for identifiers. Use transaction names only on the outermost pair of nested **begin transaction** / **commit** or **begin transaction** / **rollback** statements.

Examples

```
1. begin transaction
     insert into publishers (pub_id) values ("9999")
   commit transaction
```

Explicitly begins a transaction for the **insert** statement.

Comments

- Define a transaction by enclosing SQL statements and / or system procedures within the phrases **begin transaction** and **commit**. If you set chained transaction mode, Adaptive Server implicitly invokes a **begin transaction** before the following statements: **delete**, **insert**, **open**, **fetch**, **select**, and **update**. You must still explicitly close the transaction with a **commit**.
- To cancel all or part of a transaction, use the **rollback** command. The **rollback** command must appear within a transaction; you cannot roll back a transaction after it is committed.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

begin transaction permission defaults to all users. No permission is required to use it.

See Also

Commands	**commit**, **rollback**, **save transaction**

break

Function

Causes an exit from a **while** loop. **break** is often activated by an **if** test.

Syntax

```
while logical_expression
        statement
   break
        statement
   continue
```

Keywords and Options

logical_expression – is an expression (a column name, constant, any combination of column names and constants connected by arithmetic or bitwise operators, or a subquery) that returns TRUE, FALSE, or NULL. If the logical expression contains a **select** statement, enclose the **select** statement in parentheses.

Examples

1.
```
while (select avg(price) from titles) < $30
 begin
  update titles
  set price = price * 2
  select max(price) from titles
  if (select max(price) from titles) > $50
     break
  else
     continue
 end
 begin
  print "Too much for the market to bear"
 end
```

If the average price is less than $30, double the prices. Then, select the maximum price. If it is less than or equal to $50, restart the **while** loop and double the prices again. If the maximum price is more than $50, exit the **while** loop and print a message.

Comments

- **break** causes an exit from a **while** loop. Statements that appear after the keyword **end**, which marks the end of the loop, are then executed.

- If two or more **while** loops are nested, the inner **break** exits to the next outermost loop. First, all the statements after the end of the inner loop run; then, the next outermost loop restarts.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

break permission defaults to all users. No permission is required to use it.

See Also

Commands	**continue**, **while**

case

Function

Supports conditional SQL expressions. **case** expressions can be used anywhere a value expression can be used.

Syntax

```
case
    when search_condition then expression
    [when search_condition then expression]...
    [else expression]
end
```

case and values syntax:

```
case expression
    when expression then expression
    [when expression then expression]...
    [else expression]
end
```

coalesce syntax:

```
coalesce(expression, expression [, expression]...)
```

nullif syntax:

```
nullif(expression, expression)
```

Keywords and Options

case – begins the **case** expression.

search_condition – is used to set conditions for the results that are selected. Search conditions for **case** expressions are similar to the search conditions in a **where** clause. Search conditions are detailed under "Selecting Rows: The where Clause" on page 2-19 in the Transact-SQL User's Guide.

when – precedes the search condition or the expression to be compared.

then – precedes the expression that specifies a result value of **case**.

coalesce – evaluates the listed expressions and returns the first non-null value. If all the expressions are null, **coalesce** returns a null.

nullif – compares the values of the two expressions. If the first expression equals the second expression, **nullif** returns NULL. If the first expression does not equal the second expression, **nullif** returns the first expression.

Examples

1.
```
select au_lname, postalcode,
    case
        when postalcode = "94705"
            then "Berkeley Author"
        when postalcode = "94609"
            then "Oakland Author"
        when postalcode = "94612"
            then "Oakland Author"
        when postalcode = "97330"
            then "Corvallis Author"
    end
from authors
```

   Selects all the authors from the *authors* table and, for certain authors, specifies the city in which they live.

2.
```
select stor_id, discount,
    coalesce (lowqty, highqty)
from discounts
```

   Returns the first occurrence of a non-NULL value in either the *lowqty* or *highqty* column of the *discounts* table:

3.
```
select stor_id, discount,
    case
        when lowqty is not NULL then lowqty
        else highqty
    end
from discounts
```

   This is an alternative way of writing example 2.

4.
```
select title,
    nullif(type, "UNDECIDED")
from titles
```

   Selects the *titles* and *type* from the *titles* table. If the book type is UNDECIDED, **nullif** returns a NULL value.

5.
```
select title,
    case
        when type = "UNDECIDED" then NULL
        else type
    end
from titles
```

   This is an alternative way of writing example 4.

Comments

- **case** expression simplifies standard SQL expressions by allowing you to express a search condition using a **when...then** construct instead of an **if** statement.
- **case** expressions can be used anywhere an expression can be used in SQL.
- **nullif** is an abbreviated form of a **case** expression. Example 5 describes an alternative way of writing **nullif**.
- **coalesce** is an abbreviated form of a **case** expression. Example 3 describes an alternative way of writing the **coalesce** statement.
- **coalesce** must be followed by at least two expressions. For example:

  ```
  select stor_id, discount,
      coalesce (highqty)
  from discounts
  ```

  results in the following error message:

```
A single coalesce element is illegal in a COALESCE expression.
```

- If your query produces a variety of different datatypes, the datatype of a **case** expression result is determined by datatype hierarchy, as described in “Datatype of Mixed-Mode Expressions” in Chapter 7, “System and User-Defined Datatypes”. If you specify two datatypes that Adaptive Server cannot implicitly convert (for example, *char* and *int*), the query fails.
- At least one result of the case expression must return a non-null value. For example:

  ```
  select price,
      coalesce (NULL, NULL, NULL)
  from titles
  ```

  results in the following error message:

```
All result expressions in a CASE expression must not be NULL.
```

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

case permission defaults to all users. No permission is required to use it.

See Also

Commands	**select, if...else, where Clause**

checkpoint

Function

Writes all **dirty** pages (pages that have been updated since they were last written) to the database device.

Syntax

```
checkpoint
```

Examples

1. **`checkpoint`**

   Writes all dirty pages in the current database to the database device, regardless of the system checkpoint schedule.

Comments

- Use **checkpoint** only as a precautionary measure in special circumstances. For example, Adaptive Server instructs you to issue the **checkpoint** command after resetting database options.
- Use **checkpoint** each time you change a database option with the system procedure **sp_dboption**.

Automatic Checkpoints

- Checkpoints caused by the **checkpoint** command supplement automatic checkpoints, which occur at intervals calculated by Adaptive Server on the basis of the configurable value for maximum acceptable recovery time.
- The **checkpoint** shortens the automatic recovery process by identifying a point at which all completed transactions are guaranteed to have been written to the database device. A typical **checkpoint** takes about 1 second, although checkpoint time varies, depending on the amount of activity on Adaptive Server.
- The automatic **checkpoint** interval is calculated by Adaptive Server on the basis of system activity and the recovery interval value in the system table *syscurconfigs*. The recovery interval determines **checkpoint** frequency by specifying the maximum amount of time it should take for the system to recover. Reset this value by executing the system procedure **sp_configure**.
- If the housekeeper task is able to flush all active buffer pools in all configured caches during the server’s idle time, it wakes up the

checkpoint task. The checkpoint task determines whether it can checkpoint the database.

Checkpoints that occur as a result of the housekeeper task are known as **free checkpoints**. They do not involve writing many dirty pages to the database device, since the housekeeper task has already done this work. They may improve recovery speed for the database.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

checkpoint permission defaults to the Database Owner. It cannot be transferred.

See Also

System procedures	**sp_configure**, **sp_dboption**

close

Function

Deactivates a cursor.

Syntax

```
close cursor_name
```

Parameters

cursor_name – is the name of the cursor to close.

Examples

1. **`close authors_crsr`**

   Closes the cursor named *authors_crsr*.

Comments

- The **close** command essentially removes the cursor's result set. The cursor position within the result set is undefined for a closed cursor.
- Adaptive Server returns an error message if the cursor is already closed or does not exist.

Standards and Compliance

Standard	Compliance Level
SQL92	Entry level compliant

Permissions

close permission defaults to all users. No permission is required to use it.

See Also

Commands	**deallocate cursor**, **declare cursor**, **fetch**, **open**

commit

Function

Marks the ending point of a user-defined transaction.

Syntax

```
commit [tran[saction] | work] [transaction_name]
```

Keywords and Options

transaction | **tran** | **work** – is optional.

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 transaction** / **commit** or **begin transaction** / **rollback** statements.

Examples

1.
```
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

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

rollback transaction percentchanged

commit transaction
```

After updating the *royaltyper* entries for the two authors, insert the savepoint *percentchanged*, then determine how a 10 percent increase in the book's price would affect the authors' royalty earnings. The transaction is rolled back to the savepoint with the **rollback transaction** command.

Comments

- Define a transaction by enclosing SQL statements and / or system procedures with the phrases **begin transaction** and **commit.** If you set the chained transaction mode, Adaptive Server implicitly invokes a **begin transaction** before the following statements: **delete, insert, open, fetch, select,** and **update.** You must still explicitly enclose the transaction with a **commit.**
- To cancel all or part of an entire transaction, use the **rollback** command. The **rollback** command must appear within a transaction. You cannot roll back a transaction after the **commit** has been entered.
- If no transaction is currently active, the **commit** or **rollback** statement has no effect on Adaptive Server.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The **commit transaction** and **commit tran** forms of the statement are Transact-SQL extensions.

Permissions

commit permission defaults to all users.

See Also

Commands	**begin transaction, rollback, save transaction**

compute Clause

Function

Generates summary values that appear as additional rows in the query results.

Syntax

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

Keywords and Options

row_aggregate – is one of the following:

Function	Meaning
sum	Total of values in the (numeric) column
avg	Average of values in the (numeric) column
min	Lowest value in the column
max	Highest value in the column
count	Number of values in the column

column_name – is the name of a column. It must be enclosed in parentheses. Only numeric columns can be used with **sum** and **avg**.

One **compute** clause can apply several aggregate functions to the same set of grouping columns (see examples 2 and 3). To create more than one group, use more than one **compute** clause (see example 5).

by – calculates the row aggregate values for subgroups. Whenever the value of the **by** item changes, row aggregate values are generated. If you use **by**, you must use **order by**.

Listing more than one item after **by** breaks a group into subgroups and applies a function at each level of grouping.

Examples

1. **select type, price**
   from titles
   where price > $12
   and type like "%cook"
   order by type, price
   compute sum(price) by type

```
type       price
---------  ------------
mod_cook          19.99
           sum
           ------------
                  19.99
type       price
---------  ------------
trad_cook         14.99
trad_cook         20.95
           sum
           ------------
                  35.94
(5 rows affected)
```

   Calculates the sum of the prices of each type of cook book that costs more than $12.

2. **select type, price, advance**
   from titles
   where price > $12
   and type like "%cook"
   order by type, price
   compute sum(price), sum(advance) by type

```
type       price      advance
---------  ---------  ------------
mod_cook       19.99          0.00
          sum         sum
           ---------  ------------
               19.99          0.00

type       price      advance
---------  ---------  ------------
trad_cook      14.99      8,000.00
trad_cook      20.95      7,000.00
           sum        sum
           ---------  ------------
               35.94     15,000.00
(5 rows affected)
```

Calculates the sum of the prices and advances for each type of cook book that costs more than $12.

3.
```
select type, price, advance
from titles
where price > $12
  and type like "%cook"
  order by type, price
compute sum(price), max(advance) by type
```

```
type      price     advance
--------- --------- -------------
mod_cook      19.99          0.00
          sum
          ---------
              19.99
                    max
                    -------------
                             0.00
type      price     advance
--------- --------- -------------
trad_cook     14.99      8,000.00
trad_cook     20.95      7,000.00
          sum
          ---------
              35.94
                     max
                     -------------
                          8,000.00
(5 rows affected)
```

Calculates the sum of the prices and maximum advances of each type of cook book that costs more than $12.

4.
```
select type, pub_id, price
from titles
where price > $10
  and type = "psychology"
  order by type, pub_id, price
compute  sum(price) by type, pub_id
```

```
type         pub_id    price
------------ --------- -----------
psychology   0736          10.95
psychology   0736          19.99
                         sum
                         ---------
                            30.94
```

```
type         pub_id    price
------------ --------- ---------
psychology   0877          21.59
                       sum
                       ---------
                           21.59
(5 rows affected)
```

Breaks on *type* and *pub_id* and calculates the sum of the prices of psychology books by a combination of type and publisher ID.

5.
```
select type, pub_id, price
from titles
where price > $10
  and 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          10.95
psychology   0736          19.99
                       sum
                       ---------
                           30.94
type         pub_id    price
------------ --------- ---------
psychology   0877          21.59
                       sum
                       ---------
                           21.59
                       sum
                       ---------
                           52.53
(6 rows affected)
```

Calculates the grand total of the prices of psychology books that cost more than $10 in addition to calculating sums by *type* and *pub_id*.

6.
```
select type, price, advance
from titles
where price > $10
  and type like "%cook"
compute sum(price), sum(advance)
```

```
type       price        advance
---------  -----------  --------------
mod_cook         19.99            0.00
trad_cook        20.95        8,000.00
trad_cook        11.95        4,000.00
trad_cook        14.99        7,000.00
            sum         sum
           -----------  --------------
                 67.88       19,000.00
(5 rows affected)
```

Calculates the grand totals of the prices and advances of cook books that cost more than $10.

7. **`select type, price, price*2`**

```
from titles
  where type like "%cook"
compute sum(price), sum(price*2)
```

```
type         price
------------ -------------  ------------
mod_cook             19.99         39.98
mod_cook              2.99          5.98
trad_cook            20.95         41.90
trad_cook            11.95         23.90
trad_cook            14.99         29.98
             sum            sum
             =============  ============
                     70.87        141.74
```

Calculates the sum of the price of cook books and the sum of the price used in an expression.

Comments

- The **compute** clause allows you to see the detail and summary rows in one set of results. You can calculate summary values for subgroups, and you can calculate more than one aggregate for the same group.
- **compute** 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**. See example 6.
- If you use **compute by**, you must also use an **order by** clause. The columns listed after **compute 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, if the **order by** clause is:

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

the **compute by** clause can be any (or all) of these:

```
compute by a, b, c
compute by a, b
compute by a
```

Restrictions

- You cannot use a **compute** clause in a cursor declaration.
- Summary values can be computed for both expressions and columns. Any expression or column that appears in the **compute** clause must appear in the **select** list.
- Aliases for column names are not allowed as arguments to the row aggregate in a **compute** clause, although they can be used in the **select** list, the **order by** clause, and the **by** clause of **compute**.
- You cannot use **select into** in the same statement as a **compute** clause, because statements that include **compute** do not generate normal tables.

compute Results Appear As a New Row or Rows

- The aggregate functions ordinarily produce a single value for all the selected rows in the table or for each group, and these summary values are shown as new columns. For example:

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

```
type
-------------  ---------  ----------
mod_cook           22.98  15,000.00
trad_cook          47.89  19,000.00

(2 rows affected)
```

- The **compute** clause makes it possible to retrieve detail and summary rows with one command. For example:

```
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)
```

- Table 1-3 lists the output and grouping of different types of **compute** clauses.

Table 1-3: compute by clauses and detail rows

Clauses and Grouping	Output	Examples
One **compute** clause, same function	One detail row	1, 2, 4, 6, 7
One **compute** clause, different functions	One detail row per type of function	3
More than one **compute** clause, same grouping columns	One detail row per **compute** clause; detail rows together in the output	Same results as having one compute clause with different functions
More than one **compute** clause, different grouping columns	One detail row per **compute** clause; detail rows in different places, depending on the grouping	5

Case Sensitivity

- If your server has a case-insensitive sort order installed, **compute** ignores the case of the data in the columns you specify. For example, given this data:

```
select * from groupdemo
```

```
lname      amount
---------- ------------------
Smith                 10.00
smith                  5.00
SMITH                  7.00
Levi                   9.00
Lévi                  20.00
```

compute by on *lname* produces these results:

```
 select lname, amount from groupdemo
 order by lname
 compute sum(amount) by lname
```

```
 lname      amount
 ---------- ------------------------
 Levi                           9.00

 Compute Result:
 ------------------------
                     9.00

 lname      amount
 ---------- ------------------------
 Lévi                          20.00

Compute Result:
 ------------------------
                    20.00

 lname      amount
 ---------- ------------------------
 smith                          5.00
 SMITH                          7.00
 Smith                         10.00

Compute Result:
 ------------------------
                    22.00
```

The same query on a case- and accent-insensitive server produces these results:

```
lname      amount
---------- ------------------------
Levi                           9.00
Lévi                          20.00

Compute Result:
------------------------
                   29.00

lname      amount
---------- ------------------------
smith                          5.00
SMITH                          7.00
Smith                         10.00

Compute Result:
------------------------
                   22.00
```

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

See Also

Commands	**group by and having Clauses**, **select**
Functions	**avg**, **count**, **max**, **min**, **sum**

connect to...disconnect

(Component Integration Services Only)

Function

Connects to the specified server and disconnects the connected server.

Syntax

```
connect to server_name
   disconnect
```

Keywords and Options

server_name – is the server to which a passthrough connection is required.

Examples

1. **`conect to SYBASE`**

   Establishes a passthrough connection to the server named SYBASE.

2. **`disconnect`**

   Disconnects the connected server.

Comments

- **connect to** specifies the server to which a passthrough connection is required. Passthrough mode enables you to perform native operations on a remote server.
- *server_name* must be the name of a server in the *sysservers* table, with its server class and network name defined.
- When establishing a connection to *server_name* on behalf of the user, Component Integration Services uses one of the following identifiers:
  - A remote login alias described in *sysattributes*, if present
  - The user's name and password

  In either case, if the connection cannot be made to the specified server, Adaptive Server returns an error message.
- For more information about adding remote servers, see **sp_addserver**.

- After making a passthrough connection, Component Integration Services bypasses the Transact-SQL parser and compiler when subsequent language text is received. It passes statements directly to the specified server, and converts the results into a form that can be recognized by the Open Client interface and returned to the client program.
- To close the connection created by the **connect to** command, use the **disconnect** command. You can use this command only after the connection has been made using **connect to**.
- The **disconnect** command can be abbreviated to **disc**.
- The **disconnect** command returns an error unless **connect to** has been previously issued and the server is connected to a remote server.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Permission to use the **connect to** command must be explicitly granted by the System Administrator. The syntax is:

```
grant connect to user_name
```

The System Administrator can grant or revoke connect permission to *public* globally while in the *master* database. If the System Administrator wants to grant or revoke **connect to** permission for a particular user, the user must be a valid user of the *master* database, and the System Administrator must first revoke permission from *public* as follows:

```
use master
go
revoke connect from public
go
sp_adduser fred
go
grant connect to fred
go
```

See Also

Commands	create existing table, grant

System Procedures	**sp_addserver**, **sp_autoconnect**, **sp_helpserver**, **sp_passthru**, **sp_remotesql**, **sp_serveroption**

continue

Function

Restarts the **while** loop. **continue** is often activated by an **if** test.

Syntax

```
while boolean_expression
        statement
    break
        statement
    continue
```

Examples

1.
```
while (select avg(price) from titles) < $30
begin
  update titles
  set price = price * 2
  select max(price) from titles

  if (select max(price) from titles) > $50
      break
  else
      continue
end

begin
print "Too much for the market to bear"
end
```

If the average price is less than $30, double the prices. Then, select the maximum price. If it is less than or equal to $50, restart the **while** loop and double the prices again. If the maximum price is more than $50, exit the **while** loop and print a message.

Comments

- **continue** restarts the **while** loop, skipping any statements after **continue**.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

continue permission defaults to all users. No permission is required to use it.

See Also

Commands	**break**, **while**

create database

Function

Creates a new database. Use **create database** from the *master* database.

Syntax

```
create database database_name
   [on {default | database_device} [= size]
       [, database_device [= size]]...]
   [log on database_device [= size]
       [, database_device [= size]]...]
   [with override]
   [for load]
```

Keywords and Options

database_name – is the name of the new database. It must conform to the rules for identifiers and cannot be a variable.

on – indicates a location and size for the database.

default – indicates that **create database** can put the new database on any default database device(s), as shown in *sysdevices.status*. To specify a size for the database without specifying a location, use this command:

```
on default = size
```

To change a database device's status to "default," use the system procedure **sp_diskdefault**.

database_device – is the logical name of the device on which to locate the database. A database can occupy different amounts of space on each of several database devices. Add database devices to Adaptive Server with **disk init**.

size – is the amount of space (in megabytes) allocated to the database. The Adaptive Server-supplied default size is 2MB.

log on – specifies the logical name of the device for the database logs. You can specify more than one device in the **log on** clause.

with override – forces Adaptive Server to accept your device specifications, even if they mix data and transaction logs on the same device, thereby endangering up-to-the-minute recoverability for your database. If you attempt to mix log and data on the same device without using this clause, the **create database** command fails. If you mix log and data, and use **with override**, you are warned, but the command succeeds.

for load – invokes a streamlined version of **create database** that can be used only for loading a database dump. See "Using the for load Option", below, for more information.

Examples

1. `create database pubs`

   Creates a database named *pubs*.

2.
   ```
   create database pubs
   on default = 4
   ```

   Creates a 4MB database named *pubs*.

3.
   ```
   create database pubs
   on datadev = 3, moredatadev = 2
   ```

   Creates a database named *pubs* with 3MB on the *datadev segment* and 2 MB on the *moredatadev* segment.

4.
   ```
   create database pubs
   on datadev = 3
   log on logdev = 1
   ```

   Creates a database named *pubs* with 3MB of data on the *datadev* segment and a 1MB log on the *logdev* segment.

Comments

- If you do not specify a location and size for a database, the default location is any default database device(s) indicated in *master..sysdevices*. The default size is the larger of the size of the *model* database or the **default database size** parameter in *sysconfigures*.

  System Administrators can increase the default size by using **sp_configure** to change the value of **default database size** and restarting Adaptive Server. The **default database size** parameter must be at least as large as the *model* database. If you increase the size of the *model* database, the default size must also be increased.

  If Adaptive Server cannot give you as much space as you want where you have requested it, it comes as close as possible, on a per-device basis, and prints a message telling how much space was allocated and where it was allocated. The maximum size of a database is system-dependent.

Restrictions

- Adaptive Server can manage up to 32,767 databases.
- Adaptive Server can only create one database at a time. If two database creation requests collide, one user will get this message:

  ```
  model database in use: cannot create new database
  ```
- The maximum number of device fragments for a database is 128. Each time you allocate space on a database device with **create database** or **alter database**, that allocation represents a device fragment, and the allocation is entered as a row in *sysusages*.
- The maximum number of named segments for a database is 32. Segments are named subsets of database devices available to a particular Adaptive Server. For more information on segments, see Chapter 17, "Creating and Using Segments," in the *System Administration Guide*.

New Databases Are Created from *model*

- Adaptive Server creates a new database by copying the *model* database.
- You can customize *model* by adding tables, stored procedures, user-defined datatypes, and other objects, and by changing database option settings. New databases inherit these objects and settings from *model*.
- To guarantee recoverability, the **create database** command must clear every page that was not initialized when the *model* database was copied. This may take several minutes, depending on the size of the database and the speed of your system.

  If you are creating a database in order to load a database dump into it, you can use the **for load** option to skip the page-clearing step. This makes database creation considerably faster.

Ensuring Database Recoverability

- Back up the *master* database each time you create a new database. This makes recovery easier and safer in case *master* is damaged.

➤ ***Note***

If you create a database and fail to back up *master*, you may be able to recover the changes with **disk refit**.

- The **with override** clause allows you to mix log and data segments on a single device. However, for full recoverability, the device or devices specified in **log on** should be different from the physical device that stores the data. In the event of a hard disk crash, the database can be recovered from database dumps and transaction logs.

  A small database can be created on a single device that is used to store both the transaction log and the data, but you **must** rely on the **dump database** command for backups.

- 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 device 10–25 percent of the space you allocate to the database itself. It is best to start small, since space allocated to a transaction log device cannot be reclaimed and cannot be used for storing data.

Using the *for load* Option

You can use the **for load** option for recovering from media failure or for moving a database from one machine to another, if you have not added to the database with **sp_addsegment**. Use **alter database for load** to create a new database in the image of the database from which the database dump to be loaded was made. See Chapter 21, "Backing Up and Restoring User Databases," in the *System Administration Guide* for a discussion of duplicating space allocation when loading a dump into a new database.

- When you create a database using the **for load** option, you can run only the following commands in the new database before loading a database dump:
  - **alter database for load**
  - **drop database**
  - **load database**

  After you load the database dump into the new database, you can also use some **dbcc** diagnostic commands in the databases. After you issue the **online database** command, there are no restrictions on the commands you can use.

- A database created with the **for load** option has a status of "don't recover" in the output from **sp_helpdb**.

Getting Information About Databases

- To get a report on a database, execute the system procedure **sp_helpdb**.
- For a report on the space used in a database, use **sp_spaceused**.

Standards and Compliance

Standard	**Compliance Level**
SQL92	Transact-SQL extension

Permissions

create database permission defaults to System Administrators, who can transfer it to users listed in the *sysusers* table of the master database. However, **create database** permission is often centralized in order to maintain control over database storage allocation.

If you are creating the *sybsecurity* database, you must be a System Security Officer.

create database permission is not included in the **grant all** command.

See Also

Commands	**alter database**, **disk init**, **drop database**, **dump database**, **load database**, **online database**
System procedures	**sp_changedbowner**, **sp_diskdefault**, **sp_helpdb**, **sp_logdevice**, **sp_renamedb**, **sp_spaceused**

create default

Function

Specifies a value to insert in a column (or in all columns of a user-defined datatype) if no value is explicitly supplied at insert time.

Syntax

```
create default [owner.]default_name
   as constant_expression
```

Keywords and Options

default_name – is the name of the default. It must conform to the rules for identifiers and cannot be a variable. Specify the owner's name to create another default of the same name owned by a different user in the current database. The default value for *owner* is the current user.

constant_expression – is an expression that does not include the names of any columns or other database objects. You can include built-in functions that do not reference database objects. Enclose character and date constants in quotes and use a "0x" prefix for binary constants.

Examples

1. **create default phonedflt as "UNKNOWN"**

   Defines a default value. Now, you need to bind it to the appropriate column or user-defined datatype with **sp_bindefault**.

2. **sp_bindefault phonedflt, "authors.phone"**

   The default takes effect only if there is no entry in the *phone* column of the *authors* table. No entry is different from a null value entry. To get the default, issue an **insert** command with a column list that does not include the column that has the default.

3. **create default todays_date as getdate()**

   Creates a default value, *todays_date*, that inserts the current date into the columns to which it is bound.

Comments

- Bind a default to a column or user-defined datatype—but not a Adaptive Server-supplied datatype—with **sp_bindefault**.

- You can bind a new default to a datatype without unbinding the old one. The new default overrides and unbinds the old one.
- To hide the source test of a default, use **sp_hidetext**.

Restrictions

- You can create a default only in the current database.
- **create default** statements cannot be combined with other statements in a single batch.
- You must drop a default with **drop default** before you create a new one of the same name, and you must unbind a default (with the system procedure **sp_unbindefault**) before you drop it.

Datatype Compatibility

- Adaptive Server generates an error message when it tries to insert a default value that is not compatible with the column's datatype. For example, if you bind a character expression such as "N / A" to an *integer* column, any **insert** that does not specify the column value fails.
- If a default value is too long for a character column, Adaptive Server either truncates the string or generates an exception, depending on the setting of the **string_rtruncation** option. For more information, see the **set** command.

Getting Information About Defaults

- Default definitions are stored in *syscomments*.
- After a default is bound to a column, its object ID is stored in *syscolumns*. After a default is bound to a user-defined datatype, its object ID is stored in *systypes*.
- To rename a default, use **sp_rename**.
- For a report on the text of a default, use **sp_helptext**.

Defaults and Rules

- If a column has both a default and a rule associated with it, the default value must not violate the rule. A default that conflicts with a rule cannot be inserted. Adaptive Server generates an error message each time it attempts to insert such a default.

Defaults and Nulls

- If a column does not allow nulls, and you do not create a default for the column, when a user attempts to insert a row but does not include a value for that column, the insert fails and Adaptive Server generates an error message.

  Table 1-4 illustrates the relationship between the existence of a default and the definition of a column as NULL or NOT NULL.

Table 1-4: Relationship between nulls and column defaults

Column Null Type	No Entry, No Default	No Entry, Default Exists	Entry Is Null, No Default	Entry Is Null, Default Exists
NULL	Null inserted	Default value inserted	Null inserted	Null inserted
NOT NULL	Error, command fails	Default value inserted	Error, command fails	Error, command fails

Specifying a Default Value in *create table*

- You can define column defaults using the **default** clause of the **create table** statement as an alternative to using **create default**. However, these column defaults are specific to that table; you cannot bind them to other tables. See **create table** and **alter table** for information about integrity constraints.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Transact-SQL extension	Use the **default** clause of the **create table** statement to create defaults that are SQL92-compliant.

Permissions

create default permission defaults to the Database Owner, who can transfer it to other users.

See Also

Commands	**alter table**, **create rule**, **create table**, **drop default**, **drop rule**
System procedures	**sp_bindefault**, **sp_help**, **sp_helptext**, **sp_rename**, **sp_unbindefault**

create existing table

(Component Integration Services only)

Function

Creates a proxy table, then retrieves and stores metadata from a remote table and places the data into the proxy table.

Syntax

```
create existing table table_name (column_list)
   [ on segment_name ]
```

Keywords and Options

table_name – specifies the name of the table for which you want to create a proxy table.

column_list – specifies the name of the column list that stores information about the remote table.

on *segment_name* – specifies the segment that contains the remote table.

Examples

1.
```
create existing table authors
(
au_id        id,
au_lname     varchar(40)    NOT NULL,
au_fname     varchar(20)    NOT NULL,
phone        char(12),
address      varchar(40)    NULL,
city         varchar(20)    NULL,
state        char(2)        NULL,
zip          char(5)        NULL,
contract     bit
)
```

Creates the proxy table *authors*.

2.
```
create existing table syb_columns
(
id          int,
number      smallint,
colid       tinyint,
status      tinyint,
type        tinyint,
length      tinyint,
offset      smallint,
usertype    smallint,
cdefault    int,
domain      int,
name        varchar(30),
printfmt    varchar(255)    NULL,
prec        tinyint         NULL,
scale       tinyint         NULL
)
```

Creates the proxy table *syb_columns*.

Comments

- The **create existing table** command does not create a new table. Instead, Component Integration Services checks the table mapping to confirm that the information in *column_list* matches the remote table, verifies the existence of the underlying object, and retrieves and stores meta data about the remote table.
- If the host data file or remote server object does not exist, the command is rejected with an error message.
- If the object exists, the system tables *sysobjects*, *syscolumns*, and *sysindexes* are updated. The verification is a three-step operation:
  - The nature of the existing object is determined. For host data files, this requires determining file organization and record format. For remote server objects, this requires determining whether the object is a table, a view, or an RPC.
  - For remote server objects (other than RPCs), column attributes obtained for the table or view are compared with those defined in the *column_list*.
  - Index information from the host data file or remote server table is extracted and used to create rows for the system table *sysindexes*. This defines indexes and keys in Adaptive Server terms and enables the query optimizer to consider any indexes that might exist on this table.

- The **on *segment_name*** clause is processed locally and is not passed to a remote server.
- After successfully defining an existing table, issue an **update statistics** command for the table. This allows the query optimizer to make intelligent choices regarding index selection and join order.
- Component Integration Services allows you to create a proxy table with a column defined as NOT NULL even though the remote column is defined as NULL. It displays a warning to notify you of the mismatch.

Datatype Conversions

- When using the **create existing table** command, you must specify all datatypes with recognized Adaptive Server datatypes. If the remote server tables reside on a class of server that is heterogeneous, the datatypes of the remote table are automatically converted into the specified Adaptive Server types when the data is retrieved. If the conversion cannot be made, Component Integration Services does not allow the table to be defined.
- The *Component Integration Services User's Guide* contains a section for each supported server class and identifies all possible datatype conversions that are implicitly performed by Component Integration Services.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

create existing table permission defaults to the table owner and is not transferable.

See Also

Commands	**alter table**, **create table**, **drop index**, **insert**, **order by Clause**, **set**, **update**, **update statistics**

create index

Function

Creates an index on one or more columns in a table.

Syntax

```
create [unique] [clustered | nonclustered]
      index index_name
   on [[database.]owner.]table_name (column_name
       [, column_name]...)
   [with {{fillfactor | max_rows_per_page} = x,
       consumers = x, ignore_dup_key, sorted_data,
       [ignore_dup_row | allow_dup_row]}]
   [on segment_name]
```

Keywords and Options

unique – prohibits duplicate index values (also called “key values”). The system checks for duplicate key values when the index is created (if data already exists), and each time data is added with an **insert** or **update**. If there is a duplicate key value or if more than one row contains a null value, the command fails, and Adaptive Server prints an error message giving the duplicate entry.

◆ ***WARNING!***

Adaptive Server does not detect duplicate rows if a table contains any non-null *text* or *image* columns.

update and **insert** commands that generate duplicate key values fail, unless the index was created with **ignore_dup_row** or **ignore_dup_key.**

Composite indexes (indexes in which the key value is composed of more than one column) can also be unique.

The default is nonunique. To create a nonunique clustered index on a table that contains duplicate rows, you must specify **allow_dup_row** or **ignore_dup_row**. See “Duplicate Rows”, below.

clustered – means that the physical order of rows on the current database device is the same as the indexed order of the rows. The bottom, or **leaf level**, of the clustered index contains the actual data pages. A clustered index almost always retrieves data faster than a nonclustered index. Only one clustered index per table is permitted. See "Creating Clustered Indexes", below.

If **clustered** is not specified, **nonclustered** is assumed.

nonclustered – means that 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. You can have up to 249 nonclustered indexes per table.

index_name – is the name of the index. Index names must be unique within a table, but need not be unique within a database.

table_name – is the name of the table in which the indexed column or columns are located. Specify the database name if the table is in another database, and specify the owner's name if more than one table of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

column_name – is the column or columns to which the index applies. Composite indexes are based on the combined values of up to 16 columns. The sum of the maximum lengths of all the columns used in a composite index cannot exceed 600 bytes. List the columns to be included in the composite index (in the order in which they should be sorted) inside the parentheses following *table_name*.

fillfactor – specifies how full Adaptive Server will make each page when it is creating a new index on existing data. The **fillfactor** percentage is relevant only at the time the index is created. As the data changes, the pages are not maintained at any particular level of fullness.

The default for **fillfactor** is 0; this is used when you do not include **with fillfactor** in the **create index** statement (unless the value has been changed with **sp_configure**). When specifying a **fillfactor**, use a value between 1 and 100.

A **fillfactor** of 0 creates clustered indexes with completely full pages and nonclustered indexes with completely full leaf pages. It leaves a comfortable amount of space within the index B-tree in both the clustered and nonclustered indexes. There is seldom a reason to change the **fillfactor.**

If the **fillfactor** is set to 100, Adaptive Server creates both clustered and nonclustered indexes with each page 100 percent full. A

fillfactor of 100 only makes sense for read-only tables—tables to which no additional data will ever be added.

fillfactor values smaller than 100 (except 0, which is a special case) cause Adaptive Server to create new indexes with pages that are not completely full. A **fillfactor** of 10 might be a reasonable choice if you are creating an index on a table that will eventually hold a great deal more data, but small **fillfactor** values cause each index (or index and data) to take more storage space.

◆ ***WARNING!***

Creating a clustered index with a fillfactor affects the amount of storage space your data occupies, since Adaptive Server redistributes the data as it creates the clustered index.

max_rows_per_page – limits the number of rows on data pages and the leaf level pages of indexes. **max_rows_per_page** and **fillfactor** are mutually exclusive. Unlike **fillfactor**, the **max_rows_per_page** value is maintained until it is changed with **sp_chgattribute**.

If you do not specify a value for **max_rows_per_page**, Adaptive Server uses a value of 0 when creating the table. Values for tables and clustered indexes are between 0 and 256. The maximum number of rows per page for nonclustered indexes depends on the size of the index key. Adaptive Server returns an error message if the specified value is too high.

A **max_rows_per_page** value of 0 creates clustered indexes with full pages and nonclustered indexes with full leaf pages. It leaves a comfortable amount of space within the index B-tree in both clustered and nonclustered indexes.

If **max_rows_per_page** is set to 1, Adaptive Server creates both clustered and nonclustered indexes with one row per page at the leaf level. Use low values to reduce lock contention on frequently accessed data. However, low **max_rows_per_page** values cause Adaptive Server to create new indexes with pages that are not completely full, uses more storage space, and may cause more page splits.

If Component Integration Services is enabled, you cannot use **max_rows_per_page** for remote servers.

◆ ***WARNING!***

Creating a clustered index with max_rows_per_page can affect the amount of storage space your data occupies, since Adaptive Server redistributes the data as it creates the clustered index.

ignore_dup_key – cancels attempts of duplicate key entry into a table that has a unique index (clustered or nonclustered). Adaptive Server cancels the attempted **insert** or **update** of a duplicate key with an informational message. After the cancellation, the transaction containing the duplicate keys proceeds to completion.

You cannot create a unique index on a column that includes duplicate values or more than one null value, whether or not **ignore_dup_key** is set. If you attempt to do so, Adaptive Server prints an error message that gives the first of the duplicate values. You must eliminate duplicates before Adaptive Server can create a unique index on the column.

ignore_dup_row – allows you to create a new, nonunique clustered index on a table that includes duplicate rows by deleting the duplicate rows from the table, and cancels any **insert** or **update** that would create a duplicate row, but does not roll back the entire transaction. See "Duplicate Rows", below, for more information.

allow_dup_row – allows you to create a nonunique clustered index on a table that includes duplicate rows, and allows you to duplicate rows with **update** and **insert** statements. See "Duplicate Rows", below, for an explanation of how to use these options.

sorted_data – speeds creation of clustered indexes or unique nonclustered indexes 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). See "Using the sorted_data Option to Speed Sorts" for more information.

on *segment_name* – creates the index on the named segment. Before using the **on** *segment_name* option, initialize the device with **disk init**, and add the segment to the database with the **sp_addsegment** system procedure. See your System Administrator, or use **sp_helpsegment** for a list of the segment names available in your database.

with consumers – specifies the number of consumer processes that should perform the sort operation for creating the index. The actual number of consumer processes used to sort the index may be smaller than the specified number, if fewer worker processes are available when Adaptive Server executes the sort. You cannot use the **consumers** option when creating a clustered index on a partitioned table.

Examples

1.
```
create index au_id_ind
on authors (au_id)
```
Creates an index named *au_id_ind* on the *au_id* column of the *authors* table.

2.
```
create unique clustered index au_id_ind
on authors(au_id)
```
Creates a unique clustered index named *au_id_ind* on the *au_id* column of the *authors* table.

3.
```
create index ind1
on titleauthor (au_id, title_id)
```
Creates an index named *ind1* on the *au_id* and *title_id* columns of the *titleauthor* table.

4.
```
create nonclustered index zip_ind
on authors(postalcode)
with fillfactor = 25, consumers = 4
```
Creates a nonclusters index named *zip_ind* on the *zip* column of the *authors* table, filling each index page one-quarter full and limiting the sort to 4 consumer processes.

Comments

- Run **update statistics** periodically if you add data to the table that changes the distribution of keys in the index. The query optimizer uses the information created by **update statistics** to select the best plan for running queries on the table.
- If the table contains data when you create a nonclustered index, Adaptive Server runs **update statistics** on the new index. If the table contains data when you create a clustered index, Adaptive Server runs **update statistics** on all the table's indexes.
- Index all columns that are regularly used in joins.
- When Component Integration Services is enabled, the **create index** command is reconstructed and passed directly to the Adaptive Server associated with the table.

Restrictions

- You cannot create an index on a column with a datatype of *bit*, *text*, or *image*.
- A table can have a maximum of 249 nonclustered indexes.
- A table can have a maximum of one clustered index.
- You can create an index on a temporary table. It disappears when the table disappears.
- You can create an index on a table in another database, as long as you are the owner of that table.
- You cannot create an index on a view.
- **create index** runs more slowly while a **dump database** is taking place.
- You can create a clustered index on a partitioned table or partition a table with a clustered index if the following conditions are true:
  - The **select into/bulkcopy/pllsort** database option is turned on,
  - Adaptive Server is configured for parallel processing, and
  - There is one more worker process available than the number of partitions.

  For more information about clustered indexes on partitioned tables, see "Improving Insert Performance with Partitions" in Chapter 17, "Controlling Physical Data Placement", in the *Performance and Tuning Guide.*

Creating Indexes Efficiently

- Indexes speed data retrieval, but can slow data updates. For better performance, create a table on one segment and create its nonclustered indexes on another segment, when the segments are on separate physical devices.
- Adaptive Server can create indexes in parallel if a table is partitioned and the server is configured for parallelism. It can also use sort buffers to reduce the amount of I/O required during sorting. See Chapter 15, "Parallel Sorting," in the *Performance and Tuning Guide* for more information.
- Create a clustered index before creating any nonclustered indexes, since nonclustered indexes are automatically rebuilt when a clustered index is created.

Creating Clustered Indexes

- A table "follows" its clustered index. When you create a table and then use the **on** *segment_name* extension to **create clustered index**, the table migrates to the segment where the index is created..

  If you create a table on a specific segment, then create a clustered index without specifying a segment, Adaptive Server moves the table to the default segment when it creates the clustered index there.

  Because text and image data is stored in a separate page chain, creating a clustered index with **on** *segment_name* does not move text and image columns.
- To create a clustered index, Adaptive Server duplicates the existing data; the server deletes the original data when the index is complete. Before creating a clustered index, use **sp_spaceused** to make sure that the database has at least 120 percent of the size of the table available as free space.
- The clustered index is often created on the table's primary key (the column or columns that uniquely identify the row). The primary key can be recorded in the database (for use by front-end programs and the system procedure **sp_depends**) with the system procedure **sp_primarykey**.
- To allow duplicate rows in a clustered index, specify **allow_dup_row**.

Space Requirements for Indexes

- Space is allocated to tables and indexes in increments of one extent, or eight pages, at a time. Each time an extent is filled, another extent is allocated. (Use the system procedure **sp_spaceused** to display the amount of space allocated and used by an index.)
- In some cases, using the **sorted_data** option allows Adaptive Server to skip copying the data rows as described in Table 1-7. In these cases, you need only enough additional space for the index structure itself. Depending on key size, this is usually about 20 percent of the size of the table.

Duplicate Rows

- The **ignore_dup_row** and **allow_dup_row** options are not relevant when you are creating a nonunique, nonclustered index. Because Adaptive Server attaches a unique row identification number internally in each nonclustered index, it never worries about duplicate rows, not even for identical data values.

- **ignore_dup_row** and **allow_dup_row** are mutually exclusive.
- A nonunique clustered index allows duplicate keys, but does not allow duplicate rows unless you specify **allow_dup_row**.
- **allow_dup_row** allows you to create a nonunique, clustered index on a table that includes duplicate rows. If a table has a nonunique, clustered index that was created without the **allow_dup_row** option, you cannot create new duplicate rows using the **insert** or **update** command.

  If any index in the table is unique, the requirement for uniqueness takes precedence over the **allow_dup_row** option. You cannot create an index with **allow_dup_row** if a unique index exists on any column in the table.
- The **ignore_dup_row** option is also used with a nonunique, clustered index. The **ignore_dup_row** option eliminates duplicates from a batch of data. **ignore_dup_row** cancels any **insert** or **update** that would create a duplicate row, but does not roll back the entire transaction.
- Table 1-5 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 attempts to enter duplicate rows into a table.

Table 1-5: Duplicate row options for nonunique clustered indexes

Option Setting	Create an Index on a Table That Has Duplicate Rows	Insert Duplicate Rows into a Table With an Index
Neither option set	**create index** fails.	**insert** fails.
allow_dup_row set	**create index** completes.	**insert** completes.
ignore_dup_row set	Index is created but duplicate rows are deleted; error message.	All rows are inserted except duplicates; error message.

Table 1-6 shows which index options can be used with the different types of indexes:

Table 1-6: Index options

Index Type	Options
Clustered	**ignore_dup_row** | **allow_dup_row**
Unique, clustered	**ignore_dup_key**
Nonclustered	None
Unique, nonclustered	**ignore_dup_key**, **ignore_dup_row**

Getting Information About Tables and Indexes

- Each index—including composite indexes—is represented by one row in *sysindexes*.
- For information about the order of the data retrieved through indexes and the effects of an Adaptive Server's installed sort order, see the **order by** clause.
- For information about a table's indexes, execute the system procedure **sp_helpindex**.

Using Unique Constraints in Place of Indexes

- As an alternative to **create index**, you can implicitly create unique indexes by specifying a unique constraint with the **create table** or **alter table** statement. The unique constraint creates a clustered or nonclustered unique index on the columns of a table. These **implicit** indexes are named after the constraint, and they follow the same rules for indexes created with **create index.**
- You cannot drop indexes supporting unique constraints using the **drop index** statement. They are dropped when the constraints are dropped through an **alter table** statement or when the table is dropped. See **create table** for more information about unique constraints.

Using the *sorted_data* Option to Speed Sorts

- The **sorted_data** option can reduce the time needed to create an index by skipping the sort step and by eliminating the need to copy the data rows to new pages in certain cases. 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, Adaptive Server displays an error message, and the command fails.

Creating a nonunique, nonclustered index succeeds, unless there are rows with duplicate keys. If there are rows with duplicate keys, Adaptive Server displays an error message, and the command fails.

- The effects of **sorted_data** for creating a clustered index depend on whether the table is partitioned and whether certain other options are used in the **create index** command. Some options require data copying, if used at all, for nonpartitioned tables and sorts plus data copying for partitioned tables, while others require data copying only if you use one of the following options:
  - Using the **ignore_dup_row** option
  - Using the **fillfactor** option
  - Using the **on** *segmentname* clause to specify a segment that is different from the segment where the table data is located
  - Using the **max_rows_per_page** clause to specify a value that is different from the value associated with the table
- Table 1-7 shows when the sort is required and when the table is copied for partitioned and nonpartitioned tables.

Table 1-7: Using the sorted_data option for creating a clustered index

Options	**Partitioned Table**	**Unpartitioned table**
No options specified	Parallel sort; copies data, distributing evenly on partitions; creates index tree.	Either parallel or nonparallel sort; copies data, creates index tree.
with sorted_data only or **with sorted_data on** *same_segment*	Creates index tree only. Does not perform the sort or copy data. Does not run in parallel.	Creates index tree only. Does not perform the sort or copy data. Does not run in parallel.
with sorted_data and **ignore_dup_row** or **fillfactor** or **on** *other_segment* or **max_rows_per_page**	Parallel sort; copies data, distributing evenly on partitions; creates index tree.	Copies data and creates the index tree. Does not perform the sort. Does not run in parallel.

Standards and Compliance

Standard	**Compliance Level**
SQL92	Transact-SQL extension

Permissions

create index permission defaults to the table owner and is not transferable.

See Also

Commands	**alter table**, **create table**, **drop index**, **insert**, **order by Clause**, **set**, **update**, **update statistics**
System procedures	**sp_addsegment**, **sp_helpindex**, **sp_helpsegment**, **sp_spaceused**

create procedure

Function

Creates a stored procedure or an extended stored procedure (ESP) that can take one or more user-supplied parameters.

Syntax

```
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}
```

Keywords and Options

procedure_name – is the name of the procedure. It must conform to the rules for identifiers and cannot be a variable. Specify the owner's name to create another procedure of the same name owned by a different user in the current database. The default value for *owner* is the current user.

;number – is an optional integer used 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, if the procedures used with the application named **orders** are named *orderproc;1*, *orderproc;2*, and so on, the statement:

```
drop proc orderproc
```

drops the entire group.

Once procedures have been grouped, individual procedures within the group cannot be dropped. For example, the statement:

```
drop procedure orderproc;2
```

is not allowed.

You cannot group procedures if you are running Adaptive Server in the **evaluated configuration**. The evaluated configuration requires that you disallow procedure grouping so 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 configuration parameter **allow procedure grouping** with the system procedure **sp_configure**. For more information about the evaluated configuration, see Chapter 1, "Overview of Security Features," in the *Security Administration Guide*.

parameter_name – is the name of an argument to the procedure. The value of each parameter is supplied when the procedure is executed. (Parameter names are optional in **create procedure** statements—a procedure need not take any arguments.)

Parameter names must be preceded by the @ sign and conform to the rules for identifiers. A parameter name, including the @ sign, can be a maximum of 30 characters. Parameters are local to the procedure: the same parameter names can be used in other procedures.

If the value of a parameter contains non-alphanumeric characters, it must be enclosed in quotes. This includes object names qualified by a database name or owner name, since they include a period. If the value of a character parameter begins with a numeric character, it also must be enclosed in quotes.

datatype [(*length*) | (*precision* [, *scale*])] – is the datatype of the parameter. See Chapter 7, "System and User-Defined Datatypes," for more information about datatypes. Stored procedure parameters cannot have a datatype of *text* or *image* or a user-defined datatype whose underlying type is *text* or *image*.

The *char, varchar, nchar, nvarchar, binary*, and *varbinary* datatypes should include a *length* in parentheses. If you omit the length, Adaptive Server truncates the parameter value to 1 character.

The *float* datatype expects a binary *precision* in parentheses. If you omit the precision, Adaptive Server uses the default precision for your platform.

The *numeric* and *decimal* datatypes expect a *precision* and *scale*, enclosed in parentheses and separated by a comma. If you omit the precision and scale, Adaptive Server uses a default precision of 18 and a scale of 0.

default – defines a default value for the procedure's parameter. If a default is defined, a user can execute the procedure without giving a parameter value. The default must be a constant. It can include the wildcard characters (%, _, [ ], and [^]) if the procedure uses the parameter name with the keyword **like** (see example 2).

The default can be NULL. The procedure definition can specify that some action be taken if the parameter value is NULL (see example 3).

output – indicates that the parameter is a return parameter. Its value can be returned to the **execute** command that called this procedure. Use return parameters to return information to the calling procedure (see example 5).

To return a parameter value through several levels of nested procedures, each procedure must include the **output** option with the parameter name, including the **execute** command that calls the highest level procedure.

The **output** keyword can be abbreviated to **out**.

with recompile – means that Adaptive Server never saves a plan for this procedure; a new plan is created each time it is executed. Use this optional clause when you expect that the execution of a procedure will be atypical—that is, when you need a new plan. The **with recompile** clause has no impact on the execution of an extended stored procedure.

SQL_statements – specify the actions the procedure is to take. Any number and kind of SQL statements can be included, with the exception of **create view**, **create default**, **create rule**, **create procedure**, **create trigger**, and **use**.

create procedure SQL statements often include control-of-flow language, including one or more of the following: **declare**; **if...else**; **while**; **break**; **continue**; **begin...end**; **goto** label; **return**; **waitfor**; **/*** *comment* ***/**. They can also refer to parameters defined for the procedure.

The SQL statements can reference objects in another database, as long as they are properly qualified.

- **external name** – creates an extended stored procedure. If the **as external name** syntax is used, you cannot use the *number* parameter with **as external name**.

dll_name – specifies the name of the dynamic link library (DLL) or shared library containing the functions that implement the extended stored procedure. The *dll_name* can be specified with no extension or with a platform-specific extension, such as *.dll* on Windows NT or *.so* on Sun Solaris. If you specify the extension, enclose the entire *dll_name* in quotation marks.

Examples

1.
```
create procedure showind @tabname varchar(30)
as
  select sysobjects.name, sysindexes.name, indid
  from sysindexes, sysobjects
  where sysobjects.name = @tabname
  and sysobjects.id = sysindexes.id
```

   Given a table name, the procedure *showind* displays its name and the names and identification numbers of any indexes on any of its columns.

   Here are the acceptable syntax forms for executing *showind*:

   execute showind titles

   execute showind @tabname = "titles"

   Or, if this is the first statement in a file or batch:

   showind titles

2.
```
create procedure
showsysind @table varchar(30) = "sys%"
as
  select sysobjects.name, sysindexes.name, indid
  from sysindexes, sysobjects
  where sysobjects.name like @table
  and sysobjects.id = sysindexes.id
```

   This procedure displays information about the system tables if the user does not supply a parameter.

3.
```
create procedure
showindnew @table varchar(30) = null
as
  if @table is null
   print "Please give a table name"
  else
   select sysobjects.name, sysindexes.name, indid
   from sysindexes, sysobjects
   where sysobjects.name = @table
   and sysobjects.id = sysindexes.id
```

   This procedure specifies an action to be taken if the parameter is NULL (that is, if the user does not give a parameter).

4. **create procedure mathtutor @mult1 int, @mult2 int,**

```
create procedure mathtutor @mult1 int, @mult2 int,
  @result int output
as
select @result = @mult1 * @mult2
```

This procedure multiplies two integer parameters and returns the product in the **output** parameter, *@result*. If the procedure is executed by passing it 3 integers, the **select** statement performs the multiplication and assigns the values, but does not print the return parameter:

```
mathtutor 5, 6, 32
```

```
(return status 0)
```

5. **declare @guess int**

```
declare @guess int
select @guess = 32
exec mathtutor 5, 6, @result = @guess output
```

```
(1 row affected)
(return status = 0)

Return parameters:

@result
-----------
         30
```

In this example, both the procedure and the **execute** statement include **output** with a parameter name so that the procedure can return a value to the caller. The output parameter and any subsequent parameters in the **execute** statement, *@result*, **must** be passed as:

@parameter = value

- The value of the return parameter is always reported, whether or not its value has changed.
- *@result* does not need to be declared in the calling batch because 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*).

6.
```
declare @guess int
declare @store int
select @guess = 32
select @store = @guess
execute mathtutor 5, 6, @result = @guess output
select Your_answer = @store, Right_answer = @guess
if @guess = @store
    print "Right-o"
else
    print "Wrong, wrong, wrong!"
```

```
(1 row affected)
(1 row affected)
(return status = 0)

Return parameters:

@result
-----------
         30

Your_answer Right_answer
----------- ------------
         32           30

(1 row affected)
Wrong, wrong, wrong!
```

Return parameters can be used in additional SQL statements in the batch or calling procedure. This example shows how to use the value of *@guess* in conditional clauses after the **execute** statement by storing it in another variable name, *@store*, during the procedure call. When return parameters are used in an **execute** statement that is part of a SQL batch, the return values are printed with a heading before subsequent statements in the batch are executed.

7.
```
create procedure xp_echo @in varchar(255),
    @out varchar(255) output
as external name "sqlsrvdll.dll"
```

Creates an extended stored procedure named *xp_echo*, which takes an input parameter, *@in*, and echoes it to an output parameter, *@out*. The code for the procedure is in a function named **xp_echo**, which is compiled and linked into a DLL named *sqlsrvdll.dll*.

Comments

- Once a procedure is created, you can run it by issuing the **execute** command along with the procedure's name and any parameters. If a procedure is the first statement in a batch, you can give its name without the keyword **execute**.
- You can hide the source text for a procedure with **sp_hidetext**.
- When a stored procedure batch executes successfully, Adaptive Server sets the *@@error* global variable to 0.

Restrictions

- The maximum number of parameters that a stored procedure can have is 255.
- The maximum number of local and global variables in a procedure is limited only by available memory.
- The maximum amount of text in a stored procedure is 16MB.
- A **create procedure** statement cannot be combined with other statements in a single batch.
- You can create a stored procedure only in the current database, although the procedure can reference objects from other databases. Any objects referenced in a procedure must exist at the time you create the procedure. You can create an object within a procedure and then reference it, as long as the object is created before it is referenced.

  Similarly, you cannot use **alter table** in a procedure to add a column and then refer to that column within the procedure.
- 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 you may have added to the table. You must **drop** the procedure and re-create it.
- Within a stored procedure, you cannot create an object (including a temporary table), 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.

◆ ***WARNING!***

Certain changes to databases, such as dropping and re-creating indexes, can cause object IDs to change. When object IDs change, stored procedures recompile automatically, and can increase slightly in size. Leave some space for this increase.

Extended Stored Procedures

- If the **as** *external name* syntax is used, **create procedure** registers an extended stored procedure (ESP). Extended stored procedures execute procedural language functions rather than Transact-SQL commands.
- On Windows NT, an ESP function should not call a C run-time signal routine. This can cause XP Server to fail, because Open Server™ does not support signal handling on Windows NT.
- To support multi-threading, ESP functions should use the Open Server **srv_yield** function, which suspends and reschedules the XP Server thread to allow another thread of the same or higher priority to execute.
- The DLL search mechanism is platform-dependent. On Windows NT, the sequence of a DLL file name search is as follows:

  a. The directory from which the application is loaded

  b. The current directory

  c. The system directory (SYSTEM32)

  d. Directories listed in the PATH environment variable

  If the DLL is not in the first three directories, set the PATH to include the directory in which it is located.

  On UNIX platforms, the search method varies with the particular platfrom. If it fails to find the DLL or shared library, it searches *$SYBASE/lib*.

  Absolute path names are not supported.

System Procedures

- System Administrators can create new system procedures in the *sybsystemprocs* database. System procedure names must begin with the characters “sp_”. These procedures can be executed from any database by specifying the procedure name; it is not necessary to qualify it with the *sybsystemprocs* database name. For more information about creating system procedures, see "Creating System Procedures" in Chapter 1, "Overview of System Administration," of the *System Administration Guide*.

- System procedure results may vary depending on the context in which they are executed. For example, the system procedure *sp_foo*, which executes the **db_name()** system function, returns the name of the database from which it is executed. When executed from the *pubs2* database, it returns the value "pubs2":

```
use pubs2

sp_foo

------------------------------
pubs2
```

When executed from *sybsystemprocs*, it returns the value "sybsystemprocs":

```
use sybsystemprocs
sp_foo

------------------------------
sybsystemprocs
```

Procedure Return Status

- Stored procedures can return an integer value called a **return status**. The return status either indicates that the procedure executed successfully or specifies the type of error that occurred.
- When you execute a stored procedure, it automatically returns the appropriate status code. Adaptive Server currently returns the following status codes:

Code	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

Code	Meaning
-13	Database is corrupt
-14	Hardware error

Codes -15 through -99 are reserved for future use.

- Users can generate a user-defined return status with the **return** statement. The status can be any integer other than 0 through -99. The following example returns "1" when a book has a valid contract and "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
```

```
checkcontract @titleid = "BU1111"
```

```
(return status = 1)
```

```
checkcontract @titleid = "MC3026"
```

```
(return status = 2)
```

- If more than one error occurs during execution, the code with the highest absolute value is returned. User-defined return values take precedence over system-defined values.

Object Identifiers

- To change the name of a stored procedure, use **sp_rename**. To change the name of an extended stored procedure, drop the procedure, rename and recompile the supporting function, and then recreate the procedure.
- If a procedure references table names, column names, or view names that are not valid identifiers, you must **set quoted_identifier on** before the **create procedure** command and enclose each such name in double quotes. The **quoted_identifier** option does **not** need to be on when you execute the procedure.
- You must drop and re-create the procedure if any of the objects it references have been renamed.
- Inside a stored procedure, object names used with the **create table** and **dbcc** commands must be qualified with the object owner's name if other users are to make use of the stored procedure. For example, user "mary," who owns table *marytab*, should qualify

the name of her table inside a stored procedure (when it is used with these commands) if she wants other users to be able to execute it. This is because the object names are resolved when the procedure is run. When another user tries to execute the procedure, Adaptive Server looks for a table called *marytab* owned by the user "mary" and not a table called *marytab* owned by the user executing the stored procedure.

Object names used with other statements (for example, **select** or **insert**) inside a stored procedure need not be qualified because the names are resolved when the procedure is compiled.

Temporary Tables and Procedures

- You can create a procedure to reference a temporary table if the temporary table is created in the current session. A temporary table created within a procedure disappears when the procedure exits. See "Manipulating Temporary Tables in Stored Procedures" in Chapter 7, "Creating Databases and Tables," in the *Transact-SQL User's Guide* for more information.
- System procedures such as **sp_help** work on temporary tables, but only if you use them from *tempdb*.

Setting Options in Procedures

- You can use the **set** command inside a stored procedure. Most **set** options remain in effect during the execution of the procedure and then revert to their former settings.

  However, if you use a **set** option (such as **identity_insert**) which requires the user to be the object owner, a user who is not the object owner cannot execute the stored procedure.

Getting Information About Procedures

- For a report on the objects referenced by a procedure, use **sp_depends**.
- To display the text of a **create procedure** statement, which is stored in *syscomments*, use the system procedure **sp_helptext** with the procedure name as the parameter. You must be using the database where the procedure resides when you use **sp_helptext.** To display the text of a system procedure, execute **sp_helptext** from the *sybsystemprocs* database.

- To see a list of system extended stored procedures and their supporting DLLs, use **sp_helpextendedproc** from the *sybsystemprocs* database.

Nested Procedures

- Procedure nesting occurs when one stored procedure calls another.
- If you execute a procedure that calls another procedure, the called procedure can access objects created by the calling procedure.
- The nesting level increments when the called procedure begins execution and decrements when the called procedure completes execution. Exceeding the maximum of 16 levels of nesting causes the transaction to fail.
- You can call another procedure by name or by a variable name in place of the actual procedure name.
- The current nesting level is stored in the *@@nestlevel* global variable.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

create procedure permission defaults to the Database Owner, who can transfer it to other users.

Permission to use a procedure must be granted explicitly with the **grant** command and may be revoked with the **revoke** command.

Permissions on Objects: Procedure Creation Time

When you create a procedure, Adaptive Server makes no permission checks on objects, such as tables and views, that are referenced by the procedure. Therefore, you can create a procedure successfully even though you do not have access to its objects. All permission checks occur when a user executes the procedure.

Permissions on Objects: Procedure Execution Time

When the procedure is executed, permission checks on objects depend upon whether the procedure and all referenced objects are owned by the same user.

- If the procedure's objects are owned by differerent users, the invoker must have been granted direct access to the objects. For example, if the procedure performs a select from a table that the user cannot access, the procedure execution fails.
- If a procedure and its objects are owned by the same user, however, special rules apply. The invoker automatically has "implicit permission" to access the procedure's objects even though the invoker could not access them directly. Without having to grant users direct access to your tables and views, you can give them restricted access with a stored procedure. In this way, a stored procedure can be a security mechanism. For example, invokers of the procedure might be able to access only certain rows and columns of your table.

  A detailed description of the rules for implicit permissions is discussed in Chapter 5, "Managing User Permissions," in the *Security Administration Guide*.

See Also

Commands	**begin...end**, **break**, **continue**, **declare**, **drop procedure**, **execute**, **goto Label**, **grant**, **if...else**, **return**, **select**, **waitfor**, **while**
System procedures	**sp_addextendedproc**, **sp_helpextendedproc**, **sp_helptext**, **sp_hidetext**, **sp_rename**

create role

Function

Creates a user-defined role.

Syntax

```
create role role_name [with passwd "password"]
```

Keywords and Options

role_name – is the name of the role. It must be unique to the server and conform to the rules for identifiers. It cannot be a variable.

with passwd *password* – attaches a password the user must enter to activate the role.

Examples

1. **`create role doctor_role`**

   Creates a role named doctor_role.

2. **`create role doctor_role with passwd "physician"`**

   Creates a role named doctor_role with the password physician.

Comments

- The **create role** command creates a role with privileges, permissions, and limitations that you design. For more information on how to use **create role**, see Chapter 4, "Administering Roles," in the *Security Administration Guide*.

  For more information on monitoring and limiting access to objects, see the **set role** command.
- Use **create role** from the *master* database.
- Use the **with passwd** *password* clause to attach a password to a role at creation. If you attach a password to the role, the user granted this role must specify the password to activate the role.
- Role names must be unique to the server.

Restrictions

- You can create up to 1024 user-defined roles.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

You must be a System Security Officer to use **create role**.

create role permission is not included in the **grant all** command.

See Also

Commands	**alter role**, **drop role**, **grant**, **revoke**, **set**
System procedures	**sp_activeroles**, **sp_displaylogin**, **sp_displayroles**, **sp_helprotect**, **sp_modifylogin**

create rule

Function

Specifies the domain of acceptable values for a particular column or for any column of a user-defined datatype.

Syntax

```
create rule [owner.]rule_name
   as condition_expression
```

Keywords and Options

rule_name – is the name of the new rule. It must conform to the rules for identifiers and cannot be a variable. Specify the owner's name to create another rule of the same name owned by a different user in the current database. The default value for *owner* is the current user.

condition_expression – specifies the conditions that define the rule. It can be any expression that is valid in a **where** clause, and can include arithmetic operators, relational operators, **in**, **like**, **between**, and so on. However, it cannot reference a column or any other database object. Built-in functions that do not reference database objects **can** be included.

A *condition_expression* takes one argument. The argument is prefixed by the @ sign and refers to the value that is entered via the **update** or **insert** command. You can use any name or symbol to represent the value when you write the rule, but the first character must be the @ sign. Enclose character and date constants in quotes, and precede binary constants with "0x".

Examples

1.
   ```
   create rule limit
   as @advance < $1000
   ```

   Creates a rule named *limit* which limits the value of *advance* to less than $1000.

2.
   ```
   create rule pubid_rule
   as @pub_id in ('1389', '0736', '0877')
   ```

   Creates a rule named *pubid_rule* which restricts the values of *pub_id* to 1389, 0736, or 0877.

3. **`create rule picture`**
   `as @value like '_-%[0-9]'`

   Creates a rule named *picture* which restricts the value of *value* to always begin with the indicated characters.

Comments

- To hide the text of a rule, use **sp_hidetext**.
- To rename a rule, use **sp_rename**.

Restrictions

- You can create a rule only in the current database.
- Rules do not apply to the data that already exists in the database at the time the rules are created.
- **create rule** statements cannot be combined with other statements in a single batch.
- You cannot bind a rule to a Adaptive Server-supplied datatype or to a column of type *text*, *image*, or *timestamp*.
- You must drop a rule before you create a new one of the same name, and you must unbind a rule before you drop it. Use:

  `sp_unbindrule` ***`objname`*** **`[, futureonly]`**

Binding Rules

- Use the system procedure **sp_bindrule** to bind a rule to a column or user-defined datatype. Its syntax is:

  `sp_bindrule` ***`rulename`*****`,`** ***`objname`*** **`[, futureonly]`**
- A rule that is bound to a user-defined datatype is activated when you insert a value into, or update, a column of that type. Rules do **not** test values inserted into variables of that type.
- The rule must be compatible with the datatype of the column. For example, you cannot use:

  ```
  @value like A%
  ```

  as a rule for an exact or approximate numeric column. If the rule is not compatible with the column to which it is bound, Adaptive Server generates an error message when it tries to insert a value, not when you bind it.
- You can bind a rule to a column or datatype without unbinding an existing rule.

- Rules bound to columns always take precedence over rules bound to user-defined datatypes, regardless of which rule was most recently bound. The following chart indicates the precedence when binding rules to columns and user-defined datatypes where rules already exist:

Table 1-8: Rule binding precedence

New Rule Bound To	Old Rule Bound to User-Defined Datatype	Old Rule Bound to Column
User-defined datatype	New rule replaces old	No change
Column	New rule replaces old	New rule replaces old

Rules and Nulls

- Rules do not override column definitions. If a rule is bound to a column that allows null values, you can insert NULL into the column, implicitly or explicitly, even though NULL is not included in the text of the rule. For example, if you create a rule specifying "@val in (1,2,3)" or "@amount > 10000", and bind this rule to a table column that allows null values, you can still insert NULL into that column. The column definition overrides the rule.

Getting Information About Rules

- To get a report on a rule, use **sp_help**.
- To display the text of a rule, which is stored in the *syscomments* system table, execute the system procedure **sp_helptext** with the rule name as the parameter.
- After a rule is bound to a particular column or user-defined datatype, its ID is stored in the *syscolumns* or *systypes* system tables.

Defaults and Rules

- If a column has both a default and a rule associated with it, the default must fall within the domain defined by the rule. A default that conflicts with a rule will never be inserted. Adaptive Server generates an error message each time it attempts to insert the default.

Using Integrity Constraints in Place of Rules

- You can also define rules using **check** integrity constraints with the **create table** statement. However, these constraints are specific for that table; you cannot bind them to other tables. See **create table** and **alter table** for information about integrity constraints.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Transact-SQL extension	To create rules using SQL92-compliant syntax, use the **check** clause of the **create table** statement.

Permissions

create rule permission defaults to the Database Owner, who can transfer it to other users.

See Also

Commands	**alter table**, **create default**, **create table**, **drop default**, **drop rule**
System procedures	**sp_bindrule**, **sp_help**, **sp_helptext**, **sp_hidetext**, **sp_rename**, **sp_unbindrule**

create schema

Function

Creates a new collection of tables, views, and permissions for a database user.

Syntax

```
create schema authorization authorization_name
   create_oject_statement
       [ create_object_statement ... ]
   [ permission_statement ... ]
```

Keywords and Options

authorization_name – must be the name of the current user in the database.

create_object_statement – is a **create table** or **create view** statement.

permission_statement – is a **grant** or **revoke** command.

Examples

```
1. create schema authorization pogo
       create table newtitles (
           title_id tid not null,
           title varchar(30) not null)

       create table newauthors (
           au_id id not null,
           au_lname varchar(40) not null,
           au_fname varchar(20) not null)

       create table newtitleauthors (
           au_id id not null,
           title_id tid not null)

       create view tit_auth_view
       as
           select au_lname, au_fname
               from newtitles, newauthors,
                   newtitleauthors
           where
           newtitleauthors.au_id = newauthors.au_id
           and
           newtitleauthors.title_id =
                newtitles.title_id
```

```
grant select on tit_auth_view to public
revoke select on tit_auth_view from churchy
```

Creates the *newtitles*, *newauthors*, *newtitleauthors* tables, the *tit_auth_view* view, and the corresponding permissions.

Comments

- Schemas can be created only in the current database.
- The *authorization_name*, also called the **schema authorization identifier**, must be the name of the current user.
- The user must have the correct command permissions (**create table** and / or **create view**). If the user creates a view on tables owned by another database user, permissions on the view are checked when a user attempts to access data through the view, not when the view is created.
- The **create schema** command is terminated by:
  - The regular command terminator ("go" is the default in **isql**).
  - Any statement other than **create table**, **create view**, **grant**, or **revoke**.
- 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.
- **create schema** adds information about tables, views, and permissions to the system tables. Use the appropriate drop command (**drop table** or **drop view**) to drop objects created with **create schema**. Permissions granted or revoked in a schema can be changed with the standard **grant** and **revoke** commands outside the schema creation statement.

Standards and Compliance

Standard	Compliance Level
SQL92	Entry level compliant

Permissions

create schema can be executed by any user of a database. The user must have permission to create the objects specified in the schema; that is, **create table** and / or **create view** permission.

See Also

Commands	**create table**, **create view**, **grant**, **revoke**

create table

Function

Creates new tables and optional integrity constraints.

Syntax

```
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]
```

Keywords and Options

table_name – is the explicit name of the new table. Specify the database name if the table is in another database, and specify the owner's name if more than one table of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

You cannot use a variable for the table name. The table name must be unique within the database and to the owner. If you have **set quoted_identifier on**, you can use a delimited identifier for the table name. Otherwise, it must conform to the rules for identifiers. See “Identifiers” in Appendix A, “Expressions, Identifiers, and Wildcard Characters,” for more information about valid table names.

You can create a temporary table by preceding the table name with either a pound sign (#) or “tempdb..”. See “Tables Beginning with # (Temporary Tables)” in Appendix A, “Expressions, Identifiers, and Wildcard Characters,” for more information.

You can create a table in a different database, as long as you are listed in the *sysusers* table and have **create table** permission for that database. For example, to create a table called *newtable* in the database *otherdb*:

```
create table otherdb..newtable
```

or:

```
create table otherdb.yourname.newtable
```

column_name – is the name of the column in the table. It must be unique in the table. If you have **set quoted_identifier on**, you can use a delimited identifier for the column. Otherwise, it must conform to the rules for identifiers. See “Identifiers” in Appendix A, “Expressions, Identifiers, and Wildcard Characters,” for more information about valid column names.

datatype – is the datatype of the column. System or user-defined datatypes are acceptable. Certain datatypes expect a length, *n*, in parentheses:

```
 datatype(n)
```

Others expect a precision, *p*, and scale, *s*:

```
 datatype(p,s)
```

See “Datatypes” for more information.

default – specifies a default value for a column. If you specify a default, and the user does not provide a value for the column when inserting data, Adaptive Server inserts the default value. The default can be a constant expression, **user**, to insert the name of the user who is performing the insert, or **null**, to insert the null value. Adaptive Server generates a name for the default in the form of *tabname_colname_objid*, where *tabname* is the first 10 characters of the table name, *colname* is the first 5 characters of the column name, and *objid* is the object ID number for the default.

Defaults declared for columns with the IDENTITY property have no effect on column values.

constant_expression – is 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 can include built-in functions that do not reference database objects. This default value must be compatible with the datatype of the column, or Adaptive Server generates a datatype conversion error when attempting to insert the default.

user | **null** – specifies that Adaptive Server should insert the user name or the null value as the default if the user does not supply a value. For **user**, the datatype of the column must be either *char(30)* or *varchar(30*). For **null**, the column must allow null values.

identity – indicates that the column has the IDENTITY property. Each table in a database can have one IDENTITY column with a type of *numeric* and a scale of 0. IDENTITY columns are not updatable and do not allow nulls.

IDENTITY columns are used to store sequential numbers, such as invoice numbers or employee numbers, that are generated automatically by Adaptive Server. The value of the IDENTITY column uniquely identifies each row in a table.

null | **not null** – specifies Adaptive Server's behavior during data insertion if no default exists.

null specifies that Adaptive Server assigns a null value if a user does not provide a value.

not null specifies that a user must provide a non-null value if no default exists.

If you do not specify **null** or **not null**, Adaptive Server uses **not null** by default. However, you can switch this default using **sp_dboption** to make the default compatible with the SQL standards.

constraint – introduces the name of an integrity constraint.

constraint_name – is the name of the constraint. It must conform to the rules for identifiers and be unique in the database. If you do not specify the name for a referential or check constraint, Adaptive Server generates a name in the form *tabname_colname_objectid* where *tabname* is the first 10 characters of the table name, *colname* is the first 5 characters of the column name, and *objectid* is the object ID number for the constraint. If you do not specify the name for a unique or primary key constraint, Adaptive Server generates a name in the format *tabname_colname_tabindid* where *tabindid* is a string concatenation of the table ID and index ID.

unique – constrains the values in the indicated column or columns so that no two rows have the same value. This constraint creates a unique index that can be dropped only if the constraint is dropped using **alter table**.

primary key – constrains the values in the indicated column or columns so that no two rows have the same value, and so that the value cannot be NULL. This constraint creates a unique index that can be dropped only if the constraint is dropped using **alter table**.

clustered | **nonclustered** – specifies that the index created by a **unique** or **primary key** constraint is a clustered or nonclustered index. **clustered** is the default for primary key constraints; **nonclustered** is the default for unique constraints. There can be only one clustered index per table. See **create index** for more information.

fillfactor – specifies how full Adaptive Server will make each page when it is creating a new index on existing data. The **fillfactor** percentage is relevant only at the time the index is created. As the data changes, the pages are not maintained at any particular level of fullness.

The default for **fillfactor** is 0; this is used when you do not include **with fillfactor** in the **create index** statement (unless the value has been changed with **sp_configure**). When specifying a **fillfactor**, use a value between 1 and 100.

A **fillfactor** of 0 creates clustered indexes with completely full pages and nonclustered indexes with completely full leaf pages. It leaves a comfortable amount of space within the index B-tree in both the clustered and nonclustered indexes. There is seldom a reason to change the **fillfactor.**

If the **fillfactor** is set to 100, Adaptive Server creates both clustered and nonclustered indexes with each page 100 percent full. A **fillfactor** of 100 makes sense only for read-only tables—tables to which no additional data will ever be added.

fillfactor values smaller than 100 (except 0, which is a special case) cause Adaptive Server to create new indexes with pages that are not completely full. A **fillfactor** of 10 might be a reasonable choice if you are creating an index on a table that will eventually hold a great deal more data, but small **fillfactor** values cause each index (or index and data) to take more storage space.

If Component Integration Services is enabled, you cannot use **fillfactor** for remote servers.

◆ ***WARNING!***

Creating a clustered index with a fillfactor affects the amount of storage space your data occupies, since Adaptive Server redistributes the data as it creates the clustered index.

max_rows_per_page – limits the number of rows on data pages and the leaf level pages of indexes. Unlike **fillfactor**, the **max_rows_per_page** value is maintained when data is inserted or deleted.

If you do not specify a value for **max_rows_per_page**, Adaptive Server uses a value of 0 when creating the table. Values for tables and clustered indexes are between 0 and 256. The maximum number of rows per page for nonclustered indexes depends on the size of the index key; Adaptive Server returns an error message if the specified value is too high.

A **max_rows_per_page** of 0 creates clustered indexes with full data pages and nonclustered indexes with full leaf pages. It leaves a comfortable amount of space within the index B-tree in both clustered and nonclustered indexes.

Using low values for **max_rows_per_page** reduces lock contention on frequently accessed data. However, using low values also causes Adaptive Server to create new indexes with pages that are not completely full, uses more storage space, and may cause more page splits.

If Component Integration Services is enabled, you cannot use **max_rows_per_page** for remote servers.

on *segment_name* – specifies that the index is to be created on the named segment. Before the **on** *segment_name* option can be used, the device must be initialized with **disk init**, and the segment must be added to the database with the **sp_addsegment** system procedure. See your System Administrator or use **sp_helpsegment** for a list of the segment names available in your database.

If you specify **clustered** and use the **on** *segment_name* option, the entire table migrates to the segment you specify, since the leaf level of the index contains the actual data pages.

references – specifies a column list for a referential integrity constraint. You can specify only one column value for a column-constraint. By including this constraint with a table that references another table, any data inserted into the **referencing** table must already exist in the **referenced** table.

To use this constraint, you must have **references** permission on the referenced table. The specified columns in the referenced table must be constrained by a unique index (created by either a **unique** constraint or a **create index** statement). If no columns are specified, there must be a **primary key** constraint on the appropriate columns in the referenced table. Also, the datatypes of the referencing table columns must match the datatype of the referenced table columns.

foreign key – specifies that the listed column(s) are foreign keys in this table whose target keys are the columns listed in the following **references** clause. The foreign key syntax is permitted only for table-level constraints, not for column-level constraints.

ref_table – is the name of the table that contains the referenced columns. You can reference tables in another database. Constraints can reference up to 192 user tables and internally generated worktables.

ref_column – is the name of the column or columns in the referenced table.

check – specifies a *search_condition* constraint that Adaptive Server enforces for all the rows in the table. You can specify **check** constraints as table or column constraints; **create table** allows multiple **check** constraints in a column definition.

search_condition – is the **check** constraint on the column values. These constraints can include:

- A list of constant expressions introduced with **in**
- A set of conditions introduced with **like,** which may contain wildcard characters

Column and table check constraints can reference any columns in the table.

An expression can include arithmetic operators and functions. The *search_condition* cannot contain subqueries, aggregate functions, host variables, or parameters.

next_column | *next_constraint* – indicates that you can include additional column definitions or table constraints (separated by commas) using the same syntax described for a column definition or table constraint definition.

on *segment_name* – specifies the name of the segment on which to place the table. When using **on** *segment_name*, the logical device must already have been assigned to the database with **create database** or **alter database**, and the segment must have been created in the database with **sp_addsegment**. See your System Administrator or use **sp_helpsegment** for a list of the segment names available in your database.

Examples

1.
```
create table titles
(title_idtid 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)
```

   Creates the *titles* table.

2.
```
create table "compute"
("max" int, "min" int, "total score" int)
```

   Creates the *compute* table. The table name and the column names, *max* and *min*, are enclosed in double quotes because they are reserved words. The *total score* column name is enclosed in double quotes because it contains an embedded blank. Before creating this table, you must **set quoted_identifier on**.

3.
```
create table sales
(stor_id         char(4)      not null,
ord_num          varchar(20)  not null,
date             datetime     not null,
unique clustered (stor_id, ord_num))
```

   Creates the *sales* table and a clustered index in one step with a unique constraint. (In the *pubs2* database installation script, there are separate **create table** and **create index** statements.)

4.
```
create table salesdetail
(stor_id    char(4)                  not null,
ord_num     varchar(20)              not null,
title_id     tid                     not null
             references titles(title_id),
qty          smallintdefault 0       not null,
discount    float                    not null,

constraint salesdet_constr
    foreign key (stor_id, ord_num)
    references sales(stor_id, ord_num))
```

Creates the *salesdetail* table with two referential integrity constraints and one default value. There is a table-level, referential integrity constraint named *salesdet_constr* and a column-level, referential integrity constraint on the *title_id* column without a specified name. Both constraints specify columns that have unique indexes in the referenced tables (*titles* and *sales*). The **default** clause with the *qty* column specifies 0 as its default value.

5.
```
create table publishers
(pub_idchar(4)not null
   check (pub_id in ("1389", "0736", "0877",
       "1622", "1756")
    or pub_id like "99[0-9][0-9]"),
pub_name     varchar(40)  null,
city     varchar(20)  null,
state     char(2)  null)
```

Creates the table *publishers* with a check constraint on the *pub_id* column. This column-level constraint can be used in place of the *pub_idrule* included in the *pubs2* database:

```
create rule pub_idrule
as @pub_id in ("1389", "0736", "0877",
    "1622", "1756")
or @pub_id like "99[0-9][0-9]"
```

6.
```
create table sales_daily
(stor_id         char(4)       not null,
ord_num          numeric(10,0)   identity,
ord_amt          money         null)
```

Specifies the *ord_num* column as the IDENTITY column for the *sales_daily* table. The first time you insert a row into the table, Adaptive Server assigns a value of 1 to the IDENTITY column. On each subsequent insert, the value of the column increments by 1.

Comments

- **create table** creates a table and optional integrity constraints. The table is created in the currently open database unless you specify a different database in the **create table** statement. You can create a table or index in another database, if you are listed in the *sysusers* table and have **create table** permission in the database.
- Space is allocated to tables and indexes in increments of one extent, or eight pages, at a time. Each time an extent is filled, another extent is allocated. To see the amount of space allocated and used by a table, use **sp_spaceused**.
- When using **create table** from Component Integration Services with a column defined as *char*(*n*) NULL, Component Integration Services will create the column as *varchar*(*n*) on the remote server.

Restrictions

- There can be up to 2 billion tables per database and 250 user-defined columns per table. The number of rows per table is limited only by available storage.
- The maximum number of bytes per row is 1962. If you create tables with *varchar*, *nvarchar*, or *varbinary* columns whose total defined width is greater than 1962 bytes, a warning message appears, but the table is created. If you try to insert more than 1962 bytes into such a row, or to **update** a row so that its total row size is greater than 1962, Adaptive Server produces an error message, and the command fails.

➤ ***Note***

When a **create table** command occurs within an **if...else** block or a **while** loop, Adaptive Server creates the schema for the table before determining whether the condition is true. This may lead to errors if the table already exists. Make sure a table with the same name does not already exist in the database.

Column Definitions

- When you create a column from a user-defined datatype:
  - You cannot change the length, precision, or scale.
  - You can use a NULL type to create a NOT NULL column, but not to create an IDENTITY column.

- You can use a NOT NULL type to create a NULL column or an IDENTITY column.
- You can use an IDENTITY type to create a NOT NULL column, but the column inherits the IDENTITY property. You cannot use an IDENTITY type to create a NULL column.

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

The following table lists the fixed-length datatypes and the variable-length datatypes to which they are converted. Certain variable-length datatypes, such as *moneyn*, are reserved types that cannot be used to create columns, variables, or parameters:

Table 1-9: Variable-length datatypes used to store nulls

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*

• You can create column defaults in two ways: by declaring the default as a column constraint in the **create table** or **alter table** statement, or by creating the default using the **create default** statement and binding it to a column using **sp_bindefault**.

• For a report on a table and its columns, execute the system procedure **sp_help**.

Temporary Tables

• Temporary tables are stored in the temporary database, *tempdb*.

• The first 13 characters of a temporary table name that begins with a pound sign (including the pound sign) must be unique to a user, per session. Such tables can be accessed only by the current Adaptive Server session. They are stored in *tempdb..objects* by their names plus a system-supplied numeric suffix, and they disappear at the end of the current session or when they are explicitly dropped.

- Temporary tables created with the "tempdb.." prefix are shareable among Adaptive Server sessions. They exist until they are explicitly dropped by their owner or until Adaptive Server reboots. Create temporary tables with the "tempdb.." prefix from inside a stored procedure only if you intend to share the table among users and sessions. To avoid inadvertent sharing of temporary tables, use the # prefix when creating and dropping temporary tables in stored procedures.
- You can associate rules, defaults and indexes with temporary tables, but you cannot create views on temporary tables or associate triggers with them.
- When you create a temporary table, you can use a user-defined datatype only if the type is in *tempdb..systypes*. To add a user-defined datatype to *tempdb* for the current session only, execute **sp_addtype** while using *tempdb*. To add the datatype permanently, execute **sp_addtype** while using *model*, and then restart Adaptive Server so that *model* is copied to *tempdb*.

Using Indexes

- A table "follows" its clustered index. If you create a table on one segment and then create its clustered index on another segment, the table migrates to the segment where the index is created.
- You can make inserts, updates, and selects faster by creating a table on one segment and its nonclustered indexes on another segment, if the segments are on separate physical devices. See "How Object Placement Can Improve Performance" in Chapter 17, "Controlling Physical Data Placement" in the *Performance and Tuning Guide* for more information.

Renaming a Table or Its Columns

- Use **sp_rename** to rename a table or column.
- After renaming a table or any of its columns, use **sp_depends** to determine which procedures, triggers, and views depend on the table, and redefine these objects.

◆ ***WARNING!***

If you do not redefine these dependent objects, they will no longer work after Adaptive Server recompiles them.

Defining Integrity Constraints

- The **create table** statement helps control a database's integrity through a series of integrity constraints as defined by the SQL standards. These integrity constraint clauses restrict the data that users can insert into a table. You can also use defaults, rules, indexes, and triggers to enforce database integrity.

  Integrity constraints offer the advantages of defining integrity controls in one step during the table creation process 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.

- You must declare constraints that operate on more than one column as table-level constraints; declare constraints that operate on just one column as column-level constraints. The difference is syntactic: you place column-level constraints after the column name and datatype, before the delimiting comma (see example 5). You enter table-level constraints as separate comma-delimited clauses (see example 4). Adaptive Server treats table-level and column-level constraints the same way; neither way is more efficient than the other.

- You can create the following types of constraints at the table level or the column level:

  - A **unique** constraint requires that no two rows in a table have the same values in the specified columns. In addition, a **primary key** constraint requires that there be no null values in the column.
  - A referential integrity (**references**) constraint requires that the data being inserted or updated in specific columns has matching data in the specified table and columns.
  - A **check** constraint limits the values of the data inserted into the 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).

- You can use the system procedures **sp_primarykey**, **sp_foreignkey**, and **sp_commonkey** to save information in system tables, which can help clarify the relationships between tables in a database. These system procedures do not enforce the key relationships or duplicate the functions of the **primary key** and **foreign key** keywords in a **create table** statement. For a report on keys that have been defined, use **sp_helpkey**. For a report on frequently used joins, execute **sp_helpjoins**.
- Transact-SQL provides several mechanisms for integrity enforcement. In addition to the constraints you can declare as part of **create table**, you can create rules, defaults, indexes, and triggers. The following table summarizes the integrity constraints and describes the other methods of integrity enforcement:

Table 1-10: Methods of integrity enforcement

In *create table*	Other Methods
unique constraint	**create unique index** (on a column that allows null values)
primary key constraint	**create unique index** (on a column that does not allow null values)
references constraint	**create trigger**
check constraint (table level)	**create trigger**
check constraint (column level)	**create trigger** or **create rule** and **sp_bindrule**
default clause	**create default** and **sp_bindefault**

  Which method you choose depends on your requirements. For example, triggers provide more complex handling of referential integrity (such as referencing other columns or objects) than those declared in **create table**. Also, the constraints defined in a **create table** statement are specific for that table; unlike rules and 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 **create table** command can include many constraints, with these limitations:
  - The number of **unique** constraints is limited by the number of indexes that table can have.
  - A table can have only one **primary key** constraint.

- You can include only one **default** clause per column in a table, but you can define different constraints on the same column.

For example:

```
create table discount_titles
(title_id   varchar(6)   default "PS7777" not null
        unique clustered
        references titles(title_id)
        check (title_id like "PS%"),
             new_price   money)
```

Column *title_id* of the new table *discount_titles* is defined with each integrity constraint.

- You can create error messages and bind them to referential integrity and **check** constraints. Create messages with **sp_addmessage** and bind them to the constraints with **sp_bindmsg**. For more information, see **sp_addmessage** and **sp_bindmsg**.
- Adaptive Server evaluates check constraints before enforcing the referential constraints, and evaluates triggers after enforcing all the integrity constraints. If any constraint fails, Adaptive Server cancels the data modification statement; any associated triggers do not execute. However, a constraint violation **does not** roll back the current transaction.
- In a referenced table, you cannot update column values or delete rows that match values in a referencing table. Update or delete from the referencing table first, and then try updating or deleting from the referenced table.
- You must drop the referencing table before you drop the referenced table; otherwise, a constraint violation will occur.
- For information about constraints defined for a table, use **sp_helpconstraint**.

Unique and Primary Key Constraints

- You can declare **unique** constraints at the column level or the table level. **unique** constraints require that all values in the specified columns be unique. No two rows in the table can have the same value in the specified column.
- A **primary key** constraint is a more restrictive form of **unique** constraint. Columns with **primary key** constraints cannot contain null values.

➤ ***Note***

The **create table** statement's **unique** and **primary key** constraints create indexes that define unique or primary key attributes of columns. **sp_primarykey**, **sp_foreignkey**, and **sp_commonkey** define logical relationships between columns. These relationships must be enforced using indexes and triggers.

- Table-level **unique** or **primary** key constraints appear in the **create table** statement as separate items and must include the names of one or more columns from the table being created.
- **unique** or **primary key** constraints create a unique index on the specified columns. The **unique** constraint in example 3 creates a unique, clustered index, as does the statement:

  ```
  create unique clustered index salesind
      on sales (stor_id, ord_num)
  ```

  The only difference is the index name, which you could set to *salesind* by naming the constraint.
- The definition of **unique** constraints in the SQL standards specifies that the column definition cannot 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** 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.
- **unique** constraints create unique, nonclustered indexes by default; **primary key** constraints create unique, clustered indexes by default. There can be only one clustered index on a table, so you can specify only one **unique clustered** or **primary key clustered** constraint.
- The **unique** and **primary key** constraints of **create table** offer a simpler alternative to the **create index** statement. However, they have the following limitations:
  - You cannot create nonunique indexes.
  - You cannot use all the options provided by **create index**.
  - You must drop these indexes using **alter table drop constraint**.

Referential Integrity Constraints

- Referential integrity constraints require that data inserted into a **referencing** table that defines the constraint must have matching values in a **referenced** table. A referential integrity constraint is satisfied for either of the following conditions:
  - The data in the constrained column(s) of the referencing table contains a null value.
  - The data in the constrained column(s) of the referencing table matches data values in the corresponding columns of the referenced table.

  Using the *pubs2* database as an example, a row inserted into the *salesdetail* table (which records the sale of books) must have a valid *title_id* in the *titles* table. *salesdetail* is the referencing table and *titles* table is the referenced table. Currently, *pubs2* enforces this referential integrity using a trigger. However, the *salesdetail* table could include this column definition and referential integrity constraint to accomplish the same task:

  ```
  title_id tid
      references titles(title_id)
  ```

- The maximum number of table references allowed for a query is 192. Use the system procedure **sp_helpconstraint** to check a table's referential constraints.
- A table can include a referential integrity constraint on itself. For example, the *store_employees* table in *pubs3*, which lists employees and their managers, has the following self-reference between the *emp_id* and *mgr_id* columns:

  ```
  emp_id id primary key,
  mgr_id id null
         references store_employees(emp_id),
  ```

  This constraint ensures that all managers are also employees, and that all employees have been assigned a valid manager.
- You cannot drop the referenced table until the referencing table is dropped or the referential integrity constraint is removed (unless it includes only a referential integrity constraint on itself).
- Adaptive Server does not enforce referential integrity constraints for temporary tables.
- To create a table that references another user's table, you must have **references** permission on the referenced table. For

information about assigning **references** permissions, see the **grant** command.

- Table-level, referential integrity constraints appear in the **create table** statement as separate items. They must include the **foreign key** clause and a list of one or more column names.

  Column names in the **references** clause are optional only if the columns in the referenced table are designated as a primary key through a **primary key** constraint.

  The referenced columns must be constrained by a unique index in that referenced table. You can create that unique index using either the **unique** constraint or the **create index** statement.

- The datatypes of the referencing table columns must match the datatypes of the referenced table columns. For example, the datatype of *col1* in the referencing table (*test_type*) matches the datatype of *pub_id* in the referenced table (*publishers*):

```
create table test_type
(col1 char(4) not null
    references publishers(pub_id),
col2 varchar(20) not null)
```

- The referenced table must exist at the time you define the referential integrity constraint. For tables that cross-reference one another, use the **create schema** statement to define both tables simultaneously. As an alternative, create one table without the constraint and add it later using **alter table**. See **create schema** or **alter table** for more information.

- The **create table** referential integrity constraints offer a simple way to enforce data integrity. Unlike triggers, they **cannot**:

  - Cascade changes through related tables in the database
  - Enforce complex restrictions by referencing other columns or database objects
  - Perform “what-if” analysis

  Referential integrity constraints do not roll back transactions when a data modification violates the constraint. Triggers allow you to choose whether to roll back or continue the transaction depending on how you handle referential integrity.

➤ ***Note***

Adaptive Server checks referential integrity constraints before it checks any triggers, so a data modification statement that violates the constraint does not also fire the trigger.

Using Cross-Database Referential Integrity Constraints

- When you create a cross-database constraint, Adaptive Server stores the following information in the *sysreferences* system table of each database:

Information	***sysreferences*** **Column for the Referenced Table**	***sysreferences*** **Column for the Referencing Table**
Key column IDs	*refkey1* through *refkey16*	*fokey1* through *fokey16*
Table ID	*reftabid*	*tableid*
Database name	*pmrydbname*	*frgndbname*

- You can drop the referencing table or its database without problems. Adaptive Server automatically removes the foreign key information from the referenced database.
- Because the referencing table depends on information from the referenced table, Adaptive Server does not allow you to:
  - Drop the referenced table,
  - Drop the external database that contains the referenced table, or
  - Rename either database with **sp_renamedb**.

  You must remove the cross-database constraint with **alter table** before you can do any of these actions.
- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of databases containing cross-database constraints could cause database corruption.

- The *sysreferences* system table stores the **name**—not the ID number—of the external database. Adaptive Server cannot guarantee referential integrity if you use **load database** to change the database name or to load it onto a different server.

◆ ***WARNING!***

Before dumping a database in order to load it with a different name or move it to another Adaptive Server, use alter table to drop all external referential integrity constraints.

check Constraints

- A **check** constraint limits the values a user can insert into a column in a table. A **check** constraint specifies a *search_condition* that any non-null 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 can contain wildcard characters

  An expression can include arithmetic operators and Transact-SQL built-in functions. The *search_condition* cannot contain subqueries, aggregate functions, or a host variable or parameter. Adaptive Server does not enforce **check** constraints for temporary tables.
- If the **check** constraint is a column-level **check** constraint, it can reference only the column in which it is defined; it cannot reference other columns in the table. Table-level **check** constraints can reference any column in the table.
- **create table** allows multiple **check** constraints in a column definition.
- **check** integrity constraints offer an alternative to using rules and triggers. They are specific to the table in which they are created, and cannot be bound to columns in other tables or to user-defined datatypes.
- **check** constraints do not override column definitions. 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, if you create a **check** constraint specifying “pub_id in (“1389”, “0736”, “0877”, “1622”, “1756”)” or “@amount > 10000” in a table column that allows null values, you can still insert NULL into that column. The column definition overrides the **check** constraint.

IDENTITY Columns

- The first time you insert a row into the table, Adaptive Server assigns the IDENTITY column a value of 1. Each new row gets a column value that is 1 higher than the last value. This value takes precedence over any defaults declared for the column in the **create table** statement or bound to the column with the **sp_bindefault** system procedure. The maximum value that can be inserted into the IDENTITY column is $10^{\text{PRECISION}} - 1$.
- Inserting a value into the IDENTITY column allows you to specify a seed value for the column or to restore a row that was deleted in error. The table owner, Database Owner, or System Administrator can explicitly insert a value into an IDENTITY column after using **set identity_insert** *table_name* **on** for the base table. Unless you have created a unique index on the IDENTITY column, Adaptive Server does not verify the uniqueness of the value. You can insert any positive integer.
- You can reference an IDENTITY column using the **syb_identity** keyword, qualified by the table name where necessary, in place of the actual column name.
- System Administrators can use the **auto identity** database option to automatically include a 10-digit IDENTITY column in new tables. To turn on this feature in a database, use:

  ```
  sp_dboption database_name, "auto identity", "true"
  ```

  Each time a user creates a table in the database without specifying either a **primary** key, a **unique** constraint, or an IDENTITY column, Adaptive Server automatically defines an IDENTITY column. This column, SYB_IDENTITY_COL, is not visible when you retrieve columns with the **select** * statement. You must explicitly include the column name in the select list.
- Server failures can create gaps in IDENTITY column values. The maximum size of the gap depends on the setting of the **identity burning set factor** configuration parameter. Gaps can also occur due to transaction rollbacks, the deletion of rows, or the manual insertion of data into the IDENTITY column.

Getting Information on Tables

- **sp_help** displays information about tables, listing any attributes (such as cache bindings) assigned to the specified table and its indexes, giving the attribute's class, name, integer value, character value, and comments.
- **sp_depends** displays information about the view(s), trigger(s), and procedure(s) in the database that depend on a table.
- **sp_helpindex** reports information about the indexes created on a table.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The following are Transact-SQL extensions: • The **on segment_name** clause • Use of a database name to qualify a table or column name • IDENTITY columns • The **not null** column default See "System and User-Defined Datatypes" for datatype compliance information.

Permissions

create table permission defaults to the Database Owner, who can transfer it to other users. Any user can create temporary tables.

See Also

Commands	**alter table**, **create index**, **create rule**, **create schema**, **create view**, **drop index**, **drop rule**, **drop table**
System procedures	**sp_addmessage**, **sp_addsegment**, **sp_addtype**, **sp_bindmsg**, **sp_commonkey**, **sp_depends**, **sp_foreignkey**, **sp_help**, **sp_helpjoins**, **sp_helpsegment**, **sp_primarykey**, **sp_rename**, **sp_spaceused**

create trigger

Function

Creates a trigger, a type of stored procedure that is often used for enforcing integrity constraints. A trigger executes automatically when a user attempts a specified data modification statement on a specified table.

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]...
```

Keywords and Options

trigger_name – is the name of the trigger. It must conform to the rules for identifiers and be unique in the database. Specify the owner's name to create another trigger of the same name owned by a different user in the current database. The default value for *owner* is the current user. If you use an owner name to qualify a trigger, you must explicitly qualify the table name the same way.

You cannot use a variable for a trigger name.

table_name – is the name of the table on which to create the trigger. If more than one table of the same name exists in the database, specify the owner's name. The default value for *owner* is the current user.

insert, **update**, **delete** – can be included in any combination. **delete** cannot be used with the **if update** clause.

SQL_statements – specify trigger conditions and trigger actions. Trigger conditions determine whether the attempted **insert**, **update**, or **delete** causes the trigger actions to be carried out. The SQL statements often include a subquery preceded by the keyword **if**. In example 2, below, the subquery that follows the keyword **if** is the trigger condition.

Trigger actions take effect when the user action (**insert**, **update**, or **delete**) is attempted. If multiple trigger actions are specified, they are grouped with **begin** and **end**.

See "Triggers and Transactions" for a list of statements that are not allowed in a trigger definition. See "The deleted and inserted Logical Tables" for information about the *deleted* and *inserted* logical tables that can be included in trigger definitions.

if update – is used to test whether the specified column is included in the **set** list of an **update** statement or is affected by an **insert**. This allows specified trigger actions to be associated with updates to specified columns (see example 3). More than one column can be specified, and you can use more than one **if update** statement in a **create trigger** statement (see example 5).

Examples

1.
```
create trigger reminder
on titles
for insert, update as
print "Don't forget to print a report for
accounting."
```

Prints a message when anyone tries to add data or change data in the *titles* table.

2.
```
create trigger t1
on titleauthor
for insert as
if (select count(*)
     from titles, inserted
     where titles.title_id = inserted.title_id) = 0
begin
print "Please put the book's title_id in the
        titles table first."
rollback transaction
end
```

Prevents insertion of a new row into *titleauthor* if there is no corresponding *title_id* in the *titles* table.

3.
```
create trigger t2
on publishers
for update as
if update (pub_id) and @@rowcount = 1
begin
    update titles
    set titles.pub_id = inserted.pub_id
    from titles, deleted, inserted
    where deleted.pub_id = titles.pub_id
end
```

If the *pub_id* column of the *publishers* table is changed, make the corresponding change in the *titles* table.

4.
```
create trigger t3
on titleauthor
for delete as
begin
    delete titles
    from titles, deleted
    where deleted.title_id = titles.title_id
    delete titleauthor
    from titleauthor, deleted
    where deleted.title_id = titleauthor.title_id
    print "All references to this title have been
    deleted from titles and titleauthor."
end
```

Deletes title from the *titles* table if any row is deleted from *titleauthor*. If the book was written by more than one author, other references to it in *titleauthor* are also deleted.

5.
```
create trigger stopupdatetrig
on titles
for update
as
if update (title_id)
  and datename(dw, getdate())
  in ("Saturday", "Sunday")
  begin
    rollback transaction
    print "We don’t allow changes to"
    print "primary keys on the weekend!"
  end
```

```
if update (price) or update (advance)
  if (select count(*) from inserted
    where (inserted.price * inserted.total_sales)
    < inserted.advance) > 0
    begin
      rollback transaction
      print "We don’t allow changes to price or"
      print "advance for a title until its total"
      print "revenue exceeds its latest advance."
    end
```

Prevents updates to the primary key on weekends. Prevents updates to the price or advance of a title unless the total revenue amount for that title surpasses its advance amount.

Comments

- A trigger fires only once per data modification statement. A complex query containing a **while** loop may repeat an **update** or **insert** many times, and the trigger is fired each time.

Triggers and Referential Integrity

- Triggers are commonly used to enforce **referential integrity** (integrity rules about relationships between the primary and foreign keys of tables or views), to supply cascading deletes, and to supply cascading updates (see examples 2, 3, and 4, respectively).
- A trigger fires only after the data modification statement has completed and Adaptive Server has checked for any datatype, rule, or integrity constraint violations. The trigger and the statement that fires it are treated as a single transaction that can be rolled back from within the trigger. If a severe error is detected, the entire transaction is rolled back.
- You can also enforce referential integrity using constraints defined with the **create table** statement as an alternative to using **create trigger.** See **create table** and **alter table** for information about integrity constraints.

The *deleted* and *inserted* Logical Tables

- *deleted* and *inserted* are logical (conceptual) tables. They are structurally like the table for which the trigger is defined—that is, the table on which the user action is attempted—and hold the old values or new values of the rows that would be changed by the user action.

- *deleted* and *inserted* tables can be examined by the trigger to determine whether or how the trigger action(s) should be carried out, but the tables themselves cannot be altered by the trigger's actions.
- *deleted* tables are used with **delete** and **update**; *inserted* tables, with **insert** and **update**. (An update is a delete followed by an insert: it affects the *deleted* table first and then the *inserted* table).

Trigger Restrictions

- You can create a trigger only in the current database. If you use an owner name to qualify a trigger, you must explicitly qualify the table name the same way. A trigger can reference objects outside the current database.
- A trigger cannot apply to more than one table. However, the same trigger action can be defined for more than one user action (for example, **insert** and **update**) in the same **create trigger** statement. A table can have a maximum of three triggers—one each for **insert**, **update**, and **delete**.
- Each new trigger in a table or column for the same operation (**insert**, **update**, or **delete**) overwrites the previous one. No warning message is given before the overwrite occurs.
- You cannot create a trigger on a temporary table.
- You cannot create a trigger on a view.
- You cannot create a trigger on a system table.
- You cannot use triggers that select from a *text* or *image* column of the inserted or deleted table.
- It is recommended that a trigger not include **select** statements that return results to the user, since special handling for these returned results would have to be written into every application program that allows modifications to the trigger table.
- If a trigger references table names, column names, or view names that are not valid identifiers, you must **set quoted_identifier on** before the **create trigger** command and enclose each such name in double quotes. The **quoted_identifier** option does **not** need to be on when the trigger fires.

Getting Information About Triggers

- The execution plan for a trigger is stored in *sysprocedures*.

- Each trigger is assigned an identification number, which is stored as a new row in *sysobjects* with the object ID for the table to which it applies in the *deltrig* column, and also as an entry in the *deltrig*, *instrig*, and *updtrig* columns of the *sysobjects* row for the table to which it applies.
- To display the text of a trigger, which is stored in *syscomments*, use **sp_helptext**.

  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 text of the trigger through **sp_helptext**.
- For a report on a trigger, use **sp_help**.
- For a report on the tables and views that are referenced by a trigger, use **sp_depends**.

Triggers and Performance

- In performance terms, trigger overhead is usually very low. The time involved in running a trigger is spent mostly in referencing other tables, which are either in memory or on the database device.
- The *deleted* and *inserted* tables often referenced by triggers are always in memory rather than on the database device, because they are logical tables. The location of other tables referenced by the trigger determines the amount of time the operation takes.

Setting Options Within Triggers

- You can use the **set** command inside a trigger. The **set** option you invoke remains in effect during the execution of the trigger and then reverts to its former setting. In particular, the **self_recursion** option can be used inside a trigger so that data modifications by the trigger itself can cause the trigger to fire again.

Dropping a Trigger

- You must drop and re-create the trigger if you rename any of the objects referenced by the trigger. You can rename a trigger with **sp_rename**.
- When you drop a table, any triggers associated with it are also dropped.

Actions That Do Not Cause Triggers to Fire

- A **truncate table** command is not caught by a **delete** trigger. Although a **truncate table** statement is, in effect, like a **delete** without a **where** clause (it removes all rows), changes to the data rows are not logged, and so cannot fire a trigger.

  Since permission for the **truncate table** command defaults to the table owner and is not transferable, only the table owner need worry about inadvertently circumventing a **delete** trigger with a **truncate table** statement.
- The **writetext** command, whether logged or unlogged, does not cause a trigger to fire.

Triggers and Transactions

- When a trigger is defined, the action it specifies on the table to which it applies is always implicitly part of a transaction, along with the trigger itself. Triggers are often used to roll back an entire transaction if an error is detected, or they can be used roll back the effects of a specific data modification:
  - When the trigger contains the **rollback transaction** command, the rollback aborts the entire batch, and any subsequent statements in the batch are not executed.
  - When the trigger contains the **rollback trigger**, the rollback affects only the data modification that caused the trigger to fire. The **rollback trigger** command can include a **raiserror** statement. Subsequent statements in the batch are executed.
- Since triggers execute as part of a transaction, the following statements and system procedures 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**
  - **sp_configure**
  - **load database** and **load transaction**

- **disk init**, **disk mirror**, **disk refit**, **disk reinit**, **disk remirror**, **disk unmirror**
- **select into**

- If a desired result (such as a summary value) depends on the number of rows affected by a data modification, use *@@rowcount* to test for multirow data modifications (an **insert**, **delete**, or **update** based on a **select** statement), and take appropriate actions. Any Transact-SQL statement that does not return rows (such as an **if** statement) sets *@@rowcount* to 0, so the test of *@@rowcount* should occur at the beginning of the trigger.

Update and Insert Triggers

- When an **insert** or **update** command executes, Adaptive Server adds rows to both the trigger table and the *inserted* table at the same time. The rows in the *inserted* table are always duplicates of one or more rows in the trigger table.
- An **update** or **insert** trigger can use the **if update** command to determine whether the **update** or **insert** changed a particular column. **if update**(*column_name*) is true for an **insert** statement whenever the column is assigned a value in the select list or in the **values** clause. An explicit NULL or a default assigns a value to a column and thus activates the trigger. An implicit NULL, however, does not.

  For example, if you create the following table and trigger:

  ```
  create table junk
  (aaa int null,
  bbb int not null)

  create trigger trigtest on junk
  for insert as
  if update (aaa)
      print "aaa updated"
  if update (bbb)
      print "bbb updated"
  ```

  Inserting values into either column or into both columns fires the trigger for both column *aaa* and column *bbb*:

  ```
  insert junk (aaa, bbb)
  values (1, 2)

  aaa updated
  bbb updated
  ```

Inserting an explicit null into column *aaa* also fires the trigger:

```
insert junk
values (NULL, 2)
```

```
aaa updated
bbb updated
```

If there was a default for column *aaa*, the trigger would also fire.

However, with no default for column *aaa* and no value explicitly inserted, Adaptive Server generates an implicit NULL and the trigger does not fire:

```
insert junk (bbb)
values(2)
```

```
bbb updated
```

if update is never true for a **delete** statement.

Nesting Triggers and Trigger Recursion

- Adaptive Server allows nested triggers by default. To prevent triggers from nesting, use **sp_configure** to set the **allow nested triggers** option to 0 (off), as follows:

  ```
  sp_configure "allow nested triggers", 0
  ```

- Triggers can be nested to a depth of 16 levels. If a trigger changes a table on which there is another trigger, the second trigger will fire and can then call a third trigger, and so forth. If any trigger in the chain sets off an infinite loop, the nesting level will be exceeded and the trigger will abort, rolling back the transaction that contains the trigger query.

➤ ***Note***

Since triggers are put into a transaction, a failure at any level of a set of nested triggers cancels the entire transaction: all data modifications are rolled back. Supply your triggers with messages and other error handling and debugging aids in order to determine where the failure occurred.

- The global variable *@@nestlevel* contains the nesting level of the current execution. 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.

- If a trigger calls a stored procedure that performs actions that would cause the trigger to fire again, the trigger is reactivated only if nested triggers are enabled. Unless there are conditions within the trigger that limit the number of recursions, this causes a nesting-level overflow.

  For example, if an update trigger calls a stored procedure that performs an update, the trigger and stored procedure execute once if **allow nested triggers** is off. If **allow nested triggers** is on, and the number of updates is not limited by a condition in the trigger or procedure, the procedure or trigger loop continues until it exceeds the 16-level maximum nesting value.
- By default, a trigger does not call itself in response to a second data modification to the same table within the trigger, regardless of the setting of the **allow nested triggers** configuration parameter. A **set** option, **self_recursion**, enables a trigger to fire again as a result of a data modification within the trigger. For example, if an update trigger on one column of a table results in an update to another column, the update trigger fires only once when **self_recursion** is disabled, but it can fire up to 16 times if **self_recursion** is set on. The **allow nested triggers** configuration parameter must also be enabled in order for self-recursion to take place.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Only a System Security Officer can grant or revoke permissions to create triggers.

Permission to issue the **create trigger** command is granted to users by default. When you revoke permission for a user to create triggers, a revoke row is added in the *sysprotects* table for that user. To grant permission to that user to issue **create trigger**, you must issue two **grant** commands. The first command removes the revoke row from *sysprotects*; the second inserts a grant row.

If you revoke permission to create triggers, the user cannot create triggers even on tables that the user owns. Revoking permission to create triggers from a user affects only the database where the revoke command was issued.

Permissions on Objects: Trigger Creation Time

When you create a trigger, Adaptive Server makes no permission checks on objects, such as tables and views, that the trigger references. Therefore, you can create a trigger successfully, even though you do not have access to its objects. All permission checks occur when the trigger fires.

Permissions on Objects: Trigger Execution Time

When the trigger executes, permission checks on its objects depend on whether the trigger and its objects are owned by the same user.

- If the trigger and its objects are not owned by the same user, the user who caused the trigger to fire must have been granted direct access to the objects. For example, if the trigger performs a select from a table the user cannot access, the trigger execution fails. In addition, the data modification that caused the trigger to fire is rolled back.
- If a trigger and its objects are owned by the same user, special rules apply. The user automatically has implicit permission to access the trigger's objects, even though the user cannot access them directly. A detailed description of the rules for implicit permissions is discussed in Chapter 5, "Managing User Permissions," in the *Security Administration Guide*.

See Also

Commands	**alter table**, **create procedure**, **create table**, **drop trigger**, **rollback trigger**, **set**
System procedures	**sp_commonkey**, **sp_configure**, **sp_depends**, **sp_foreignkey**, **sp_help**, **sp_helptext**, **sp_primarykey**, **sp_rename**, **sp_spaceused**

create view

Function

Creates a view, which is an alternative way of looking at the data in one or more tables.

Syntax

```
create view [owner.]view_name
   [(column_name [, column_name]...)]
   as select [distinct] select_statement
   [with check option]
```

Keywords and Options

view_name – is the name of the view. The name cannot include the database name. If you have **set quoted_identifier on**, you can use a delimited identifier. Otherwise, the view name cannot be a variable and must conform to the rules for identifiers. See "Identifiers" in Appendix A, "Expressions, Identifiers, and Wildcard Characters" for more information about valid view names. Specify the owner's name to create another view of the same name owned by a different user in the current database. The default value for *owner* is the current user.

column_name – specifies names to be used as headings for the columns in the view. If you have set **quoted_identifier on**, you can use a delimited identifier. Otherwise, the column name must conform to the rules for identifiers. See "Identifiers" in Appendix A, "Expressions, Identifiers, and Wildcard Characters" for more information about valid column names.

It is always legal to supply column names, but column names are required only in the following cases:

- When a column is derived from an arithmetic expression, function, string concatenation, or constant
- When two or more columns have the same name (usually because of a join)
- When you want to give a column in a view a different name than the column from which it is derived (see example 3).

Column names can also be assigned in the **select** statement (see example 4). If no column names are specified, the view columns acquire the same names as the columns in the **select** statement.

select – begins the **select** statement that defines the view.

distinct – specifies that the view cannot contain duplicate rows.

select_statement – completes the **select** statement that defines the view. It can use more than one table and other views.

with check option – indicates that all data modification statements are validated against the view selection criteria. All rows inserted or updated through the view must remain visible through the view.

Examples

1.
   ```
   create view titles_view
   as select title, type, price, pubdate
   from titles
   ```

   Creates a view derived from the *title*, *type*, *price* and *pubdate* columns of the base table *titles*.

2.
   ```
   create view "new view" ("column 1", "column 2")
   as select col1, col2 from "old view"
   ```

   Creates the "new view" view from "old view." Both columns are renamed in the new view. All view and column names that include embedded blanks are enclosed in double quotation marks. Before creating the view, you must use **set quoted_identifier on**.

3.
   ```
   create view accounts (title, advance, amt_due)
   as select title, advance, price * total_sales
   from titles
   where price > $5
   ```

   Creates a view which contains the titles, advances and amounts due for books with a price less than $5.00.

4.
   ```
   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
   ```

   Creates a view derived from two base tables, *authors* and *publishers*. The view contains the names and cities of authors who live in a city in which there is a publisher.

5.
```
create view cities2
as select authorname = au_lname,
acity = authors.city, publishername = pub_name,
pcity = publishers.city
from authors, publishers
where authors.city = publishers.city
```

   Creates a view with the same definition as in example 3, but with column headings provided in the **select** statement.

6.
```
create view author_codes
as select distinct au_id
from titleauthor
```

   Creates a view, *author_codes*, derived from *titleauthor* that lists the unique author identification codes.

7.
```
create view price_list (price)
as select distinct price
from titles
```

   Creates a view, *price_list*, derived from *title* that lists the unique book prices.

8.
```
create view stores_cal
as select * from stores
where state = "CA"
with check option
```

   Creates a view of the *stores* table that excludes information about stores outside of California. The **with check option** clause validates each inserted or updated row against the view's selection criteria. Rows for which *state* has a value other than "CA" are rejected.

9.
```
create view stores_cal30
as select * from stores_cal
where payterms = "Net 30"
```

   Creates a view, *stores_cal30*, which is derived from *stores_cal*. The new view inherits the check option from *stores_cal*. All rows inserted or updated through *stores_cal*30 must have a *state* value of "CA". Because *stores_cal30* has no **with check option** clause, it is possible to insert or update rows through *stores_cal30* for which *payterms* has a value other than "Net 30".

10.
```
create view stores_cal30_check
as select * from stores_cal
where payterms = "Net 30"
with check option
```

Creates a view, *stores_cal30_check*, derived from *stores_cal*. The new view inherits the check option from *stores_cal*. It also has a **with check option** clause of its own. Each row that is inserted or updated through *stores_cal30_check* is validated against the selection criteria of both stores_cal and *stores_cal30_check*. Rows with a *state* value other than "CA" or a *payterms* value other than "Net 30" are rejected.

Comments

- You can use views as security mechanisms by granting permission on a view, but not on its underlying tables.
- You can rename a view with **sp_rename**.
- When you query through a view, Adaptive Server checks to make sure that all the database objects referenced anywhere in the statement exist, that they are valid in the context of the statement, and that data update commands do not violate data integrity rules. If any of these checks fail, you get an error message. If the checks are successful, **create view** "translates" the view into an action on the underlying table(s).
- For more information about views, see Chapter 9, "Views: Limiting Access to Data," in the *Transact-SQL User's Guide*.

Restrictions on Views

- You can create a view only in the current database.
- The number of columns referenced by a view cannot exceed 250.
- You cannot create a view on a temporary table.
- You cannot create a trigger or build an index on a view.
- You cannot use **readtext** or **writetext** on *text* or *image* columns in views.
- You cannot include **order by** or **compute** clauses, the keyword **into**, or the **union** operator in the **select** statements that define views.
- **create view** statements can be combined with other SQL statements in a single batch.

◆ ***WARNING!***

When a create view command occurs within an if...else block or a while loop, Adaptive Server creates the schema for the view before determining whether the condition is true. This may lead to errors if the view already exists. Make sure a view with the same name does not already exist in the database.

View Resolution

- If you alter the structure of a view's underlying table(s) by adding or deleting columns, the new columns will not appear in a view defined with a **select** * clause unless the view is dropped and redefined. The asterisk shorthand is interpreted and expanded when the view is first created.
- If a view depends on a table (or view) that has been dropped, Adaptive Server produces an error message when anyone tries to use the view. If a new table (or view) with the same name and schema is created to replace the one that has been dropped, the view again becomes usable.
- You can redefine a view without redefining other views that depend on it, unless the redefinition makes it impossible for Adaptive Server to translate the dependent view(s).

Modifying Data Through Views

- **delete** statements are not allowed on multitable views.
- **insert** statements are not allowed unless all **not null** columns in the underlying table or view are included in the view through which you are inserting new rows. (Adaptive Server cannot supply values for **not null** columns in the underlying table or view.)
- You cannot insert a row through a view that includes a computed column.
- **insert** statements are not allowed on join views created with **distinct** or **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.

- You cannot update or insert into a view defined with the **distinct** clause.
- Data update statements cannot change any column in a view that is a computation and cannot change a view that includes aggregates.

IDENTITY Columns and Views

- You cannot add a new IDENTITY column to a view with the *column_name* = **identity**(*precision*) syntax.
- To insert an explicit value into an IDENTITY column, the table owner, Database Owner, or System Administrator must **set identity_insert** *table_name* **on** for the column's base table, not through the view through which it is being inserted.

group by Clauses and Views

- When creating a view for security reasons, be careful when using aggregate functions and the **group by** clause. A Transact-SQL extension allows you to name columns that do not appear in the **group by** clause. If you name a column that is not in the **group by** clause, Adaptive Server returns detailed data rows for the column. For example, this query:

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

  returns a row for every (18 rows)—more data than you might intend. While this query:

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

  returns one row for each type (6 rows).

  For more information about **group by**, see "group by and having Clauses."

distinct Clauses and Views

- The **distinct** clause defines a view as a database object that contains no duplicate rows. A row is defined to be a duplicate of another row if all of its column values match the same column values in another row. Null values are considered to be duplicates of other null values.

Querying a subset of a view's columns can result in what appear to be duplicate rows. If you select a subset of columns, some of which contain the same values, the results appear to contain duplicate rows. However, the underlying rows in the view are still unique. Adaptive Server applies the **distinct** requirement to the view's definition when it accesses the view for the first time (before it does any projection and selection) so that all the view's rows are distinct from each other.

You can specify **distinct** more than once in the view definition's **select** statement to eliminate duplicate rows, as part of an aggregate function or a **group by** clause. For example:

```
select distinct count(distinct title_id), price
from titles
```

- The scope of the **distinct** applies only for that view; it does not cover any new views derived from the **distinct** view.

with check option Clauses and Views

- If a view is created **with check option**, each row that is inserted or updated through the view must meet the selection criteria of the view.
- If a view is created **with check option**, all views derived from the "base" view must satisfy its check option. Each row inserted or updated through the derived view must remain visible through the base view.

Getting Information About Views

- To get a report of the tables or views on which a view depends, and of objects that depend on a view, execute the system procedure **sp_depends**.
- To display the text of a view, which is stored in *syscomments*, execute the system procedure **sp_helptext** with the view name as the parameter.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The use of more than one **distinct** keyword and the use of "*column_heading* = *column_name*" in the **select** list are Transact-SQL extensions.

Permissions

create view permission defaults to the Database Owner, who can transfer it to other users.

Permissions on Objects: View Creation Time

When you create a view, Adaptive Server makes no permission checks on objects, such as tables and views, that are referenced by the view. Therefore, you can create a view successfully even if you do not have access to its objects. All permission checks occur when a user invokes the view.

Permissions on Objects: View Execution Time

When a view is invoked, permission checks on its objects depend on whether the view and all referenced objects are owned by the same user.

- If the view and its objects are not owned by the same user, the invoker must have been granted direct access to the objects. For example, if the view performs a select from a table the invoker cannot access, the select statement fails.
- If the view and its objects are owned by the same user, special rules apply. The invoker automatically has implicit permission to access the view's objects even though the invoker could not access them directly. Without having to grant users direct access to your tables, you can give them restricted access with a view. In this way, a view can be a security mechanism. For example, invokers of the view might be able to access only certain rows and columns of your table. A detailed description of the rules for implicit permissions is discussed in Chapter 5, "Managing User Permissions," in the *Security Administration Guide*.

See Also

Commands	**create schema**, **drop view**, **update**
System procedures	**sp_depends**, **sp_help**, **sp_helptext**, **sp_rename**

dbcc

Function

Database Consistency Checker (**dbcc**) checks the logical and physical consistency of a database. Use **dbcc** regularly as a periodic check or if you suspect any damage.

Syntax

```
dbcc checkalloc [(database_name [, fix | nofix])]

dbcc checkcatalog [(database_name)]

dbcc checkdb [(database_name [, skip_ncindex])]

dbcc checkstorage [(database_name)]

dbcc checktable({table_name|table_id}[, skip_ncindex])

dbcc dbrepair (database_name, dropdb)

dbcc fix_text ({table_name | table_id})

dbcc indexalloc ({table_name | table_id}, index_id
       [, {full | optimized | fast | null}
       [, fix | nofix]])

dbcc reindex ({table_name | table_id})

dbcc tablealloc ({table_name | table_id}
       [, {full | optimized | fast | null}
       [, fix | nofix]])|

dbcc { traceon | traceoff } (3604,302)

dbcc tune ( { ascinserts, {0 | 1 } , tablename |
               cleanup, {0 | 1 } |
               cpuaffinity, start_cpu [, on| off] |
               deviochar vdevno, "batch_size" |
               doneinproc { 0 | 1 } |
               maxwritedes, writes_per_batch } )
```

Keywords and Options

checkalloc – checks the specified database to see that all pages are correctly allocated and that no page that is allocated is not used. If no database name is given, **checkalloc** checks the current database. It always uses the **optimized** report option (see **tablealloc**).

checkalloc reports on the amount of space allocated and used.

database_name – is the name of the database to check. If no database name is given, **dbcc** uses the current database.

fix | **nofix** – determines whether **dbcc** fixes the allocation errors found. The default mode for **checkalloc** is **nofix**. You must put the database into single-user mode in order to use the **fix** option.

For a discussion of page allocation in Adaptive Server, see Chapter 4, “Diagnosing System Problems,” in the *System Administration Guide*.

checkcatalog – checks for consistency in and between system tables. For example, it makes sure that every type in *syscolumns* has a matching entry in *systypes*, that every table and view in *sysobjects* has at least one column in *syscolumns*, and that the last checkpoint in *syslogs* is valid. **checkcatalog** also reports on any segments that have been defined. If no database name is given, **checkcatalog** checks the current database.

checkdb – runs the same checks as **checktable**, but on each table, including *syslogs*, in the specified database. If no database name is given, **checkdb** checks the current database.

skip_ncindex – causes **dbcc checktable** or **dbcc checkdb** to skip checking the nonclustered indexes on user tables. The default is to check all indexes.

checkstorage – checks the specified database for allocation, OAM page entries, page consistency, text valued columns, allocation of text valued columns, and text column chains. The results of each **dbcc checkstorage** operation are stored in the *dbccdb* database. For details on using **dbcc checkstorage**, and on creating, maintaining, and generating reports from *dbccdb*, see the *System Administration Guide*.

checktable – checks the specified table to see that index and data pages are correctly linked, that indexes are in properly sorted order, that all pointers are consistent, that the data information on each page is reasonable, and that page offsets are reasonable. If the log segment is on its own device, running **dbcc checktable** on the *syslogs* table reports the log(s) used and free space. For example:

```
Checking syslogs
The total number of data pages in this table is 1.
*** NOTICE: Space used on the log segment is 0.20 Mbytes, 0.13%.
*** NOTICE: Space free on the log segment is 153.4 Mbytes, 99.87%.
DBCC execution completed.  If dbcc printed error messages, see your
System Administrator.
```

If the log segment is not on its own device, the following message appears:

```
*** NOTICE:  Notification of log space used/free cannot be
reported because the log segment is not on its own device.
```

table_name | *table_id* – is the name or object ID of the table to check.

dbrepair (*database_name*, **dropdb**) – drops a damaged database. The **drop database** command does not work on a damaged database.

Users cannot be using the database being dropped when this **dbcc** statement is issued (including the user issuing the statement).

fix_text – upgrades *text* values after an Adaptive Server's character set has been changed from any character set to a new multibyte character set.

Changing to a multibyte character set makes the internal management of *text* data more complicated. Since a *text* value can be large enough to cover several pages, Adaptive Server must be able to handle characters that span page boundaries. To do so, the server requires additional information on each of the *text* pages. The System Administrator or table owner must run **dbcc fix_text** on each table that has *text* data to calculate the new values needed. See the *System Administration Guide* for more information.

indexalloc – checks the specified index to see that all pages are correctly allocated and that no page that is allocated is not used. This is a smaller version of **checkalloc**, providing the same integrity checks on an individual index.

indexalloc produces the same three types of reports as **tablealloc**: **full**, **optimized**, and **fast**. If no type is indicated, or if you use **null**, Adaptive Server uses **optimized**. The **fix|nofix** option functions the same with **indexalloc** as with **tablealloc**.

➤ ***Note***

You can specify **fix** or **nofix** only if you include a value for the type of report (**full**, **optimized**, **fast**, or **null**).

table_name | *table_id*, *index_id* – is the table name or the table's object ID (the *id* column from *sysobjects)* plus the index's *indid* from *sysindexes*.

full – reports all types of allocation errors.

optimized – produces a report based on the allocation pages listed in the object allocation map (OAM) pages for the index. It does not report and cannot fix unreferenced extents on allocation pages that are not listed in the OAM pages. The **optimized** option is the default.

fast – does not produce an allocation report, but produces an exception report of pages that are referenced but not allocated in the extent (2521-level errors).

fix | **nofix** – determines whether **indexalloc** fixes the allocation errors found in the table. The default is **fix** for all indexes except indexes on system tables, for which the default is **nofix**. To use the **fix** option with system tables, you must first put the database in single-user mode.

You can specify **fix** or **nofix** only if you include a value for the type of report (**full**, **optimized**, **fast**, or **null**).

reindex – checks the integrity of indexes on user tables by running a fast version of **dbcc checktable**. It can be used with the table name or the table's object ID (the *id* column from *sysobjects)*. **reindex** prints a message when it discovers the first index-related error and then drops and re-creates the suspect indexes. The System Administrator or table owner must run **dbcc reindex** after Adaptive Server's sort order has been changed and indexes have been marked "suspect" by Adaptive Server.

When **dbcc** finds corrupt indexes, it drops and re-creates the appropriate indexes. If the indexes for a table are already correct, or if the table has no indexes, **dbcc reindex** does not rebuild the index, but prints an informational message instead.

dbcc reindex aborts if a table is suspected of containing corrupt data. When that happens, an error message instructs the user to run **dbcc checktable**. **dbcc reindex** does not allow reindexing of system tables. System indexes are checked and rebuilt, if necessary, as an automatic part of recovery after Adaptive Server is restarted following a sort order change.

tablealloc – checks the specified table to see that all pages are correctly allocated and that no page that is allocated is not used. This is a smaller version of **checkalloc**, providing the same integrity checks on an individual table. It can be used with the table name or the table's object ID (the *id* column from *sysobjects)*. For an example of

tablealloc output, see Chapter 4, "Diagnosing System Problems," in the *System Administration Guide*.

Three types of reports can be generated with **tablealloc**: **full**, **optimized**, and **fast**. If no type is indicated, or if you use **null**, Adaptive Server uses **optimized**.

full – is equivalent to **checkalloc** at a table level; it reports all types of allocation errors.

optimized – produces a report based on the allocation pages listed in the object allocation map (OAM) pages for the table. It does not report and cannot fix unreferenced extents on allocation pages that are not listed in the OAM pages. The **optimized** option is the default.

fast – does not produce an allocation report, but produces an exception report of pages that are referenced but not allocated in the extent (2521-level errors).

fix | **nofix** – determines whether or not **tablealloc** fixes the allocation errors found in the table. The default is **fix** for all tables except system tables, for which the default is **nofix**. To use the **fix** option with system tables, you must first put the database in single user mode.

You can specify **fix** or **nofix** only if you include a value for the type of report (**full**, **optimized**, **fast**, or **null**).

traceon | traceoff – toggles the printing of optimizer diagnostics during index selection. For more information, see Chapter 10, "Advanced Optimizing Techniques," in the *Performance and Tuning Guide*.

tune – enables or disables tuning flags for special performance auscultations. See the *Performance and Tuning Guide* for more information on the individual options.

Examples

1. **`dbcc checkalloc(pubs2)`**

   Checks *pubs2* for page allocation errors.

2. **`dbcc checkstorage(pubs2)`**

   Checks database consistency for *pubs2* and places the information in the *dbccdb* database.

3. **`dbcc tablealloc(publishers, null, nofix)`**

Adaptive Server returns an optimized report of allocation for this table, but does not fix any allocation errors.

4. **`dbcc checktable(salesdetail)`**

```
Checking salesdetail
The total number of pages in partition 1 is 3.
The total number of pages in partition 2 is 1.
The total number of pages in partition 3 is 1.
The total number of pages in partition 4 is 1.
The total number of data pages in this table is 10.
Table has 116 data rows.
DBCC execution completed. If DBCC printed error messages,
contact a user with System Administrator (SA) role.
```

5. **`dbcc indexalloc ("pubs..titleauthor", 2, full)`**

   Adaptive Server returns a full report of allocation for the index with an *indid* of 2 on the *titleauthor* table and fixes any allocation errors.

6. **`dbcc dbrepair(pubs2, dropdb)`**

   Drope the damaged database *pubs2*.

7. **`dbcc reindex(titles)`**

   ```
   One or more indexes are corrupt. They will be
        rebuilt.
   ```

   dbcc reindex has discovered one or more corrupt indexes in the *titles* table.

8. **`dbcc fix_text(texttest)`**

   Upgrades text values for *texttest* after a character set change.

Comments

- **dbcc** (Database Consistency Checker) can be run while the database is active, except for the **dbrepair**(*database_name*, **dropdb**) option and **dbcc checkalloc** with the **fix** option.
- **dbcc** locks database objects as it checks them. See the **dbcc** discussion in the *System Administration Guide* for information on minimizing performance problems while using **dbcc.**
- To qualify a table or an index name with a user name or database name, enclose the qualified name in single or double quotation marks. For example:

  ```
  dbcc tablealloc("pubs2.pogo.testtable")
  ```
- **dbcc reindex** cannot be run within a user-defined transaction.

- **dbcc fix_text** can generate a large number of log records, which may fill up the transaction log. **dbcc fix_text** is designed so that updates are done in a series of small transactions: in case of a log space failure, only a small amount of work is lost. If you run out of log space, clear your log and restart **dbcc fix_text** using the same table that was being upgraded when the original **dbcc fix_text** failed.
- If you attempt to use **select**, **readtext**, or **writetext** on *text* values after changing to a multibyte character set, and you have not run **dbcc fix_text**, the command fails, and an error message instructs you to run **dbcc fix_text** on the table. However, you can delete *text* rows after changing character sets without running **dbcc fix_text**.
- **dbcc** output is sent as messages or errors, rather than as result rows. Client programs and scripts should check the appropriate error handlers.
- If a table is partitioned, **dbcc checktable** returns information about each partition.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Only the table owner can execute **dbcc** with the **checktable**, **fix_text**, or **reindex** keywords. Only the Database Owner can use the **checkstorage**, **checkdb**, **checkcatalog**, **checkalloc**, **indexalloc**, and **tablealloc** keywords. Only a System Administrator can use the **dbrepair** keyword.

See Also

Commands	**drop database**
System procedures	**sp_configure**, **sp_helpdb**

deallocate cursor

Function

Makes a cursor inaccessible and releases all memory resources committed to that cursor.

Syntax

```
deallocate cursor cursor_name
```

Parameters

cursor_name – is the name of the cursor to deallocate.

Examples

1. `deallocate cursor authors_crsr`

   Deallocates the cursor named "authors_crsr."

Comments

- Adaptive Server returns an error message if the cursor does not exist.
- You must deallocate a cursor before you can use its cursor name as part of another **declare cursor** statement.
- **deallocate cursor** has no effect on memory resource usage when specified in a stored procedure or trigger.
- You can **deallocate** a cursor whether it is open or closed.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

deallocate cursor permission defaults to all users. No permission is required to use it.

See Also

Commands	**close**, **declare cursor**

declare

Function

Declares the name and type of local variables for a batch or procedure.

Syntax

Variable declaration:

```
declare @variable_name datatype
   [, @variable_name datatype]...
```

Variable assignment:

```
select @variable = {expression | select_statement}
   [, @variable = {expression | select_statement} ...]
   [from table_list]
   [where search_conditions]
   [group by group_by_list]
   [having search_conditions]
   [order by order_by_list]
   [compute function_list [by by_list]]
```

Keywords and Options

@variable_name – must begin with @ and must conform to the rules for identifiers.

datatype – can be either a system datatype or a user-defined datatype.

Examples

1.
```
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"
```

```
you got one
you got two
```

Declares two variables and prints strings according to the values in the variables.

2.
```
declare @veryhigh money
select @veryhigh = max(price)
  from titles
if @veryhigh > $20
  print "Ouch!"
```

Prints “Ouch!” if the maximum book price in the *titles* table is more than $20.00.

Comments

- Assign values to local variables with a **select** statement.
- The maximum number of parameters in a procedure is 255. The number of local or global variables is limited only by available memory. The @ sign denotes a variable name.
- Local variables are often used as counters for **while** loops or **if...else** blocks. In stored procedures, they are declared for automatic, noninteractive use by the procedure when it executes. Local variables must be used in the batch or procedure in which they are declared.
- The **select** statement that assigns a value to the local variable usually returns a single value. If there is more than one value to return, the variable is assigned the last one. The **select** statement that assigns values to variables cannot be used to retrieve data in the same statement.
- The **print** and **raiserror** commands can take local variables as arguments.
- Users cannot create global variables and cannot update the value of global variables directly in a **select** statement.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

declare permission defaults to all users. No permission is required to use it.

See Also

Commands	**print**, **raiserror**, **select**, **while**

declare cursor

Function

Defines a cursor.

Syntax

```
declare cursor_name cursor
   for select_statement
   [for {read only | update [of column_name_list]}]
```

Parameters

cursor_name – is the name of the cursor being defined.

select_statement – is the query that defines the cursor result set. See **select** for more information.

for read only – specifies that the cursor result set cannot be updated.

for update – specifies that the cursor result set is updatable.

of *column_name_list* – is the list of columns from the cursor result set (specified by the *select_statement*) defined as updatable. Adaptive Server also allows you to include columns that are not specified in the list of columns of the cursor's *select_statement* (and excluded from the result set), but that are part of the tables specified in the *select_statement*.

Examples

1.
   ```
   declare authors_crsr cursor
   for select au_id, au_lname, au_fname
   from authors
   where state != 'CA'
   ```

   Defines a result set for the *authors_crsr* cursor that contains all authors from the *authors* table who do not reside in California.

2.
   ```
   declare titles_crsr cursor
   for select title, title_id from titles
   where title_id like "BU%"
   for read only
   ```

   Defines a read-only result set for the *titles_crsr* cursor that contains the business-type books from the *titles* table.

3.
```
declare pubs_crsr cursor
for select pub_name, city, state
from publishers
for update of city, state
```

Defines an updatable result set for the *pubs_crsr* cursor that contains all of the rows from the *publishers* table. It defines the address of each publisher (*city* and *state* columns) for update.

Comments

Restrictions on Cursors

- A **declare cursor** statement must precede any **open** statement for that cursor.
- You cannot include other statements with **declare cursor** in the same Transact-SQL batch.
- *cursor_name* must be a valid Adaptive Server identifier.

Cursor *select* Statements

- *select_statement* can use the full syntax and semantics of a Transact-SQL **select** statement, with these restrictions:
  - *select_statement* must contain a **from** clause.
  - *select_statement* cannot contain a **compute**, **for browse**, or **into** clause.
  - *select_statement* can contain the **holdlock** keyword.
- The *select_statement* can contain references to Transact-SQL parameter names or Transact-SQL local variables (for all cursor types except language). The names must reference the Transact-SQL parameters and local variables defined in the procedure, trigger, or statement batch that contains the **declare cursor** statement.

  The parameters and local variables referenced in the **declare cursor** statement do not have to contain valid values until the cursor is opened.
- The *select_statement* can contain references to the *inserted* and *deleted* temporary tables that are used in triggers.

Scope

- A cursor's existence depends on its **scope**. The scope refers to the context in which the cursor is used, that is, within a user session, within a stored procedure, or within 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 started by other users. 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 stored procedure or trigger scope and to the scope that launched the stored procedure or trigger. Cursors declared inside a trigger on an *inserted* or a *deleted* table are not accessible to any nested stored procedures or triggers. However, cursors declared inside a trigger on an *inserted* or a *deleted* table **are** accessible within the scope of the trigger. Once the stored procedure or trigger completes, Adaptive Server deallocates the cursors created within it.

Figure 1-1 illustrates how cursors operate between scopes.

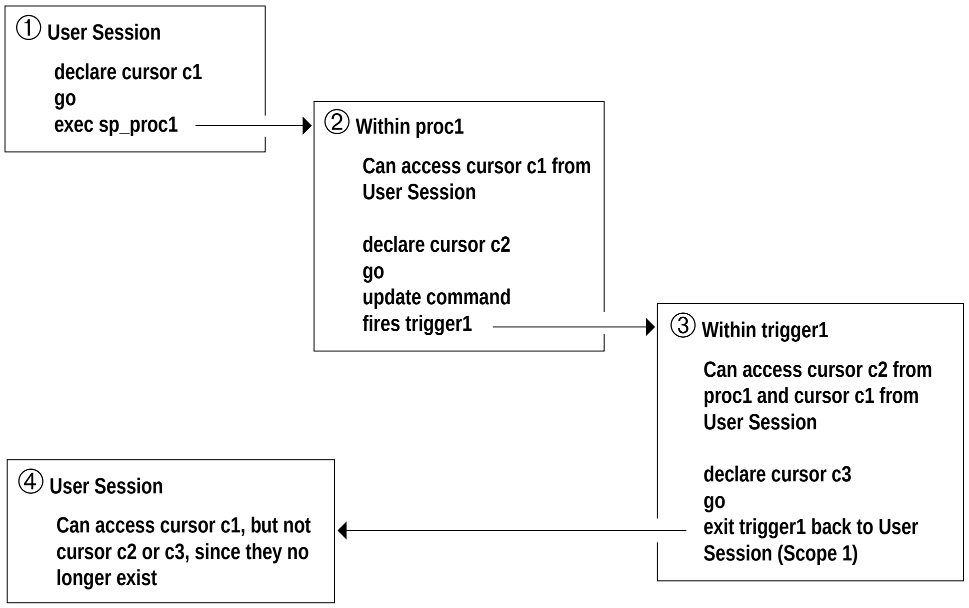

① User Session
declare cursor c1
go
exec sp_proc1
② Within proc1
Can access cursor c1 from User Session
declare cursor c2
go
update command fires trigger1
③ Within trigger1
Can access cursor c2 from proc1 and cursor c1 from User Session
declare cursor c3
go
exit trigger1 back to User Session (Scope 1)
④ User Session
Can access cursor c1, but not cursor c2 or c3, since they no longer exist

Figure 1-1: 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
```

Result Set

- Cursor result set rows may not reflect the values in the actual base table rows. For example, a cursor declared with an **order by** clause usually requires the creation of an internal table to order the rows for the cursor result set. Adaptive Server does not lock the rows in the base table that correspond to the rows in the internal table, which permits other clients to update these base table rows. In that case, the rows returned to the client from the cursor result set would not be in sync with the base table rows.
- A cursor result set is generated as the rows are returned through a **fetch** of that cursor. This means that a cursor **select** query is processed like a normal **select** query. This process, known as a **cursor scan**, provides a faster turnaround time and eliminates the need to read rows that are not required by the application.

  A restriction of cursor scans is that they can only use the unique indexes of a table. However, if none of the base tables referenced by the cursor result set are updated by another process in the same lock space as the cursor, the restriction is unnecessary. Adaptive Server allows the declaration of cursors on tables without unique indexes, but any attempt to update those tables in the same lock space closes all cursors on the tables.

Updatable Cursors

- After defining a cursor using **declare cursor**, Adaptive Server determines whether the cursor is **updatable** or **read-only**. If a cursor is updatable, you can update or delete rows within the cursor result set. If a cursor is read-only, you cannot change the result set.

- Use the **for update** or **for read only** clause to explicitly define a cursor as updatable or read-only. You cannot define an updatable cursor if its *select_statement* contains one of the following constructs:
  - **distinct** option
  - **group by** clause
  - Aggregate function
  - Subquery
  - **union** operator
  - **at isolation read uncommitted** clause

  If you omit either the **for update** or the **read only** clause, Adaptive Server checks to see whether the cursor is updatable.

  Adaptive Server also defines a cursor as read-only if you declare a language- or server-type cursor that includes an **order by** clause as part of its *select_statement*. Adaptive Server handles updates differently for client- and execute-type cursors, thereby eliminating this restriction.
- 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.

  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 none exists, it returns an error.
- In most cases, 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, include its column in the **for update** *column_name_list*, so that Adaptive Server can find it quickly for the cursor scan.

This allows Adaptive Server to use that unique index for its cursor scan, which helps prevent an update anomaly called the **Halloween problem**. Another way to prevent the Halloween problem is to create tables with the **unique auto_identity index** database option. See the **unique auto_identity index** database option description in the *System Administration Guide* for more information.

This problem occurs when a client updates a column of a cursor result set row that defines the order in which the rows are returned from the base tables. For example, if 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. This is a result of an updatable cursor only logically creating a cursor result set. The cursor result set is actually the base tables that derive the cursor.

- If you specify the **read only** option, the cursor result set cannot be updated using the **delete** or **update** statement.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The **for update** and **for read only** options are Transact-SQL extensions.

Permissions

declare cursor permission defaults to all users. No permission is required to use it.

See Also

Commands	**open**

delete

Function

Removes rows from a table.

Syntax

```
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
```

Keywords and Options

from – (after **delete**) is an optional keyword used for compatibility with other versions of SQL.

view_name | *table_name* – is the name of the view or table from which to remove rows. Specify the database name if the view or table is in another database, and specify the owner's name if more than one view or table of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

where – is a standard **where** clause. See "where Clause" for more information.

from – (after *table_name* or *view_name*) lets you name more than one table or view to use with a **where** clause when specifying which rows to delete. This **from** clause allows you to delete rows from one table based on data stored in other tables, giving you much of the power of an embedded **select** statement.

index *index_name* – specifies an index to use for accessing *table_name*. You cannot use this option when you delete from a view.

prefetch *size* – specifies the I/O size, in kilobytes, for tables that are bound to caches with large I/Os configured. Valid values for size are 2, 4, 8, and 16. You cannot use this option when you delete from a view. The procedure **sp_helpcache** shows the valid sizes for the cache an object is bound to or for the default cache.

If Component Integration Services is enabled, you cannot use the **prefetch** keyword for remote servers.

lru | mru – specifies the buffer replacement strategy to use for the table. Use **lru** to force the optimizer to read the table into the cache on the MRU/LRU (most recently used/least recently used) chain. Use **mru** to discard the buffer from cache, and replace it with the next buffer for the table. You cannot use this option when you delete from a view.

where current of *cursor_name* – causes Adaptive Server to delete the row of the table or view indicated by the current cursor position for *cursor_name*.

Examples

1.
```
delete authors
```
Deletes all rows from the *authors* table.

2.
```
delete from authors
where au_lname = "McBadden"
```
Deletes a row or rows from the *authors* table.

3.
```
delete titles
from titles, authors, titleauthor
where authors.au_lname = 'Bennet'
  and authors.au_id = titleauthor.au_id
  and titleauthor.title_id = titles.title_id
```
Deletes rows for books written by Bennet from the *titles* table. (The *pubs2* database includes a trigger (*deltitle*) that prevents the deletion of the titles recorded in the *sales* table; drop this trigger for this example to work.)

4.
```
delete titles where current of title_crsr
```
Deletes a row from the *titles* table currently indicated by the cursor *title_crsr.*

5.
```
delete authors
where syb_identity = 4
```

Determines which row has a value of 4 for the IDENTITY column and deletes it from the *authors* table. Note the use of the **syb_identity** keyword instead of the actual name of the IDENTITY column.

Comments

- **delete** removes rows from the specified table.
- You can refer to up to 15 tables in a **delete** statement.

Restrictions

- You cannot use **delete** with a multitable view (one whose **from** clause names more than one table), even though you may be able to use **update** or **insert** on that same view. Deleting a row through a multitable view would change multiple tables, which is not permitted. **insert** and **update** statements that affect only one base table of the view are permitted.
- Adaptive Server treats two different designations for the same table in a **delete** as two tables. For example, the following **delete** issued in *pubs2* specifies *discounts* as two tables (*discounts* and *pubs2..discounts)*:

```
delete discounts
from pubs2..discounts, pubs2..stores
where pubs2..discounts.stor_id =
    pubs2..stores.stor_id
```

  In this case, the join does not include *discounts*, so the **where** condition remains true for every row; Adaptive Server deletes all rows in *discounts* (which is not the desired result). To avoid this problem, use the same designation for a table throughout the statement.
- If you are deleting a row from a table that is referenced from other tables via referential constraints, Adaptive Server checks all the referencing tables before permitting the delete. If the row you are attempting to delete contains a primary key that is being used as a foreign key by one of the referencing tables, the delete is not allowed.

Deleting All Rows from a Table

- If you do not use a **where** clause, **all** rows in the table named after **delete** [**from**] are removed. The table, though empty of data, continues to exist until you issue a **drop table** command.
- **truncate table** and **delete** without a row specification are functionally equivalent, but **truncate table** is faster. **delete** removes rows one at a time and logs these transactions. **truncate table** removes whole data pages, and the rows are not logged.

  Both **delete** and **truncate table** reclaim the space occupied by the data and its associated indexes.
- You cannot use the **truncate table** command on a partitioned table. To remove all rows from a partitioned table, either use the **delete** command without a **where** clause or unpartition the table before issuing the **truncate table** command.

delete and Transactions

- In chained transaction mode, each **delete** statement implicitly begins a new transaction if no transaction is currently active. Use **commit** to complete any deletes, or use **rollback** to undo the changes. For example:

  ```
  delete from sales where date < '01/01/89'
  if exists (select stor_id
      from stores
      where stor_id not in
      (select stor_id from sales))
          rollback transaction
  else
          commit transaction
  ```

  This batch begins a transaction (using the chained transaction mode) and deletes rows with dates earlier than Jan. 1, 1989 from the *sales* table. If it deletes all sales entries associated with a store, it rolls back all the changes to *sales* and ends the transaction. Otherwise, it commits the deletions and ends the transaction. For more information about the chained mode, see Chapter 18, “Transactions: Maintaining Data Consistency and Recovery,” in the *Transact-SQL User’s Guide*.

Delete Triggers

- You can define a trigger that will take a specified action when a **delete** command is issued on a specified table.

Using *delete where current of*

- Use the clause **where current of** with cursors. Before deleting rows using the clause **where current of**, you must first define the cursor with **declare cursor** and open it using the **open** statement. Position the cursor on the row you want to delete using one or more **fetch** statements. The cursor name cannot be a Transact-SQL parameter or local variable. The cursor must be an updatable cursor or Adaptive Server returns an error. Any deletion to the cursor result set also affects the base table row from which the cursor row is derived. You can delete only one row at a time using the cursor.
- You cannot delete rows in a cursor result set if the cursor's **select** statement contains a join clause, even though the cursor is considered updatable. 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.
- After the deletion of a row from the cursor's result set, the cursor is positioned before the next row in the cursor's result set. You must issue a **fetch** to access the next row. If the deleted row is the last row of the cursor result set, the cursor is positioned after the last row of the result set. The following describes the position and behavior of open cursors affected by a **delete**:
  - If a client deletes a row (using another cursor or a regular **delete**) and that row represents the current cursor position of other opened cursors owned by the same client, the position of each affected cursor is implicitly set to precede the next available row. However, it is not possible for one client to delete a row representing the current cursor position of another client's cursor.
  - If a client deletes a row that represents the current cursor position of another cursor defined by a join operation and owned by the same client, Adaptive Server accepts the **delete** statement. However, it implicitly closes the cursor defined by the join.

Using *index*, *prefetch*, or *lru | mru*

- The **index**, **prefetch**, and **lru | mru** options override the choices made by the Adaptive Server optimizer. Use these options with caution, and always check the performance impact with **set statistics io on**. See the *Performance and Tuning Guide* for more information about using these options.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The use of more than one table in the **from** clause and qualification of table name with database name are Transact-SQL extensions.

Permissions

delete permission defaults to the table or view owner, who can transfer it to other users.

If you have **set ansi_permissions on**, you must have **select** permission on all columns appearing in the **where** clause, in addition to the regular permissions required for **delete** statements. By default, **ansi_permissions** is **off**.

See Also

Commands	**create trigger**, **drop table**, **drop trigger**, **truncate table**, **where Clause**

disk init

Function

Makes a physical device or file usable by Adaptive Server.

Syntax

```
disk init
   1name = "device_name" ,
   physname = "physicalname" ,
   vdevno = virtual_device_number ,
   size = number_of_blocks
   [, vstart = virtual_address ,
   cntrltype = controller_number ]
   [, contiguous]
```

Keywords and Options

name – is the name of the database device or file. The name must conform to the rules for identifiers and must be enclosed in single or double quotes. This name is used in the **create database** and **alter database** commands.

physname – is the full specification of the database device. This name must be enclosed in single or double quotes.

vdevno – is the virtual device number. It must be unique among the database devices associated with Adaptive Server. The device number 0 is reserved for the master device. Valid device numbers are between 1 and 255, but the highest number must be one less than the number of database devices for which your Adaptive Server is configured. For example, for an Adaptive Server with the default configuration of 10 devices, the available device numbers are 1–9. To see the maximum number of devices available on Adaptive Server, run **sp_configure**, and check the **number of devices** value.

To determine the virtual device number, look at the *device_number* column of the **sp_helpdevice** report, and use the next unused integer.

If you drop a device with **sp_dropdevice**, you cannot reuse its **vdevno** until the server is rebooted.

size – is the size of the database device in 2K blocks.

If you plan to use the new device for the creation of a new database, the minimum **size** is the size of the *model* database, 1024 2K blocks (2MB). If you are initializing a log device, the **size** can be as small as 512 2K blocks (1MB). The maximum size is system-dependent.

➤ ***Note***

The **disk init** command fails if the number of 2K blocks on the physical device is less than the sum of **size** and **vstart**.

vstart – is the starting virtual address, or the starting offset, in 2K blocks. The value for **vstart** should be 0 (the default) unless you are running the Logical Volume Manager on an AIX operating system, in which case, *vstart* should be 2.

Specify **vstart** only if instructed to do so by Sybase Technical Support.

cntrltype – specifies the disk controller. Its default value is 0. Reset **cntrltype** only if instructed to do so by Sybase Technical Support.

contiguous – (OpenVMS only) forces contiguous database file creation. This option is meaningful only when you are initializing a **file**; it has no effect when initializing a **foreign device**. If you include the **contiguous** option, the system creates a contiguous file or the command fails with an error message. If you do not include the **contiguous** option, the system still tries to create a contiguous file. If the system fails to create the file contiguously, it creates a file that does not force contiguity. In either case, the system displays a message indicating the type of file that is created.

Examples

1.
```
disk init
  name = "user_disk",
  physname = "/dev/rxy1a",
  vdevno = 2, size = 5120
```

Initializes 5MB of a disk on a UNIX system.

2.
```
disk init
 name = "user_disk",
 physname = "disk$rose_1:[dbs]user.dbs",
 vdevno = 2, size = 5120,
 contiguous
```

Initializes 5MB of a disk on an OpenVMS system, forcing the database file to be created contiguously.

Comments

- The master device is initialized by the installation program; it is not necessary to initialize this device with **disk init**.
- To successfully complete disk initialization, the "sybase" user must have the appropriate operating system permissions on the device that is being initialized.
- Use **disk init** for each new database device. Each time **disk init** is issued, a row is added to *master..sysdevices*. A new database device does not automatically become part of the pool of default database storage. Assign default status to a database device with the system procedure **sp_diskdefault**.
- On OpenVMS systems, using a logical name to refer to the **physname** offers more flexibility than using a hard-coded path name. For example, if you define the logical name ‘‘userdisk’’ as:

  ```
  disk$rose_1:[dbs]user.dbs
  ```

  you can change the **physname** in the example 2 above to "userdisk". To reorganize your disk or to move ‘‘user.dbs’’, just redefine the logical name as the new path.

  Any logical name used by an Adaptive Server must be:

  - A system logical name, or
  - A process logical name defined in the runserver file for that Adaptive Server.
- Back up the *master* database with the **dump database** or **dump transaction** command after each use of **disk init**. This makes recovery easier and safer in case *master* is damaged. (If you add a device with **disk init** and fail to back up *master*, you may be able to recover the changes by using **disk reinit** and then stopping and restarting Adaptive Server.)
- User databases are assigned to database devices with the optional clause:

  on ***device_name***

  of the **create database** or **alter database** command.
- The preferred method for placing a database's transaction log (that is, the system table *syslogs*) on a different device than the one on which the rest of the database is stored, is the **log on** extension to **create database**. Alternatively, you can name at least two devices

when you create the database, and then execute the system procedure **sp_logdevice**. You can also use **alter database** to extend the database onto a second device and then run **sp_logdevice**.

➤ ***Note***

The **log on** extension immediately moves the entire log to a separate device. The **sp_logdevice** method retains part of the system log on the original database device until transaction activity causes the migration to become complete.

- For a report on all Adaptive Server devices on your system (both database and dump devices), execute the system procedure **sp_helpdevice**.
- Remove a database device with the system procedure **sp_dropdevice**. You must first drop all existing databases on that device.

  After dropping a database device, you can create a new one with the same name (using **disk init**), as long as you give it a different physical name and virtual device number. If you want to use the same physical name and virtual device number, you must restart Adaptive Server.
- If **disk init** failed because the **size** value is too large for the database device, use a different virtual device number or restart Adaptive Server before executing **disk init** again.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

disk init permission defaults to System Administrators and is not transferable. You must be using the *master* database to use **disk init**.

See Also

Commands	**alter database**, **create database**, **disk refit**, **disk reinit**, **dump database**, **dump transaction**, **load database**, **load transaction**
System procedures	**sp_diskdefault**, **sp_dropdevice**, **sp_helpdevice**, **sp_logdevice**

disk mirror

Function

Creates a software mirror that immediately takes over when the primary device fails.

Syntax

```
disk mirror
   name = "device_name" ,
   mirror = "physicalname"
   [ ,writes = { serial | noserial }]
   [ ,contiguous ] (OpenVMS only)
```

Keywords and Options

name – is the name of the database device that you want to mirror. This is recorded in the *name* column of the *sysdevices* table. The name must be enclosed in single or double quotes.

mirror – is the full path name of the database mirror device that is to be your secondary device. It must be enclosed in single or double quotes. If the secondary device is a file, *physicalname* should be a path specification that clearly identifies the file, which Adaptive Server will create. It cannot be an existing file.

writes – allows you to choose whether to enforce serial writes to the devices. In the default case (**serial**), the write to the primary database device is guaranteed to finish before the write to the secondary device begins. If the primary and secondary devices are on different physical devices, serial writes can ensure that at least one of the disks will be unaffected in the event of a power failure.

contiguous – (OpenVMS only) is meaningful only if the mirror is a file rather than a foreign device. This option forces the file that will be used as the secondary device to be created contiguously. If you include the **contiguous** option, the system creates a contiguous file or the command fails with an error message. If you do not include the **contiguous** option, the system still tries to create a contiguous file. If it fails to create the file contiguously, the system creates a file that does not force contiguity. In either case, the system displays a message indicating the type of file that is created. The **contiguous** option is also available with **disk init** for OpenVMS users.

Examples

```
1. disk mirror
     name = "user_disk",
     mirror = "/server/data/mirror.dat"
```

Creates a software mirror for the database device *user_disk* on the file *mirror.dat*.

Comments

- Disk mirroring creates a software mirror of a user database device, the master database device, or a database device used for user database transaction logs. If a database device fails, its mirror immediately takes over.

  Disk mirroring does not interfere with ongoing activities in the database. You can mirror or unmirror database devices without shutting down SQL Server.
- Back up the *master* database with the **dump database** command after each use of **disk mirror.** This makes recovery easier and safer in case *master* is damaged.
- When a read or write to a mirrored device is unsuccessful, Adaptive Server unmirrors the bad device and prints error messages. Adaptive Server continues to run, unmirrored. The System Administrator must use the **disk remirror** command to restart mirroring.
- You can mirror the master device, devices that store data, and devices that store transaction logs. However, you cannot mirror dump devices.
- Devices are mirrored; databases are not.
- A device and its mirror constitute one logical device. Adaptive Server stores the physical name of the mirror device in the *mirrorname* column of the *sysdevices* table. It does not require a separate entry in *sysdevices* and should not be initialized with **disk init.**
- To retain use of asynchronous I/O, always mirror devices that are capable of asynchronous I/O to other devices capable of asynchronous I/O. In most cases, this means mirroring raw devices to raw devices and operating system files to operating system files.

  If the operating system cannot perform asynchronous I/O on files, mirroring a raw device to a regular file produces an error message. Mirroring a regular file to a raw device will work, but will not use asynchronous I/O.

- Mirror all default database devices so that you are still protected if a **create** or **alter database** command affects a database device in the default list.
- For greater protection, mirror the database device used for transaction logs.
- Always put user database transaction logs on a separate database device. To put a database's transaction log (that is, the system table *syslogs*) on a different device than the one on which the rest of the database is stored, name the database device and the log device when you create the database. You could also use **alter database** to extend the database onto a second device and then run the system procedure **sp_logdevice**.
- If you mirror the database device for the *master* database, you can use the **-r** option and the name of the mirror for UNIX, or the **mastermirror** option for OpenVMS, when you restart Adaptive Server with the **dataserver** utility program. Add this to the *RUN_servername* file for that server so that the **startserver** utility program knows about it. For example:

  ```
  dataserver -dmaster.dat -rmirror.dat
  ```

  starts a master device named *master.dat* and its mirror, *mirror.dat*. For more information, see **dataserver** and **startserver** in the *Utility Programs* manual for your platform.
- If you mirror a database device that has unallocated space (room for additional **create database** and **alter database** statements to allocate part of the device), **disk mirror** begins mirroring these allocations when they are made, not when the **disk mirror** command is issued.
- For a report on all Adaptive Server devices on your system (user database devices and their mirrors, as well as dump devices), execute the system procedure **sp_helpdevice**.
- For more details about disk mirroring in the OpenVMS environment, see the the configuration documentation for your platform.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

disk mirror permission defaults to the System Administrator and is not transferable. You must be using the *master* database in order to use **disk mirror.**

See Also

Commands	**alter database**, **create database**, **disk init**, **disk refit**, **disk reinit**, **disk remirror**, **disk unmirror**, **dump database**, **dump transaction**, **load database**, **load transaction**
System procedures	**sp_diskdefault**, **sp_helpdevice**, **sp_logdevice**
Utility programs	**dataserver**, **startserver**

disk refit

Function

Rebuilds the *master* database's *sysusages* and *sysdatabases* system tables from information contained in *sysdevices*.

Syntax

```
disk refit
```

Examples

```
1. disk refit
```

Comments

- Adaptive Server automatically shuts down after **disk refit** rebuilds the system tables.
- Use **disk refit** after **disk reinit** as part of the procedure to restore the master database. For more information, see Chapter 22, "Restoring the System Databases," in the *System Administration Guide*.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

disk refit permission defaults to System Administrators and is not transferable. You must be in the *master* database to use **disk refit**.

See Also

Commands	**disk init**, **disk reinit**
System procedures	**sp_addumpdevice**, **sp_helpdevice**

disk reinit

Function

Rebuilds the *master* database's *sysdevices* system table. Use **disk reinit** as part of the procedure to restore the *master* database.

Syntax

```
disk reinit
   name = "device_name",
   physname = "physicalname" ,
   vdevno = virtual_device_number ,
   size = number_of_blocks
   [, vstart = virtual_address ,
   cntrltype = controller_number]
```

Keywords and Options

name – is the name of the database device. It must conform to the rules for identifiers, and it must be enclosed in single or double quotes. This name is used in the **create database** and **alter database** commands.

physname – is the name of the database device. The physical name must be enclosed in single or double quotes.

vdevno – is the virtual device number. It must be unique among devices used by Adaptive Server. The device number 0 is reserved for the *master* database device. Legal numbers are between 1 and 255, but cannot be greater than the number of database devices for which your system is configured. The default is 50 devices.

size – is the size of the database device in 2K blocks. The minimum usable size is 1024 2K blocks (2MB).

vstart – is the starting virtual address, or the starting offset, in 2K blocks. The value for **vstart** should be 0 (the default) unless you are running the Logical Volume Manager on an AIX operating system, in which case, *vstart* should be 2.

Specify **vstart** only if instructed to do so by Sybase Technical Support.

cntrltype – specifies the disk controller. Its default value is 0. Reset it only if instructed to do so by Sybase Technical Support.

Examples

```
1. disk reinit
     name = "user_disk",
     physname = "/server/data/userdata.dat",
     vdevno = 2, size = 5120
```

Comments

- **disk reinit** ensures that *master..sysdevices* is correct if the master database has been damaged or if devices have been added since the last dump of *master*.
- **disk reinit** is similar to **disk init**, but does not initialize the database device.
- For complete information on restoring the *master* database, see Chapter 22, “Restoring the System Databases,” in the *System Administration Guide*.

Standards and Compliance

Standard	Compliance level
SQL92	Transact-SQL extension

Permissions

disk reinit permission defaults to System Administrators and is not transferable. You must be in the *master* database to use **disk reinit**.

See Also

Commands	**alter database**, **create database**, **dbcc**, **disk init**, **disk refit**
System procedures	**sp_addumpdevice**, **sp_helpdevice**

disk remirror

Function

Restarts disk mirroring after it is stopped by failure of a mirrored device or temporarily disabled by the **disk unmirror** command.

Syntax

```
disk remirror
   name = "device_name"
```

Keywords and Options

name – is the name of the database device that you want to remirror. This is recorded in the *name* column of the *sysdevices* table. The name must be enclosed in single or double quotes.

Examples

```
1. disk remirror
     name = "user_disk"
```

Resumes software mirroring on the database device *user_disk*.

Comments

- Disk mirroring creates a software mirror of a user database device, the master database device, or a database device used for user database transaction logs. If a database device fails, its mirror immediately takes over.

  Use the **disk remirror** command to reestablish mirroring after it has been temporarily stopped by failure of a mirrored device or temporarily disabled with the **mode = retain** option of the **disk unmirror** command. The **disk remirror** command copies data on the retained disk to the mirror.
- It is important to back up the *master* database with the **dump database** command after each use of **disk remirror**. This makes recovery easier and safer in case *master* is damaged.
- If mirroring was permanently disabled with the **mode = remove** option, you must remove the operating system file that contains the mirror before using **disk remirror.**
- Database devices, not databases, are mirrored.

- You can mirror, remirror, or unmirror database devices without shutting down Adaptive Server. Disk mirroring does not interfere with ongoing activities in the database.
- When a read or write to a mirrored device is unsuccessful, Adaptive Server unmirrors the bad device and prints error messages. Adaptive Server continues to run, unmirrored. The System Administrator must use **disk remirror** to restart mirroring.
- In addition to mirroring user database devices, always put user database transaction logs on a separate database device. The database device used for transaction logs can also be mirrored for even greater protection. To put a database's transaction log (that is, the system table *syslogs*) on a different device than the one on which the rest of the database is stored, name the database device and the log device when you create the database. You could also **alter database** to a second device and then run the system procedure **sp_logdevice**.
- If you mirror the database device for the *master* database, you can use the **-r** option and the name of the mirror for UNIX, or the **mastermirror** option for OpenVMS, when you restart Adaptive Server with the **dataserver** utility program. Add this option to the *RUN_servername* file for that server so that the **startserver** utility program knows about it. For example:

  ```
  dataserver -dmaster.dat -rmirror.dat
  ```

  starts a master device named *master.dat* and its mirror, *mirror.dat*. For more information, see **dataserver** and **startserver** in the *Utility Programs* manual for your platform.
- For a report on all Adaptive Server devices on your system (user database devices and their mirrors, as well as dump devices), execute the system procedure **sp_helpdevice**.
- For more details about disk mirroring in the OpenVMS environment, see your the configuration documentation for your platform.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

disk remirror permission defaults to the System Administrator and is not transferable. You must be using the *master* database to use **disk remirror.**

See Also

Commands	**alter database**, **create database**, **disk init**, **disk mirror**, **disk refit**, **disk reinit**, **disk unmirror**, **dump database**, **dump transaction**, **load database**, **load transaction**
System procedures	**sp_diskdefault**, **sp_helpdevice**, **sp_logdevice**
Utility programs	**dataserver**, **startserver**

disk unmirror

Function

Suspends disk mirroring initiated with the **disk mirror** command to allow hardware maintenance or the changing of a hardware device.

Syntax

```
disk unmirror
   name = "device_name"
   [ ,side = { "primary" | secondary }]
   [ ,mode = { retain | remove }]
```

Keywords and Options

name – is the name of the database device that you want to unmirror. The name must be enclosed in single or double quotes.

side – specifies whether to disable the **primary** device or the **secondary** device (the mirror). By default, the secondary device is unmirrored.

mode – determines whether the unmirroring is temporary (**retain**) or permanent (**remove**). By default, unmirroring is temporary.

Specify **retain** when you plan to remirror the database device later in the same configuration. This option mimics what happens when the primary device fails:

- I/O is directed only at the device **not** being unmirrored
- The *status* column of *sysdevices* indicates that mirroring is deactivated

remove eliminates all *sysdevices* references to a mirror device:

- The *status* column indicates that the mirroring feature is ignored
- The *phyname* column is replaced by the name of the secondary device in the *mirrorname* column if the primary device is the one being deactivated
- The *mirrorname* column is set to NULL

Examples

1. **`disk unmirror`**
   `name = "user_disk"`

   Suspends software mirroring for the database device *user_disk*.

2. **`disk unmirror name = "user_disk", side = secondary`**

   Suspends software mirroring for the database device *user_disk* on the secondary side.

3. **`disk unmirror name = "user_disk", mode = remove`**

   Suspends software mirroring for the database device *user_disk* and removes all device references to the mirror device.

Comments

- Disk mirroring creates a software mirror of a user database device, the master database device, or a database device used for user database transaction logs. If a database device fails, its mirror immediately takes over.

  disk unmirror disables either the original database device or the mirror, either permanently or temporarily, so that the device is no longer available to Adaptive Server for reads or writes. It does not remove the associated file from the operating system.

- Disk unmirroring alters the *sysdevices* table in the *master* database. It is important to back up the *master* database with the **dump database** command after each use of **disk unmirror.** This makes recovery easier and safer in case *master* is damaged.
- You can unmirror a database device while it is in use.
- You cannot unmirror any of a database's devices while a **dump database**, **load database**, or **load transaction** is in progress. Adaptive Server displays a message asking whether to abort the dump or load or to defer the **disk unmirror** until after the dump or load completes.
- You cannot unmirror a database's log device while a **dump transaction** is in progress. Adaptive Server displays a message asking whether to abort the dump or defer the **disk unmirror** until after the dump completes.

➤ ***Note***

dump transaction with truncate_only and **dump transaction with no_log** are not affected when a log device is unmirrored.

- You should mirror all the default database devices so that you are still protected if a **create** or **alter database** command affects a database device in the default list.
- When a read or write to a mirrored device is unsuccessful, Adaptive Server automatically unmirrors the bad device and prints error messages. Adaptive Server continues to run, unmirrored. A System Administrator must restart mirroring with the **disk remirror** command.
- For a report on all Adaptive Server devices on your system (user database devices and their mirrors, as well as dump devices), execute the system procedure **sp_helpdevice**.
- For more details about disk mirroring in the OpenVMS environment, see your the configuration documentation for your platform.
- Use **disk remirror** to reestablish mirroring after it is temporarily stopped with the **mode = retain** option of the **disk unmirror** command. If mirroring is permanently disabled with the **mode = remove** option, you must remove the operating system file that contains the mirror before using **disk remirror.**

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

disk unmirror permission defaults to the System Administrator, and is not transferable. You must be using the *master* database to use **disk unmirror.**

See Also

Commands	**alter database**, **create database**, **disk init**, **disk mirror**, **disk refit**, **disk reinit**, **disk remirror**, **dump database**, **dump transaction**, **load database**, **load transaction**
System procedures	**sp_diskdefault**, **sp_helpdevice**, **sp_logdevice**
Utility programs	**dataserver**, **startserver**

drop database

Function

Removes one or more databases from Adaptive Server.

Syntax

```
drop database database_name [, database_name]...
```

Keywords and Options

database_name – is the name of a database to remove. Use **sp_helpdb** to get a list of databases.

Examples

1. `drop database publishing`
2. `drop database publishing, newpubs`

   The dropped databases (and their contents) are gone.

Comments

- You must be using the *master* database to drop a database.
- Removing a database deletes the database and all its objects, frees its storage allocation, and erases its entries from the *sysdatabases* and *sysusages* system tables in the *master* database.
- You cannot drop a database that is in use (open for reading or writing by any user).
- You cannot use **drop database** to remove a database that is referenced by a table in another database. Execute the following query to determine which tables and external databases have foreign key constraints on primary key tables in the current database:

  ```
  select object_name(tableid), db_name(frgndbname)
  from sysreferences
  where frgndbname is not null
  ```

  Use **alter table** to drop these cross-database constraints, and then reissue the **drop database** command.
- You cannot use **drop database** to remove a damaged database. Use the **dbcc dbrepair** command:

  ```
  dbcc dbrepair (database_name, dropdb)
  ```

- You cannot drop the *sybsecurity* database if auditing is enabled. When auditing is disabled, only the System Security Officer can drop *sybsecurity*.
- **drop database** clears the suspect page entries pertaining to the dropped database from *master..sysattributes*.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Only the Database Owner can execute **drop database**, except for the *sybsecurity* database, which can be dropped only by the System Security Officer.

See Also

Commands	**alter database**, **create database**, **dbcc**, **use**
System procedures	**sp_changedbowner**, **sp_helpdb**, **sp_renamedb**, **sp_spaceused**

drop default

Function

Removes a user-defined default.

Syntax

```
drop default [owner.]default_name
   [, [owner.]default_name]...
```

Keywords and Options

default_name – is the name of an existing default. Execute **sp_help** to get a list of existing defaults. Specify the owner's name to drop a default of the same name owned by a different user in the current database. The default value for *owner* is the current user.

Examples

1. **drop default datedefault**

   Removes the user-defined default *datedefault* from the database.

Comments

- You cannot drop a default that is currently bound to a column or to a user-defined datatype. Use the system procedure **sp_unbindefault** to unbind the default before you drop it.
- You can bind a new default to a column or user-defined datatype without unbinding its current default. The new default overrides the old one.
- When you drop a default for a NULL column, NULL becomes the column's default value. When you drop a default for a NOT NULL column, an error message appears if users do not explicitly enter a value for that column when inserting data.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

drop default permission defaults to the owner of the default and is not transferable.

See Also

Commands	**create default**
System procedures	**sp_help**, **sp_helptext**, **sp_unbindefault**

drop index

Function

Removes an index from a table in the current database.

Syntax

```
drop index table_name.index_name
   [, table_name.index_name]...
```

Keywords and Options

table_name – is the table in which the indexed column is located. The table must be in the current database.

index_name – is the index to drop. In Transact-SQL, index names need not be unique in a database, though they must be unique within a table.

Examples

1. **drop index authors.au_id_ind**

   The index *au_id_ind* in the *authors* table no longer exists.

Comments

- Once the **drop index** command is issued, you regain all the space that was previously occupied by the index. This space can be used for any database objects.
- You cannot use **drop index** on system tables.
- **drop index** cannot remove indexes that support unique constraints. To drop such indexes, drop the constraints through **alter table** or drop the table. See **create table** for more information about unique constraint indexes.
- You cannot drop indexes that are currently used by any open cursor. For information about which cursors are open and what indexes they use, use **sp_cursorinfo**.
- To get information about what indexes exist on a table, use:

  ```
  sp_helpindex objname
  ```

  where *objname* is the name of the table.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

drop index permission defaults to the index owner and is not transferable.

See Also

Commands	**create index**
System procedures	**sp_cursorinfo**, **sp_helpindex**, **sp_spaceused**

drop procedure

Function

Removes a procedure.

Syntax

```
drop proc[edure] [owner.]procedure_name
   [, [owner.]procedure_name] ...
```

Keywords and Options

procedure_name – is the name of the procedure to drop. Specify the owner's name to drop a procedure of the same name owned by a different user in the current database. The default value for *owner* is the current user.

Examples

1. **drop procedure showind**

   Deletes the stored procedure **showind**.

2. **drop procedure xp_echo**

   Unregisters the extended stored procedure *xp_echo*.

Comments

- **drop procedure** drops user-defined stored procedures, system procedures, and extended stored procedures (ESPs).
- SQL server checks the existence of a procedure each time a user or a program executes that procedure.
- A procedure group (more than one procedure with the same name but with different ;*number* suffixes) can be dropped with a single **drop procedure** statement. For example, if the procedures used with the application named **orders** were named *orderproc;1*, *orderproc;2*, and so on, the following statement:

  ```
  drop proc orderproc
  ```

  drops the entire group. Once procedures have been grouped, individual procedures within the group cannot be dropped. For example, the statement:

  ```
  drop procedure orderproc;2
  ```

  is not allowed.

You cannot drop extended stored procedures as a procedure group.

- The system procedure **sp_helptext** displays the procedure's text, which is stored in *syscomments*.
- The system procedure **sp_helpextendedproc** displays ESPs and their corresponding DLLs.
- Dropping an ESP unregisters the procedure by removing it from the system tables. It has no effect on the underlying DLL.
- **drop procedure** drops only user-created procedures from your current database.

Standards and Compliance

Standard	Compliance level
SQL92	Transact-SQL extension

Permissions

drop procedure permission defaults to the procedure owner and is not transferable.

See Also

Commands	create procedure
System procedures	**sp_depends**, **sp_dropextendedproc**, **sp_helpextendedproc**, **sp_helptext**, **sp_rename**

drop role

Function

Drops a user-defined role.

Syntax

```
drop role role_name [with override]
```

Keywords and Options

role_name – is the name of the role you want to drop.

with override – overrides any restrictions on dropping a role. When you use the **with override** option, you can drop any role without having to check whether the role permissions have been dropped in each database.

Examples

1. **`drop role doctor_role`**

   Drops the named role only if all permissions in all databases have been revoked. The System Administrator or object owner must revoke permissions granted in each database before dropping a role, or the command fails.

2. **`drop role doctor_role with override`**

   Drops the named role and removes permission information and any other reference to the role from all databases.

Comments

- You need not drop memberships before dropping a role. Dropping a role automatically removes any user's membership in that role, regardless of whether you use the **with override** option.
- Use **drop role** from the *master* database.

Restrictions

- You cannot use **drop role** to drop system roles.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

You must be a System Security Officer to use **drop role**.

drop role permission is not included in the **grant all** command.

See Also

Commands	**alter role**, **create role**, **grant**, **revoke**, **set**
System procedures	**sp_activeroles**, **sp_displaylogin**, **sp_displayroles**, **sp_helprotect**, **sp_modifylogin**

drop rule

Function

Removes a user-defined rule.

Syntax

```
drop rule [owner.]rule_name [, [owner.]rule_name]...
```

Examples

1. **drop rule pubid_rule**

   Removes the rule *pubid_rule* from the current database.

Keywords and Options

rule_name – is the name of the rule to drop. Specify the owner's name to drop a rule of the same name owned by a different user in the current database. The default value for *owner* is the current user.

Comments

- Before dropping a rule, you must unbind it using the system procedure **sp_unbindrule**. If the rule has not been unbound, an error message appears, and the **drop rule** command fails.
- You can bind a new rule to a column or user-defined datatype without unbinding its current rule. The new rule overrides the old one.
- After you drop a rule, Adaptive Server enters new data into the columns that were previously governed by the rule without constraints. Existing data is not affected in any way.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

drop rule permission defaults to the rule owner and is not transferable.

See Also

Commands	**create rule**
System procedures	**sp_bindrule**, **sp_help**, **sp_helptext**, **sp_unbindrule**

drop table

Function

Removes a table definition and all of its data, indexes, triggers, and permissions from the database.

Syntax

```
drop table [[database.]owner.]table_name
   [, [[database.]owner.]table_name ]...
```

Keywords and Options

table_name – is the name of the table to drop. Specify the database name if the table is in another database, and specify the owner's name if more than one table of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

Examples

1. **`drop table roysched`**

   Removes the table *roysched* and its data and indexes from the current database.

Comments

- When you use **drop table**, any rules or defaults on the table lose their binding, and any triggers associated with it are automatically dropped. If you re-create a table, you must rebind the appropriate rules and defaults and re-create any triggers.
- The system tables affected when a table is dropped are *sysobjects*, *syscolumns*, *sysindexes*, *sysprotects*, and *syscomments*.
- If Component Integration Services is enabled, and if the table being dropped was created with **create existing table**, the table is not dropped from the remote server. Instead, Adaptive Server removes references to the table from the system tables.

Restrictions

- You cannot use the **drop table** command on system tables.
- Once you have partitioned a table, you cannot drop it. You must use the **unpartition** clause of the **alter tabl**e command before you can issue the **drop table** command.

- You can drop a table in any database, as long as you are the table owner. For example, to drop a table called *newtable* in the database *otherdb*:

  ```
  drop table otherdb..newtable
  ```

  or:

  ```
  drop table otherdb.yourname.newtable
  ```

- If you **delete** all the rows in a table or use the **truncate table** command, the table still exists until you **drop** it.

Dropping Tables with Cross-Database Referential Integrity Constraints

- When you create a cross-database constraint, Adaptive Server stores the following information in the *sysreferences* system table of each database:

Table 1-11: Information stored about referential integrity constraints

Information Stored in *sysreferences*	Columns with Information About Referenced Table	Columns with Information About Referencing Table
Key Column IDs	*refkey1* through *refkey16*	*fokey1* through *fokey16*
Table ID	*reftabid*	*tableid*
Database Name	*pmrydbname*	*frgndbname*

- Because the referencing table depends on information from the referenced table, Adaptive Server does not allow you to:
  - Drop the referenced table,
  - Drop the external database that contains it, or
  - Rename either database with **sp_renamedb.**

  Use the **sp_helpconstraint** system procedure to determine which tables reference the table you want to drop. Use **alter table** to drop the constraints before reissuing the **drop table** command.
- You can drop a referencing table or its database without problems. Adaptive Server automatically removes the foreign key information from the referenced database.
- Each time you add or remove a cross-database constraint or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases could cause database corruption. For more information about loading databases with cross-database referential integrity constraints, see "Cross-Database Constraints and Loading Databases" in Chapter 21, "Backing Up and Restoring User Databases" in the *System Administration Guide*.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

drop table permission defaults to the table owner and is not transferable.

See Also

Commands	**alter table**, **create table**, **delete**, **truncate table**
System procedures	**sp_depends**, **sp_help**, **sp_spaceused**

drop trigger

Function

Removes a trigger.

Syntax

```
drop trigger [owner.]trigger_name
   [, [owner.]trigger_name]...
```

Keywords and Options

trigger_name – is the name of the trigger to drop. Specify the owner's name to drop a trigger of the same name owned by a different user in the current database. The default value for *owner* is the current user.

Examples

1. `drop trigger trigger1`

   Removes the trigger *trigger1* from the current database.

Comments

- **drop trigger** drops a trigger in the current database.
- You do not need to explicitly drop a trigger from a table in order to create a new trigger for the same operation (**insert**, **update**, or **delete**). In a table or column each new trigger for the same operation overwrites the previous one.
- When a table is dropped, Adaptive Server automatically drops any triggers associated with it.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

drop trigger permission defaults to the trigger owner and is not transferable.

See Also

Commands	**create trigger**
System procedures	**sp_depends**, **sp_help**, **sp_helptext**

drop view

Function

Removes one or more views from the current database.

Syntax

```
drop view [owner.]view_name [, [owner.]view_name]...
```

Keywords and Options

view_name – is the name of the view to drop. Specify the owner's name to drop a view of the same name owned by a different user in the current database. The default value for *owner* is the current user.

Examples

1. **`drop view new_price`**

   Removes the view *new_price* from the current database.

Comments

- When you use **drop view**, the definition of the view and other information about it, including privileges, is deleted from the system tables *sysobjects*, *syscolumns*, *syscomments*, *sysdepends*, *sysprocedures*, and *sysprotects*.
- Existence of a view is checked each time the view is referenced, for example, by another view or by a stored procedure.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

drop view permission defaults to the view owner and is not transferable.

See Also

Commands	**create view**
System procedures	**sp_depends**, **sp_help**, **sp_helptext**

dump database

Function

Makes a backup copy of the entire database, including the transaction log, in a form that can be read in with **load database**. Dumps and loads are performed through Backup Server.

Syntax

```
dump database database_name
   to stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]]
   [[stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]]...]
   [with {
        density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name,
       [dismount | nodismount],
       [nounload | unload],
       retaindays = number_days,
       [noinit | init],
       notify = {client | operator_console}
       }]
```

Keywords and Options

database_name – is the name of the database from which you are copying data. The database name can be specified as a literal, a local variable, or a parameter to a stored procedure.

to *stripe_device* – is the device to which to copy the data. See "Specifying Dump Devices" in this section for information about what form to use when specifying a dump device.

at *backup_server_name* – is the name of the Backup Server. Do not specify this parameter when dumping to the default Backup Server. Specify this parameter only when dumping over the network to a remote Backup Server. You can specify up to 32 remote Backup Servers with this option. When dumping across the network, specify the *network name* of a remote Backup Server running on the machine to which the dump device is attached. For platforms that use interfaces files, the *backup_server_name* must appear in the interfaces file.

density = *density_value* – overrides the default density for a tape device. **Use this option only when reinitializing a volume on OpenVMS systems**. Valid densities are **800**, **1600**, **6250**, **6666**, **10000**, and **38000**. Not all values are valid for every tape drive; use the correct density for your tape drive.

blocksize = *number_bytes* – overrides the default block size for a dump device. (**Wherever possible, use the default block size**; it is the "best" block size for your system.) The block size must be at least one database page (2048 bytes for most systems) and must be an exact multiple of the database page size. On OpenVMS systems, block size cannot exceed 55,296 bytes.

capacity = *number_kilobytes* – is the maximum amount of data that the device can write to a single tape volume. The capacity must be at least five database pages and should be less than the recommended capacity for your device.

A general rule for calculating capacity is to use 70 percent of the manufacturer's maximum capacity for the device, allowing 30 percent for overhead such as inter-record gaps and tape marks. The maximum capacity is the capacity of the device on the drive, not the drive itself. This rule works in most cases, but may not work in all cases due to differences in overhead across vendors and across devices.

On UNIX platforms that cannot reliably detect the end-of-tape marker, indicate how many kilobytes can be dumped to the tape. You **must** supply a **capacity** for dump devices specified as a physical path name. If a dump device is specified as a logical device name, the Backup Server uses the *size* parameter stored in the *sysdevices* system table unless you specify a capacity.

dumpvolume = *volume_name* – establishes the name that is assigned to the volume. The maximum length of *volume_name* is 6 characters. Backup Server writes the *volume_name* in the ANSI tape label when overwriting an existing dump, dumping to a brand new tape, or dumping to a tape whose contents are not recognizable. The **load database** command checks the label and generates an error message if the wrong volume is loaded.

◆ ***WARNING!***

Be sure to label each tape volume as you create it so that the operator can load the correct tape.

stripe on *stripe_device* – is an additional dump device. Dumping to multiple devices is only supported for tape devices. You can use up to 32 devices, including the device named in the **to** *stripe_device* clause. The Backup Server splits the database into approximately equal portions, and sends each portion to a different device. Dumps are made concurrently on all devices, reducing the time required to make a dump and requiring fewer volume changes during the dump. See "Specifying Dump Devices" for information about how to specify a dump device.

dismount | **nodismount** – **on platforms, such as OpenVMS, that support logical dismount**, determines whether tapes remain mounted. By default, all tapes used for a dump are dismounted when the dump completes. Use **nodismount** to keep tapes available for additional dumps or loads.

nounload | **unload** – determines whether tapes rewind after the dump completes. By default, tapes do not rewind, allowing you to make additional dumps to the same tape volume. Specify **unload** for the last dump file to be added to a multidump volume. This rewinds and unloads the tape when the dump completes.

retaindays = *number_days* – **on UNIX systems**, specifies the number of days that Backup Server protects you from overwriting a dump. **This option is meaningful for disk, 1/4-inch cartridge, and single-file media. On multifile media, this option is meaningful for all volumes but the first**. If you try to overwrite a dump before it expires, Backup Server requests confirmation before overwriting the unexpired volume.

The *number_days* must be a positive integer or 0, for dumps that you can overwrite immediately. If you do not specify a **retaindays** value, Backup Server uses the **tape retention in days** value set by **sp_configure.**

noinit | **init** – determines whether to append the dump to existing dump files or reinitialize (overwrite) the tape volume. By default, Adaptive Server appends dumps following the last end-of-tape mark, allowing you to dump additional databases to the same volume. New dumps can be appended only to the last volume of a multivolume dump. Use **init** for the first database you dump to a tape to overwrite its contents.

Use **init** when you want Backup Server to store or update tape device characteristics in the tape configuration file. For more information, see "Tape Device Determination by Backup Server" in Chapter 21, "Backing Up and Restoring User Databases," in the *System Administration Guide.*

file = *file_name* – is the name of the dump file. The name cannot exceed 17 characters and must conform to operating system conventions for file names. If you do not specify a file name, Backup Server creates a default. For more information, see "Dump Files."

notify = {**client** | **operator_console**} – overrides the default message destination.

- On operating systems (such as OpenVMS) that offer an operator terminal feature, volume change messages are always sent to the operator terminal on the machine on which Backup Server is running. Use **client** to route other Backup Server messages to the terminal session that initiated the **dump database**.
- On operating systems (such as UNIX) that do not offer an operator terminal feature, messages are sent to the client that initiated the **dump database**. Use **operator_console** to route messages to the terminal on which Backup Server is running.

Examples

1. **For UNIX:**
```
dump database pubs2
  to "/dev/nrmt0"
```
For OpenVMS:
```
dump database pubs2
  to "MTA0:"
```
Dumps the database *pubs2* to a tape device. If the tape has an ANSI tape label, this command appends this dump to the files already on the tape, since the **init** option is not specified.

2. **For UNIX:**
```
dump database pubs2
    to "/dev/rmt4" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt5" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt0" at REMOTE_BKP_SERVER
with retaindays = 14
```
   For OpenVMS:
```
dump database pubs2
        to "MTA0:" at REMOTE_BKP_SERVER
    stripe on "MTA1:" at REMOTE_BKP_SERVER
    stripe on "MTA2:" at REMOTE_BKP_SERVER
```
   Dumps the *pubs2* database, using the REMOTE_BKP_SERVER Backup Server. The command names three dump devices, so the Backup Server dumps approximately one-third of the database to each device. This command appends the dump to existing files on the tapes. On UNIX systems, the **retaindays** option specifies that the tapes cannot be overwritten for 14 days. (OpenVMS systems do not use the **retaindays** option; they always create new versions of files.)

3. **For UNIX:**
```
dump database pubs2
    to "/dev/nrmt0"
    with init
```
   For OpenVMS:
```
dump database pubs2
    to "MTA0:"
    with init
```
   The **init** option initializes the tape volume, overwriting any existing files.

4. **For UNIX:**
```
dump database pubs2
    to "/dev/nrmt0"
    with unload
```
   For OpenVMS:
```
dump database pubs2
  to "MTA0:"
    with unload
```
   Rewinds the dump volumes upon completion of the dump.

5. **For UNIX:**
```
dump database pubs2
    to "/dev/nrmt0"
      with notify = client
```

```
For OpenVMS:
dump database pubs2
  to "MTA0:"
    with notify = client
```

The **notify** clause sends Backup Server messages requesting volume changes to the client which initiated the dump request, rather than sending them to the default location, the console of the Backup Server machine.

Comments

- Table 1-12 describes the commands and system procedures used to back up databases:

Table 1-12: Commands used to back up databases and logs

Use This Command	To Do This
dump database	Make routine dumps of the entire database, including the transaction log.
dump transaction	Make routine dumps of the transaction log, and then truncate the inactive portion.
dump transaction with no_truncate	Dump the transaction log after failure of a database device.
dump transaction with truncate_only then **dump database**	Truncate the log without making a backup. Copy the entire database.
dump transaction with no_log then **dump database**	Truncate the log after your usual method fails due to insufficient log space. Copy the entire database.
sp_volchanged	Respond to the Backup Server's volume change messages.

dump database Restrictions

- You cannot dump from an 11.x Adaptive Server to a 10.x Backup Server.
- You cannot have Sybase dumps and non-Sybase data (for example, UNIX archives) on the same tape.
- If a database has cross-database referential integrity constraints, the *sysreferences* system table stores the **name**—not the ID number—of the external database. Adaptive Server cannot guarantee referential integrity if you use **load database** to change the database name or to load it onto a different server.

◆ ***WARNING!***

Before dumping a database in order to load it with a different name or move it to another Adaptive Server, use alter table to drop all external referential integrity constraints.

- You cannot use the **dump database** command in a user-defined transaction.
- If you issue a **dump database** command on a database where a **dump transaction** is already in progress, the **dump database** command sleeps until the transaction dump completes.
- When using 1/4-inch cartridge tape, you can dump only one database or transaction log per tape.
- You cannot dump a database if it has offline pages. To force offline pages online, use **sp_forceonline_db** or **sp_forceonline_page.**

Scheduling Dumps

- Adaptive Server database dumps are **dynamic**—they can take place while the database is active. However, they may slow the system down slightly, so you may want to run **dump database** when the database is not being heavily updated.
- **Back up the *master* database regularly and frequently**. In addition to your regular backups, dump *master* after each **create database, alter database**, and **disk init** command is issued.
- Back up the *model* database each time you make a change to the database.
- Use **dump database** immediately after creating a database, to make a copy of the entire database. You cannot run **dump transaction** on a new database until you have run **dump database.**
- Each time you add or remove a cross-database constraint or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases could cause database corruption.

- Develop a regular schedule for backing up user databases and their transaction logs.

- Use thresholds to automate backup procedures. To take advantage of Adaptive Server's last-chance threshold, create user databases with log segments on a device that is separate from data segments. See Chapter 23, "Managing Free Space with Thresholds," in the *System Administration Guide* for more information about thresholds.

Dumping the System Databases

- The *master*, *model*, and *sybsystemprocs* databases do not have separate segments for their transaction logs. Use **dump transaction with truncate_only** to purge the log, and then use **dump database** to back up the database.
- Backups of the *master* database are needed for recovery procedures in case of a failure that affects the *master* database. See Chapter 22, "Restoring the System Databases," in the *System Administration Guide* for step-by-step instructions for backing up and restoring the *master* database.
- If you are using removable media for backups, the entire *master* database must fit on a single volume unless you have another Adaptive Server that can respond to volume change messages.

Specifying Dump Devices

- You can specify the dump device as a literal, a local variable, or a parameter to a stored procedure.
- You cannot dump to the null device (on UNIX, */dev/null*; on OpenVMS, any device name beginning with "NL").
- Dumping to multiple devices is only supported for tape devices.
- You can specify a local dump device as:
  - A logical device name from the *sysdevices* system table
  - An absolute path name
  - A relative path name

  Backup Server resolves relative path names using Adaptive Server's current working directory.
- When dumping across the network, you must specify the absolute path name of the dump device. The path name must be valid on the machine on which Backup Server is running. If the name includes any characters except letters, numbers, or the underscore (_), you must enclose it in quotes.

- Ownership and permissions problems on the dump device may interfere with the use of **dump** commands. The **sp_addumpdevice** procedure adds the device to the system tables, but does not guarantee that you can dump to that device or create a file as a dump device.
- You can run more than one dump (or load) at the same time, as long as each uses different dump devices.
- If the device file already exists, Backup Server overwrites it; it does not truncate it. For example, suppose you dump a database to a device file and the device file becomes 10MB. If the next dump of the database to that device is smaller, the device file is still 10MB.

Determining Tape Device Characteristics

- If you issue a **dump** command without the **init** qualifier and Backup Server cannot determine the device type, the **dump** command fails. For more information, see "Tape Device Determination by Backup Server" in Chapter 21, "Backing Up and Restoring User Databases," of the *System Administration Guide*.

Backup Servers

- You must have a Backup Server running on the same machine as Adaptive Server. (On OpenVMS systems, the Backup Server can be running in the same cluster as the Adaptive Server, as long as all database devices are visible to both.) The Backup Server must be listed in the *master..sysservers* table. This entry is created during installation or upgrade, and should not be deleted.
- If your backup devices are located on another machine so that you dump across a network, you must also have a Backup Server installed on the remote machine.

Dump Files

- Dumping a database with the **init** option overwrites any existing files on the tape or disk.
- Dump file names identify which database was dumped and when the dump was made. If you do not specify a file name, Backup Server creates a default file name by concatenating the following:
  - Last seven characters of the database name
  - Two-digit year number

- Three-digit day of the year (1–366)
- Hexadecimal-encoded time at which the dump file was created

For example, the file *cations930590E100* contains a copy of the *publications* database made on the fifty-ninth day of 1993:

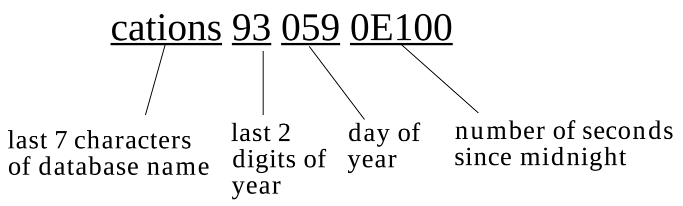

cations 93 059 0E100
last 7 characters of database name
last 2 digits of year
day of year
number of seconds since midnight

Figure 1-2: File naming convention for database dumps

- Backup Server sends the dump file name to the location specified by the **with notify** clause. Before storing a backup tape, the operator should label it with the database name, file name, date, and other pertinent information. When loading a tape without an identifying label, use the **with headeronly** and **with listonly** options to determine the contents.

Volume Names

- Dump volumes are labeled according to the ANSI tape-labeling standard. The label includes the logical volume number and the position of the device within the stripe set.
- During loads, Backup Server uses the tape label to verify that volumes are mounted in the correct order. This allows you to load from a smaller number of devices than you used at dump time.

➤ ***Note***

When dumping and loading across the network, you must specify the same number of stripe devices for each operation.

Changing Dump Volumes

- On OpenVMS systems, the operating system requests a volume change when it detects the end of a volume or when the specified drive is offline. After mounting another volume, the operator uses the **REPLY** command to reply to these messages.

- On UNIX systems, Backup Server requests a volume change when the tape capacity has been reached. After mounting another volume, the operator notifies Backup Server by executing the **sp_volchanged** system procedure on any Adaptive Server that can communicate with Backup Server.
- If Backup Server detects a problem with the currently mounted volume, it requests a volume change by sending messages to either the client or its operator console. The operator responds to these messages with the **sp_volchanged** system procedure.

Appending to or Overwriting a Volume

- By default (**noinit**), Backup Server writes successive dumps to the same tape volume, making efficient use of high-capacity tape media. Data is added following the last end-of-tape mark. New dumps can be appended only to the last volume of a multi-volume dump. Before writing to the tape, Backup Server verifies that the first file has not yet expired. If the tape contains non-Sybase data, Backup Server rejects it to avoid destroying potentially valuable information.
- Use the **init** option to reinitialize a volume. If you specify **init**, Backup Server overwrites any existing contents, even if the tape contains non-Sybase data, the first file has not yet expired, or the tape has ANSI access restrictions.
- Figure 1-3 illustrates how to dump three databases to a single volume using:
  - **init** to initialize the tape for the first dump
  - **noinit** (the default) to append subsequent dumps
  - **unload** to rewind and unload the tape after the last dump

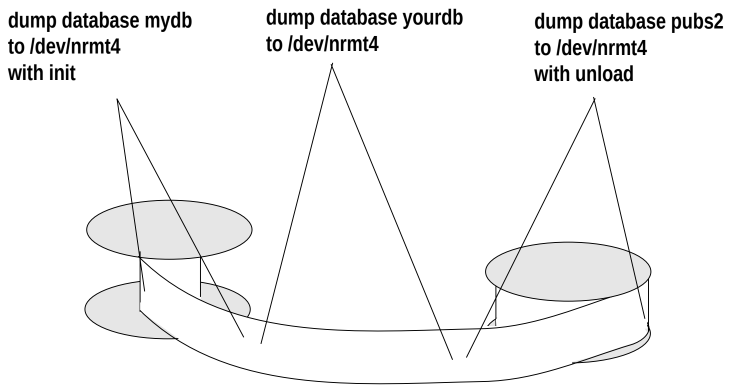


Figure 1-3: Dumping several databases to the same volume

Dumping Databases Whose Devices Are Mirrored

- At the beginning of a **dump database**, Adaptive Server passes Backup Server the primary device name of all database and log devices. If the primary device has been unmirrored, Adaptive Server passes the name of the secondary device instead. If any named device fails before the Backup Server completes its data transfer, Adaptive Server aborts the dump.
- If a user attempts to unmirror any of the named database devices while a **dump database** is in progress, Adaptive Server displays a message. The user executing the **disk unmirror** command can abort the dump or defer the **disk unmirror** until after the dump is complete.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Only the System Administrator, the Database Owner, and users with the Operator role can execute **dump database**.

See Also

Commands	**dump transaction, load database, load transaction**
System procedures	**sp_addthreshold, sp_addumpdevice, sp_dropdevice, sp_dropthreshold, sp_helpdevice, sp_helpdb, sp_helpthreshold, sp_logdevice, sp_spaceused, sp_volchanged**

dump transaction

Function

Makes a copy of a transaction log and removes the inactive portion.

Syntax

To make a routine log dump:

```
dump tran[saction] database_name
   to stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]]
   [[stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name] ]...]
   [with {
       density = density_value,
       blocksize = number_bytes,
       capacity = number_kilobytes,
       dumpvolume = volume_name,
       file = file_name,
       [dismount | nodismount],
       [nounload | unload],
       retaindays = number_days,
       [noinit | init],
       notify = {client | operator_console}}]
```

To truncate the log without making a backup copy:

```
dump tran[saction] database_name
   with truncate_only
```

To truncate a log that is filled to capacity. **Use only as a last resort**:

```
dump tran[saction] database_name
   with no_log
```

To back up the log after a database device fails:

```
dump tran[saction] database_name
   to stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]]
   [[stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name] ]...]
   [with {
       density = density_value,
       blocksize = number_bytes,
       capacity = number_kilobytes,
       dumpvolume = volume_name,
       file = file_name,
       [dismount | nodismount],
       [nounload | unload],
       retaindays = number_days,
       [noinit | init],
       no_truncate,
       notify = {client | operator_console}}]
```

Keywords and Options

database_name – is the name of the database from which you are copying data. The name can be given as a literal, a local variable, or a parameter to a stored procedure.

truncate_only – removes the inactive part of the log **without making a backup copy**. Use on databases without log segments on a separate device from data segments, Do not specify a dump device or Backup Server name.

no_log – removes the inactive part of the log **without making a backup copy and without recording the procedure in the transaction log**. Use no_log only when you have totally run out of log space and cannot run your usual **dump transaction** command. Use **no_log** as a last resort and use it only once after **dump transaction with truncate_only** fails. For additional information, see "When All Else Fails: with no_log" or "Truncating a Log That Has No Free Space" in Chapter 21, "Backing Up and Restoring User Databases," of the *System Administration Guide*.

to *stripe_device* – is the device to which data is being dumped. See "Specifying Dump Devices" for information about what form to use when specifying a dump device.

at *backup_server_name* – is the name of the Backup Server. Do not specify this parameter if dumping to the default Backup Server. Specify this parameter only if dumping over the network to a remote Backup Server. You can specify up to 32 different remote Backup Servers using this option. When dumping across the network, specify the *network name* of a remote Backup Server running on the machine to which the dump device is attached. For platforms that use interfaces files, the *backup_server_name* must appear in the interfaces file.

density = *density_value* – overrides the default density for a tape device. **Use this option only when reinitializing a volume on OpenVMS systems**. Valid densities are **800**, **1600**, **6250**, **6666**, **10000**, and **38000**. Not all values are valid for every tape drive; use the correct density for your tape drive.

blocksize = *number_bytes* – overrides the default block size for a dump device. (**Wherever possible, use the default block size**; it is the best block size for your system.) The block size must be at least one database page (2048 bytes for most systems) and must be an exact multiple of the database page size. On OpenVMS systems, block size cannot exceed 55,296 bytes.

capacity = *number_kilobytes* – is the maximum amount of data that the device can write to a single tape volume. The capacity must be at least five database pages, and should be slightly less than the recommended capacity for your device.

A general rule for calculating capacity is to use 70 percent of the manufacturer's maximum capacity for the device, leaving 30 percent for overhead such as inter-record gaps and tape marks. This rule works in most cases, but may not work in all cases because of differences in overhead across vendors and devices.

OpenVMS systems write until they reach the physical end-of-tape marker, when they send a volume change request.

On UNIX platforms that cannot reliably detect the end-of-tape marker, you must indicate how many kilobytes can be dumped to the tape. You **must** supply a **capacity** for dump devices specified as a physical path name. If a dump device is specified as a logical device name, the Backup Server uses the *size* parameter stored in the *sysdevices* system table, unless you specify a capacity.

dumpvolume = *volume_name* – establishes the name that is assigned to the volume. The maximum length of *volume_name* is 6 characters. The Backup Server writes the *volume_name* in the ANSI tape label when overwriting an existing dump, dumping to a brand new tape, or dumping to a tape whose contents are not recognizable. The **load transaction** command checks the label and generates an error message if the wrong volume is loaded.

stripe on *stripe_device* – is an additional dump device. You can use up to 32 devices, including the device named in the **to** *stripe_device* clause. The Backup Server splits the log into approximately equal portions and sends each portion to a different device. Dumps are made concurrently on all devices, reducing the time and the number of volume changes required. See “Specifying Dump Devices” for information about how to specify a dump device.

dismount | **nodismount** – **on platforms that support logical dismount** (such as OpenVMS), determines whether tapes remain mounted. By default, all tapes used for a dump are dismounted when the dump completes. Use **nodismount** to keep tapes available for additional dumps or loads.

nounload | **unload** – determines whether tapes rewind after the dump completes. By default, tapes do not rewind, allowing you to make additional dumps to the same tape volume. Specify **unload** for the last dump file to be added to a multidump volume. This rewinds and unloads the tape when the dump completes.

retaindays = *number_days* – **on UNIX systems**, specifies the number of days that Backup Server protects you from overwriting a dump. **This option is meaningful for disk, 1/4-inch cartridge, and single-file media. On multifile media, this option is meaningful for all volumes but the first**. If you try to overwrite a dump before it expires, Backup Server requests confirmation before overwriting the unexpired volume.

The *number_days* must be a positive integer or 0, for dumps you can overwrite immediately. If you do not specify a **retaindays** value, Backup Server uses the server-wide **tape retention in days** value, set by **sp_configure.**

noinit | **init** – determines whether to append the dump to existing dump files or reinitialize (overwrite) the tape volume. By default, Adaptive Server appends dumps following the last end-of-tape mark, allowing you to dump additional databases to the same volume. New dumps can be appended only to the last volume of a multivolume dump. Use **init** for the first database you dump to a tape, to overwrite its contents.

Use **init** when you want Backup Server to store or update tape device characteristics in the tape configuration file. For more information, see "Tape Device Determination by Backup Server" in Chapter 21, "Backing Up and Restoring User Databases," in the *System Administration Guide.*

file = *file_name* – is the name of the dump file. The name cannot exceed 17 characters and must conform to operating system conventions for file names. If you do not specify a file name, Backup Server creates a default file name. For more information, see "Dump Files".

no_truncate – dumps a transaction log, **even if the disk containing the data segments for a database is inaccessible**, using a pointer to the transaction log in the *master* database. The **with no_truncate** option provides up-to-the-minute log recovery when the transaction log resides on an undamaged device, and the *master* database and user databases reside on different physical devices.

notify = {**client** | **operator_console**} – overrides the default message destination.

- On operating systems (such as OpenVMS) that offer an operator terminal feature, volume change messages are always sent to the operator terminal on the machine on which the Backup Server is running. Use **client** to route other Backup Server messages to the terminal session that initiated the **dump database.**
- On operating systems (such as UNIX) that do not offer an operator terminal feature, messages are sent to the client that initiated the **dump database**. Use **operator_console** to route messages to the terminal on which the Backup Server is running.

Examples

1. **For UNIX:**
   ```
   dump transaction pubs2
     to "/dev/nrmt0"
   ```
   For OpenVMS:
   ```
   dump database pubs2
     to "MTA0:"
   ```
   Dumps the transaction log to a tape, appending it to the files on the tape, since the **init** option is not specified.

2. **For UNIX:**
   ```
   dump transaction mydb
       to "/dev/nrmt4" at REMOTE_BKP_SERVER
       stripe on "/dev/nrmt5" at REMOTE_BKP_SERVER
   with init, retaindays = 14
   ```
   For OpenVMS:
   ```
   dump transaction mydb
       to "MTA0:" at REMOTE_BKP_SERVER
       stripe on "MTA1:" at REMOTE_BKP_SERVER
   with init
   ```
   Dumps the transaction log for the *mydb* database, using the Backup Server REMOTE_BKP_SERVER. The Backup Server dumps approximately half the log to each of the two devices. The **init** option overwrites any existing files on the tape. On UNIX systems, the **retaindays** option specifies that the tapes cannot be overwritten for 14 days. (OpenVMS systems do not use **retaindays**; they always create new versions of dump files.)

Comments

- Table 1-13 describes the commands and system procedures used to back up databases and logs.:

Table 1-13: Commands used to back up databases and logs

Use This Command	To Do This
dump database	Make routine dumps of the entire database, including the transaction log.
dump transaction	Make routine dumps of the transaction log, then truncate the inactive portion.
dump transaction with no_truncate	Dump the transaction log after failure of a database device.

Table 1-13: Commands used to back up databases and logs (continued)

Use This Command	To Do This
dump transaction with truncate_only then **dump database**	Truncate the log without making a backup. Copy the entire database.
dump transaction with no_log then **dump database**	Truncate the log after your usual method fails due to insufficient log space. Copy the entire database.
sp_volchanged	Respond to the Backup Server's volume change messages.

Restrictions

- You cannot dump to the null device (on UNIX, */dev/null*; on OpenVMS, any device name beginning with "NL").
- You cannot use the **dump transaction** command in a transaction.
- When using 1/4-inch cartridge tape, you can dump only one database or transaction log per tape.
- You cannot issue dump the transaction log while the **trunc log on chkpt** database option is enabled or after enabling **select into/bulk copy/pllsort** and making minimally logged changes to the database with **select into**, fast bulk copy operations, default unlogged **writetext** operations, or a parallel sort. Use **dump database** instead.

◆ ***WARNING!***

Never modify the log table *syslogs* with a *delete*, *update*, or *insert* command.

- If a database does not have a log segment on a separate device from data segments, you cannot use **dump transaction** to copy the log and truncate it.
- If a user or threshold procedure issues a **dump transaction** command on a database where a **dump database** or another **dump transaction** is in progress, the second command sleeps until the first completes.
- To restore a database, use **load database** to load the most recent database dump; then use **load transaction** to load each subsequent transaction log dump **in the order in which it was made**.

- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases can cause database corruption.

- You cannot dump from an 11.x Adaptive Server to a 10.x Backup Server.
- You cannot have Sybase dumps and non-Sybase data (for example, UNIX archives) on the same tape.
- You cannot dump a transaction **with no_log** or **with truncate_only** if the database has offline pages.

Copying the Log After Device Failure: *with no_truncate*

- After device failure, use **dump transaction with no_truncate** to copy the log without truncating it. You can use this option only if your log is on a separate segment and your *master* database is accessible.
- The backup created by **dump transaction with no_truncate** is the most recent dump for your log. When restoring the database, load this dump last.

Databases Without Separate Log Segments: *with truncate_only*

- When a database does not have a log segment on a separate device from data segments, use **dump transaction with truncate_only** to remove committed transactions from the log without making a backup copy.

◆ ***WARNING!***

dump transaction with truncate_only provides no means to recover your databases. Run dump database at the earliest opportunity to ensure recoverability.

- Use **with truncate_only** on the *master*, *model*, and *sybsystemprocs* databases, which do not have log segments on a separate device from data segments.
- You can also use this option on very small databases that store the transaction log and data on the same device.

- Mission-critical user databases should have log segments on a separate device from data segments. Use the **log on** clause of **create database** to create a database with a separate log segment, or **alter database** and **sp_logdevice** to transfer the log to a separate device.

When All Else Fails: *with no_log*

- Use **dump transaction with no_log** only as a last resort, after your usual method of dumping the transaction log (**dump transaction** or **dump transaction with truncate_only**) fails because of insufficient log space.
- **dump transaction...with no_log** truncates the log without logging the dump transaction event. Because it copies no data, it requires only the name of the database.
- Every use of **dump transaction...with no_log** is considered an error and is recorded in Adaptive Server's error log.

◆ *WARNING!*

dump transaction with no_log provides no means to recover your databases. Run dump database at the earliest opportunity to ensure recoverability.

- If you have created your databases with log segments on a separate device from data segments, written a last-chance threshold procedure that dumps your transaction log often enough, and allocated enough space to your log and database, you should not have to use this option. If you must use **with no_log**, increase the frequency of your dumps and the amount of log space.

Scheduling Dumps

- Transaction log dumps are **dynamic**—they can take place while the database is active. They may slow the system slightly, so run dumps when the database is not being heavily updated.
- Use **dump database** immediately after creating a database to make a copy of the entire database. You cannot run **dump transaction** on a new database until you have run **dump database**.
- Develop a regular schedule for backing up user databases and their transaction logs.

- **dump transaction** uses less storage space and takes less time than **dump database**. Typically, transaction log dumps are made more frequently than database dumps.

Using Thresholds to Automate *dump transaction*

- Use thresholds to automate backup procedures. To take advantage of Adaptive Server's last-chance threshold, create user databases with log segments on a separate device from data segments.
- When space on the log segment falls below the last-chance threshold, Adaptive Server executes the last-chance threshold procedure. Including a **dump transaction** command in your last-chance threshold procedure helps protect you from running out of log space. For more information, see **sp_thresholdaction**.
- You can use **sp_addthreshold** to add a second threshold to monitor log space. For more information about thresholds, see Chapter 23, "Managing Free Space with Thresholds," in the *System Administration Guide*.

Specifying Dump Devices

- You can specify the dump device as a literal, a local variable, or a parameter to a stored procedure.
- You can specify a local dump device as:
  - A logical device name from the *sysdevices* system table
  - An absolute path name
  - A relative path name

  The Backup Server resolves relative path names using Adaptive Server's current working directory.
- When dumping across the network, specify the absolute path name of the dump device. The path name must be valid on the machine on which the Backup Server is running. If the name includes any characters except letters, numbers, or the underscore (_), enclose it in quotes.
- Ownership and permissions problems on the dump device may interfere with use of **dump** commands. The **sp_addumpdevice** procedure adds the device to the system tables, but does not guarantee that you can dump to that device or create a file as a dump device.

- You can run more than one dump (or load) at the same time, as long as they use different dump devices.

Determining Tape Device Characteristics

- If you issue a **dump transaction** command without the **init** qualifier and Backup Server cannot determine the device type, the **dump transaction** command fails. For more information, see "Tape Device Determination by Backup Server" in Chapter 21, "Backing Up and Restoring User Databases," of the *System Administration Guide*.

Backup Servers

- You must have a Backup Server running on the same machine as your Adaptive Server. (On OpenVMS systems, the Backup Server can be running in the same cluster as the Adaptive Server, as long as all database devices are visible to both.) The Backup Server must be listed in the *master..sysservers* table. This entry is created during installation or upgrade and should not be deleted.
- If your backup devices are located on another machine so that you dump across a network, you must also have a Backup Server installed on the remote machine.

Dump Files

- Dumping a log with the **init** option overwrites any existing files on the tape or disk.
- Dump file names identify which database was dumped and when the dump was made. If you do not specify a file name, Backup Server creates a default file name by concatenating the following:
  - Last seven characters of the database name
  - Two-digit year number
  - Three-digit day of the year (1– 366)
  - Hexadecimal-encoded time at which the dump file was created

For example, the file *cations930590E100* contains a copy of the *publications* database made on the fifty-ninth day of 1993:

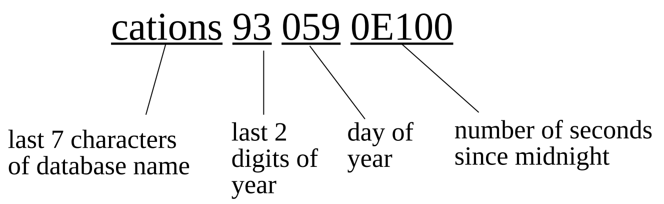

cations 93 059 0E100
last 7 characters of database name
last 2 digits of year
day of year
number of seconds since midnight

Figure 1-4: File naming convention for transaction log dumps

- The Backup Server sends the dump file name to the location specified by the **with notify** clause. Before storing a backup tape, the operator should label it with the database name, file name, date, and other pertinent information. When loading a tape without an identifying label, use the **with headeronly** and **with listonly** options to determine the contents.

Volume Names

- Dump volumes are labeled according to the ANSI tape-labeling standard. The label includes the logical volume number and the position of the device within the stripe set.
- During loads, Backup Server uses the tape label to verify that volumes are mounted in the correct order. This allows you to load from a smaller number of devices than you used at dump time.

➤ ***Note***

When dumping and loading across the network, you must specify the same number of stripe devices for each operation.

Changing Dump Volumes

- On OpenVMS systems, the operating system requests a volume change when it detects the end of a volume or when the specified drive is offline. After mounting another volume, the operator uses the **REPLY** command to reply to these messages.
- On UNIX systems, the Backup Server requests a volume change when the tape capacity has been reached. After mounting another volume, the operator notifies the Backup Server by

executing the **sp_volchanged** system procedure on any Adaptive Server that can communicate with the Backup Server.

- If the Backup Server detects a problem with the currently mounted volume (for example, if the wrong volume is mounted), it requests a volume change by sending messages to either the client or its operator console. The operator responds to these messages with the **sp_volchanged** system procedure.

Appending to/Overwriting a Volume

- By default (**noinit**), Backup Server writes successive dumps to the same tape volume, making efficient use of high-capacity tape media. Data is added following the last end-of-tape mark. New dumps can be appended only to the last volume of a multivolume dump. Before writing to the tape, Backup Server verifies that the first file has not yet expired. If the tape contains non-Sybase data, Backup Server rejects it to avoid destroying potentially valuable information.
- Use the **init** option to reinitialize a volume. If you specify **init**, Backup Server overwrites any existing contents, even if the tape contains non-Sybase data, the first file has not yet expired, or the tape has ANSI access restrictions.
- Figure 1-5 illustrates how to dump three transaction logs to a single volume. Use:
  - **init** to initialize the tape for the first dump
  - **noinit** (the default) to append subsequent dumps
  - **unload** to rewind and unload the tape after the last dump

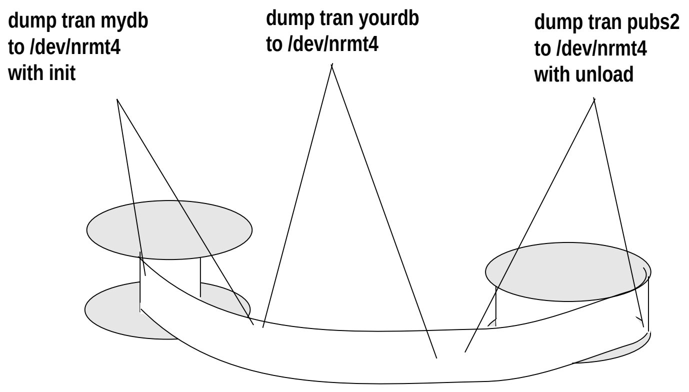


Figure 1-5: Dumping three transaction logs to a single volume

Dumping Logs Stored on Mirrored Devices

- At the beginning of a **dump transaction**, Adaptive Server passes the primary device name of each logical log device to the Backup Server. If the primary device has been unmirrored, Adaptive Server passes the name of the secondary device instead. If the named device fails before Backup Server completes its data transfer, Adaptive Server aborts the dump.
- If you attempt to unmirror a named log device while a **dump transaction** is in progress, Adaptive Server displays a message. The user executing the **disk unmirror** command can abort the dump or defer the **disk unmirror** until after the dump completes.
- **dump transaction with truncate_only** and **dump transaction with no_log** do not use the Backup Server. These commands are not affected when a log device is unmirrored, either by a device failure or by a **disk unmirror** command.
- **dump transaction** copies only the log segment. It is not affected when a data-only device is unmirrored, either by a device failure or by a **disk unmirror** command.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Only System Administrators, users who have been granted the Operator role, and the Database Owner can execute **dump transaction**.

See Also

Commands	**dump database, load database, load transaction**
System procedures	**sp_addumpdevice, sp_dboption, sp_dropdevice, sp_helpdevice, sp_logdevice, sp_volchanged**

execute

Function

Runs a procedure.

Syntax

```
[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]
```

Keywords and Options

execute | exec – is used to execute a stored procedure or an extended stored procedure (ESP). It is necessary only if the stored procedure call is **not** the first statement in a batch.

@return_status – is an optional integer variable that stores the return status of a stored procedure. It must be declared in the batch or stored procedure before it is used in an execute statement.

server – is the name of a remote server. You can execute a procedure on another Adaptive Server as long as you have permission to use that server and to execute the procedure in that database. If you specify a server name, but do not specify a database name, Adaptive Server looks for the procedure in your default database.

database – is the database name. Specify the database name if the procedure is in another database. The default value for *database* is the current database. You can execute a procedure in another database as long as you are its owner or have permission to execute it in that database.

owner – is the procedure owner's name. Specify the owner's name if more than one procedure of that name exists in the database. The default value for *owner* is the current user. The owner name is optional only if the Database Owner ("dbo") owns the procedure or if you own it.

procedure_name – is the name of a procedure defined with a **create procedure** statement.

;number – is an optional integer used 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, if the procedures used with an application named **orders** are named *orderproc;1*, *orderproc;2*, and so on, the statement:

```
drop proc orderproc
```

drops the entire group. Once procedures have been grouped, individual procedures within the group cannot be dropped. For example, you cannot execute the statement:

```
drop procedure orderproc;2
```

parameter_name – is the name of an argument to the procedure, as defined in the **create procedure** statement. Parameter names must be preceded by the @ sign.

If the "*@parameter_name* = *value*" form is used, parameter names and constants need not be supplied in the order defined in the **create procedure** statement. However, if this form is used for any parameter, it must be used for all subsequent parameters.

value – is the value of the parameter or argument to the procedure. If you do not use the "*@parameter_name* = *value*" form, you must supply parameter values in the order defined in the **create procedure** statement.

@variable – is the name of a variable used to store a return parameter.

output – indicates that the stored procedure is to return a return parameter. The matching parameter in the stored procedure must also have been created with the keyword **output**.

The **output** keyword can be abbreviated to **out**.

with recompile – forces compilation of a new plan. Use this option if the parameter you are supplying is atypical or if the data has significantly changed. The changed plan is used on subsequent executions. Adaptive Server ignores this option when executing an ESP.

Examples

1.
```
execute showind titles
```

or:

```
exec showind @tabname = titles
```

or, if this is the only statement in a batch or file:

```
showind titles
```

All three examples above execute the stored procedure *showind* with a parameter value *titles*.

2.
```
declare @retstat int
execute @retstat = GATEWAY.pubs.dbo.checkcontract
"409-56-4008"
```

   Executes the stored procedure *checkcontract* on the remote server GATEWAY. Stores the return status indicating success or failure in *@retstat*.

3.
```
declare @percent int
select @percent = 10
execute roy_check "BU1032", 1050, @pc = @percent
output
select Percent = @percent
```

   Executes the stored procedure *roy_check*, passing three parameters. The third parameter, *@pc*, is an **output** parameter. After execution of the procedure, the return value is available in the variable *@percent*.

4.
```
create procedure
showsysind @table varchar(30) = "sys%"
as
  select sysobjects.name, sysindexes.name, indid
  from sysindexes, sysobjects
  where sysobjects.name like @table
  and sysobjects.id = sysindexes.id
```

   This procedure displays information about the system tables if the user does not supply a parameter.

5.
```
declare @input varchar(12)
select @input="Hello World!"
execute xp_echo @in = @input, @out= @result output
```

   Executes the extended stored procedure *xp_echo*, passing in a value of “Hello World!”. The returned value of the extended stored procedure is stored in a variable named *result*.

Comments

- Procedure results may vary, depending on the database in which they are executed. For example, the user-defined system procedure *sp_foo*, which executes the **db_name()** system function, returns the name of the database from which it is executed. When executed from the *pubs2* database, it returns the value “pubs2”:

```
exec pubs2..sp_foo
```

```
------------------------------
pubs2

(1 row affected, return status = 0)
```

When executed from *sybsystemprocs*, it returns the value "sybsystemprocs":

```
exec sybsystemprocs..sp_foo
```

```
------------------------------
sybsystemprocs

(1 row affected, return status = 0)
```

- There are two ways to supply parameters—by position, or by using:

  @parameter_name = value

  If you use the second form, you do not have to supply the parameters in the order defined in the **create procedure** statement.

  If you are using the **output** keyword and intend to use the return parameters in additional statements in your batch or procedure, the value of the parameter must be passed as a variable. For example:

  parameter_name = @variable_name

  When executing an extended stored procedure, pass all parameters either by name or by value. You cannot mix parameters by value and parameters by name in a single invocation of the **execute** command for an ESP.
- You cannot use *text* and *image* columns as parameters to stored procedures or as values passed to parameters.
- It is an error to execute a procedure specifying **output** for a parameter that is not defined as a return parameter in the **create procedure** statement.
- You cannot pass constants to stored procedures using **output**; the return parameter requires a variable name. You must declare the variable's datatype and assign it a value before executing the procedure. Return parameters cannot have a datatype of *text* or *image*.
- It is not necessary to use the keyword **execute** if the statement is the first one in a batch. A batch is a segment of an input file terminated by the word "go" on a line by itself.

- Since the execution plan for a procedure is stored the first time it is run, subsequent run time is much shorter than for the equivalent set of standalone statements.
- Nesting occurs when one stored procedure calls another. The nesting level is incremented when the called procedure begins execution and it is decremented when the called procedure completes execution. Exceeding the maximum of 16 levels of nesting causes the transaction to fail. The current nesting level is stored in the *@@nestlevel* global variable.
- Return values 0 and -1 through -14 are currently used by Adaptive Server to indicate the execution status of stored procedures. Values from -15 through -99 are reserved for future use. See **return** for a list of values.
- Parameters are not part of transactions, so if a parameter is changed in a transaction which is later rolled back, its value does not revert to its previous value. The value that is returned to the caller is always the value at the time the procedure returns.
- If you use **select** * in your **create procedure** statement, the procedure does not pick up any new columns you may have added to the table (even if you use the **with recompile** option to execute). You must **drop** the procedure and re-create it.
- Commands executed via remote procedure calls cannot be rolled back.
- The **with recompile** option is ignored when Adaptive Server executes an extended stored procedure

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

execute permission defaults to the owner of the procedure, who can transfer it to other users.

See Also

Commands	**create procedure, drop procedure, return**
System procedures	**sp_addextendedproc**, **sp_depends**, **sp_dropextendedproc**, **sp_helptext**

fetch

Function

Returns a row or a set of rows from a cursor result set.

Syntax

```
fetch cursor_name [ into fetch_target_list ]
```

Parameters

cursor_name – the name of the cursor

into *fetch_target_list* – is a comma-separated list of parameters or local variables into which cursor results are placed. The parameters and variables must be declared prior to the **fetch**.

Examples

1. **`fetch authors_crsr`**

   Returns a row of information from the cursor result set defined by the *authors_crsr* cursor.

2. **`fetch pubs_crsr into @name, @city, @state`**

   Returns a row of information from the cursor result set defined by the *pubs_crsr* cursor into the variables *@name*, *@city*, and *@state*.

Comments

Restrictions

- Before you can use **fetch**, you must declare the cursor and **open** it.
- The *cursor_name* cannot be a Transact-SQL parameter or local variable.
- You cannot **fetch** a row that has already been fetched. There is no way to backtrack through the result set, but you can close and reopen the cursor to create the cursor result set again and start from the beginning.
- 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.

- When you set chained transaction mode, Adaptive Server implicitly begins a transaction with the **fetch** statement if no transaction is currently active. However, this situation occurs only when you set the **close on endtran** option and the cursor remains open after the end of the transaction that initially opened it, since the **open** statement also automatically begins a transaction.

Cursor Position

- 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 indicating there is no more data, and the cursor position moves beyond the end of the result set. You can no longer **update** or **delete** from that current cursor position.
- With **fetch into**, Adaptive Server does not advance the cursor position when an error occurs because the number of variables in the *fetch_target_list* does not equal the number of target list expressions specified by the query that defines the cursor. However, it does advance the cursor position, even if a compatibility error occurs between the datatypes of the variables and the datatypes of the columns in the cursor result set.

Determining How Many Rows Are Fetched

- You can **fetch** one or more rows at a time. Use the **cursor rows** option of the **set** command to specify the number of rows to **fetch**.

Getting Information About Fetches

- The *@@sqlstatus* global variable holds status information (warning exceptions) resulting from the execution of a **fetch** statement. The value of *@@sqlstatus* is 0, 1, or 2, as shown in Table 1-14.

Table 1-14: @@sqlstatus values

0	Indicates successful completion of the **fetch** statement.
1	Indicates that the **fetch** statement resulted in an error.
2	Indicates that there is no more data in the result set. This warning can occur if the current cursor position is on the last row in the result set and the client submits a **fetch** statement for that cursor.

Only a **fetch** statement can set *@@sqlstatus*. Other statements have no effect on *@@sqlstatus*.

- The *@@rowcount* global variable holds the number of rows returned from the cursor result set to the client up to the last **fetch**. In other words, it represents the total number of rows seen by the client at any one time.

  Once all the rows have been read from the cursor result set, *@@rowcount* represents the total number of rows in the cursor results set. Each open cursor is associated with a specific *@@rowcount* variable, which is dropped when you close the cursor. Check *@@rowcount* after a **fetch** to get the number of rows read for the cursor specified in that **fetch**.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The use of variables in a target list and fetch of multiple rows are Transact-SQL extensions.

Permissions

fetch permission defaults to all users.

See Also

Commands	**declare cursor, open, set**

goto Label

Function

Branches to a user-defined label.

Syntax

```
label:
   goto label
```

Examples

1.
```
declare @count smallint
select @count = 1
restart:
  print "yes"
select @count = @count + 1
while @count <=4
  goto restart
```

   Shows the use of a label called *restart*.

Comments

- The label name must conform to the rules for identifiers and must be followed by a colon (:) when it is declared. It is not followed by a colon when it is used with **goto**.
- Make the **goto** dependent on an **if** or **while** test, or some other condition, to avoid an endless loop between **goto** and the label.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

goto permission defaults to all users. No permission is required to use it.

See Also

Commands	**if...else**, **while**

grant

Function

Assigns permissions to users or to user-defined roles. Assigns roles to users or system or user-defined roles.

Syntax

To grant permission to access database objects:

```
grant {all [privileges]| permission_list}
   on { table_name [(column_list)]
       | view_name[(column_list)]
       | stored_procedure_name}
   to {public | name_list | role_name}
   [with grant option]
```

To grant permission to execute certain commands:

```
grant {all [privileges] | command_list}
   to {public | name_list | role_name}
```

To grant a role to a user or a role:

```
grant {role role_granted [, role_granted ...]}
   to grantee [, grantee...]
```

Keywords and Options

all – when used to assign permission to access database objects (the first syntax format), **all** specifies that all permissions applicable to the specified object are granted. All object owners can use **grant all** with an object name to grant permissions on their own objects.

Only a System Administrator or the Database Owner can assign permission to create database objects (the second syntax format). When used by a System Administrator, **grant all** assigns all **create** permissions (**create database**, **create default**, **create procedure**, **create rule**, **create table**, and **create view**). When the Database Owner uses **grant all**, Adaptive Server grants all **create** permissions except **create database**, and prints an informational message.

Specifying **all** does not include permission to execute **set proxy** or **set session authorization**.

permission_list – is a list of object access permissions granted. If more than one permission is listed, separate them with commas. The following table illustrates the access permissions that can be granted on each type of object:

Object	***permission_list* Can Include:**
Table	**select, insert, delete, update, references**
View	**select, insert, delete, update**
Column	**select, update, references** Column names can be specified in either *permission_list* or *column_list* (see example 2).
Stored procedure	**execute**

command_list – is a list of commands that the user can execute. If more than one command is listed, separate them with commas. The command list can include **create database**, **create default**, **create procedure**, **create rule**, **create table**, **create view**, **set proxy**, and **set session authorization.**

create database permission can be granted only by a System Administrator, and only from within the *master* database.

Only a System Security Officer can grant users permission to execute **set proxy** or **set session authorization**. Granting permission to execute **set proxy** or **set session authorization** allows the grantee to impersonate another login in the server. **set proxy** and **set session authorization** are identical, except that **set session authorization** follows the ANSI92 standard, and **set proxy** is a Transact-SQL extension.

table_name – is the name of the table on which you are granting permissions. The table must be in your current database. Only one object can be listed for each **grant** statement.

column_list – is a list of columns, separated by commas, to which the permissions apply. If columns are specified, only **select**, **references**, and **update** permissions can be granted.

view_name – is the name of the view on which you are granting permissions. The view must be in your current database. Only one object can be listed for each **grant** statement.

stored_procedure_name – is the name of the stored procedure on which you are granting permissions. The stored procedure must be in your current database. Only one object can be listed for each **grant** statement.

public – is all users. For object access permissions, **public** excludes the object owner. For object creation permissions or **set proxy** authorizations, **public** excludes the Database Owner. You cannot **grant** permissions **with grant option** to "public" or to other groups or roles.

name_list – is a list of users' database names and / or group names, separated by commas.

with grant option – allows the users specified in *name_list* to grant object access permissions to other users. You can grant permissions **with grant option** only to individual users, not to "public" or to a group or role.

role – grants a role to a user or to a system or user-defined role.

role_granted – is the name of a system or user-defined role that the System Security Officer is granting to a user or a role.

grantee – is the name of a system role, user-defined role, or a user, to whom you are granting a role.

role_name – is the name of a system or user-defined role to which you are granting the permission.

Examples

1.
```
grant insert, delete
on titles
to mary, sales
```
Grants Mary and the "sales" group permission to use the **insert** and **delete** commands on the *titles* table.

2.
```
grant update
on titles (price, advance)
to public
```
or:
```
grant update (price, advance)
on titles
to public
```
Two ways to grant **update** permission on the *price* and *advance* columns of the *titles* table to "public" (which includes all users).

3.
```
grant set proxy to harry, billy
```
Grants Harry and Billy permission to execute either **set proxy** or **set session authorization** to impersonate another user in the server.

4.
```
grant set session authorization to sso_role
```

Grants users with **sso_role** permission to execute either **set proxy** or **set session authorization** to impersonate another user in the server.

5.
```
grant set proxy to vip_role
```

Grants users with **vip_role** the ability to impersonate another user in the server. **vip_role** must be a role defined by a System Security Officer with the **create role** command.

6.
```
grant create database, create table
to mary, john
```

Grants Mary and John permission to use the **create database** and **create table** commands. Because **create database** permission is being granted, this command can be executed only by a System Administrator within the *master* database. Mary and John's **create table** permission applies only to the *master* database.

7.
```
grant all on titles
to public
```

Grants complete access permissions on the *titles* table to all users.

8.
```
grant all
to public
```

Grants all object creation permissions in the current database to all users. If this command is executed by a System Administrator from the *master* database, it includes **create database** permission.

9.
```
grant update on authors
to mary
with grant option
```

Gives Mary permission to use the **update** command on the *authors* table and to grant that permission to others.

10.
```
grant select, update on titles(price)
to bob
with grant option
```

Gives Bob permission to use the **select** and **update** commands on the *price* column of the *titles* table and to grant that permission to others.

11.
```
grant execute on new_sproc
to sso_role
```

Grants permission to execute the *new_sproc* stored procedure to all System Security Officers.

12. **`grant references on titles(price)`**
    `to james`

    Grants James permission to create a referential integrity constraint on another table that refers to the *price* column of the *titles* table.

13. **`grant role specialist_role to doctor_role`**

    Grants the role "specialist", with all its permissions and privileges, to the role "doctor".

14. **`grant role doctor_role to mary`**

    Grants the role "doctor" to Mary.

Comments

- You can substitute the word **from** for **to** in the **grant** syntax.
- Table 1-15 summarizes default permissions on Transact-SQL commands in Adaptive Server. The user listed under the "Defaults To" heading is the lowest level of user that is automatically granted permission to execute a command. This user can **grant** or **revoke** the permission if it is transferable. Users at higher levels than the default are either automatically assigned permission or (in the case of Database Owners) can get permission by using the **setuser** command.

  For example, the owner of a database does not automatically receive permission on objects owned by other users. A Database Owner can always gain such permission by assuming the identity of the object owner with the **setuser** command and then issuing the appropriate **grant** or **revoke** statement. System Administrators have permission to access all commands and objects at any time.

  The Adaptive Server installation script assigns a set of permissions to the default group "public." **grant** and **revoke** statements need not be written for these permissions.

Table 1-15 does not include the System Security Officer, who does not have any special permissions on commands and objects, but only on certain system procedures.

Table 1-15: Command and object permissions

Statement	Defaults To					Can Be Granted/Revoked		
	System Admin.	Operator	Database Owner	Object Owner	Public	Yes	No	N/A
alter database			•			(1)		
alter role								•
alter table				•			•	
begin transaction					•			•
checkpoint			•				•	
commit					•			•
create database	•					•		
create default			•			•		
create index				•			•	
create procedure			•			•		
create role								•
create rule			•			•		
create table			•		(2)	• (2)		
create trigger					•	•		
create view			•			•		
dbcc	Varies depending upon options. See **dbcc** in this manual.						•	
delete				• (3)		•		
disk init	•						•	
disk mirror	•							
disk refit	•							

(1) Transferred with database ownership
(2) Public can create temporary tables, no permission required
(3) If a view, permission defaults to view owner
(4) Defaults to stored procedure owner

(5) Transferred with **select** permission
(6) Transferred with **update** permission
"No" means use of the command is never restricted
"N/A" means use of the command is always restricted

Table 1-15: Command and object permissions (continued)

Statement	Defaults To					Can Be Granted/Revoked		
	System Admin.	Operator	Database Owner	Object Owner	Public	Yes	No	N/A
disk reinit	•							
disk remirror	•							
disk unmirror	•						•	
drop (any object)				•			•	
dump database		•	•				•	
dump transaction		•	•				•	
execute				•(4)		•		
grant on object				•		•		
grant command			•			•		
insert				• (3)		•		
kill	•						•	
load database		•	•				•	
load transaction		•	•				•	
print					•			•
raiserror					•			•
readtext				•		(5)		
revoke on object				•			•	
revoke command			•				•	
rollback					•			•
save transaction					•			•
select				• (3)		•		
set					•			•
setuser			•				•	
(1) Transferred with database ownership (2) Public can create temporary tables, no permission required (3) If a view, permission defaults to view owner (4) Defaults to stored procedure owner				(5) Transferred with **select** permission (6) Transferred with **update** permission "No" means use of the command is never restricted "N/A" means use of the command is always restricted				

Table 1-15: Command and object permissions (continued)

Statement	Defaults To					Can Be Granted/Revoked		
	System Admin.	Operator	Database Owner	Object Owner	Public	Yes	No	N/A
shutdown	•						•	
truncate table				•			•	
update				• (3)		•		
update all statistics				•			•	
update partition statistics				•			•	
update statistics				•			•	
writetext				•		(6)		
(1) Transferred with database ownership (2) Public can create temporary tables, no permission required (3) If a view, permission defaults to view owner (4) Defaults to stored procedure owner				(5) Transferred with **select** permission (6) Transferred with **update** permission “No” means use of the command is never restricted “N/A” means use of the command is always restricted				

- You can grant permissions only on objects in your current database.
- Before you create a table that includes a referential integrity constraint to reference another user’s table, you must be granted **references** permission on that referenced table (see example 10). The table must also include a unique constraint or unique index on the referenced columns. See **create table** for more information about referential integrity constraints.
- **grant** and **revoke** commands are order-sensitive. The command that takes effect when there is a conflict is the one issued most recently.
- A user can be granted permission on a view or stored procedure even if he or she has no permissions on objects referenced by the procedure or view. See “Granting Permissions on Views and Stored Procedures” in Chapter 4, “Granting Database Permissions,” of the *Security Features User’s Guide* for more information on using views and stored procedures as security mechanisms.
- Adaptive Server grants all users permission to declare cursors, regardless of the permissions defined for the base tables or views

referenced in the **declare cursor** statement. Cursors are not defined as Adaptive Server objects (such as tables), so no permissions can be applied against a cursor. When a user opens a cursor, Adaptive Server determines whether the user has **select** permissions on the objects that define that cursor's result set. It checks permissions each time a cursor is opened.

If the user has permission to access the objects defined by the cursor, Adaptive Server opens the cursor and allows the user to **fetch** row data through the cursor. Adaptive Server does not apply permission checking for each **fetch**. However, if the user performs a **delete** or an **update** through that cursor, the regular permission checking applies for deleting and updating the data of objects referenced in the cursor result set.

- A **grant** statement adds one row to the *sysprotects* system table for each user, group, or role that receives the permission. If you subsequently **revoke** the permission from the user or group, Adaptive Server removes the row from *sysprotects*. If you revoke the permission from selected group members only, but not from the entire group to which it was granted, Adaptive Server retains the original row and adds a new row for the revoke.
- If a user inherits a particular permission by virtue of being a member of a group, and the same permission is explicitly granted to the user, no row is added to *sysprotects*. For example, if "public" has been granted **select** permission on the *phone* column in the *authors* table, then John, a member of "public," is granted **select** permission on all columns of *authors*. The row added to *sysprotects* as a result of the **grant** to John will contain references to all columns in the *authors* table except for the *phone* column, on which he already had permission.
- Permission to issue the **create trigger** command is granted to users by default. When you revoke permission for a user to create triggers, a revoke row is added in the *sysprotects* table for that user. To grant permission to that user to issue **create trigger**, you must issue two **grant** commands. The first command removes the revoke row from *sysprotects*; the second inserts a grant row. If you revoke permission to create triggers, the user cannot create triggers even on tables that the user owns. Revoking permission to create triggers from a user affects only the database where the revoke command was issued.

- You can get information about permissions with these system procedures:
  - **sp_helprotect** reports permissions information for a database object or a user.
  - **sp_column_privileges** reports permissions information for one or more columns in a table or view.
  - **sp_table_privileges** reports permissions information for all columns in a table or view.
  - **sp_activeroles** displays all active roles for the current login session of Adaptive Server.
  - **sp_displayroles** displays all roles granted to another role, or displays the entire hierarchy tree of roles in table format.

grant all (Object Creation Permissions)

- When used with only user or group names (no object names), **grant all** assigns these permissions: **create database**, **create default**, **create procedure**, **create rule**, **create table**, and **create view. create database** permission can be granted only by a System Administrator and only from within the *master* database.
- Only the Database Owner and a System Administrator can use the **grant all** syntax without an object name to grant **create** command permissions to users or groups. When the **grant all** command is used by the Database Owner, an informational message is printed, stating that only a System Administrator can grant **create database** permission. All other permissions noted above are granted.
- All object owners can use **grant all** with an object name to grant permissions on their own objects. When used with a table or view name plus user or group names, **grant all** enables **delete**, **insert**, **select,** and **update** permissions on the table.

grant with grant option Rules

- You cannot grant permissions **with grant option** to “public” or to a group or role.
- In granting permissions, a System Administrator is treated as the object owner. If a System Administrator grants permission on another user’s object, the owner‘s name appears as the grantor in *sysprotects* and in **sp_helprotect** output.

- Information for each **grant** is kept in the system table *sysprotects* with the following exceptions:
  - Adaptive Server displays an informational message if a specific permission is granted to a user more than once by the same grantor. Only the first **grant** is kept.
  - If two **grant**s are exactly same except that one of them is granted **with grant option**, the **grant with grant option** is kept.
  - If two **grant** statements grant the same permissions on a particular table to a specific user, but the columns specified in the grants are different, Adaptive Server treats the grants as if they were one statement. For example, the following **grant** statements are equivalent:

```
grant select on titles(price, contract) to keiko
grant select on titles(advance) to keiko
```

```
grant select on titles(price, contract, advance)
to keiko
```

Granting Proxies and Session Authorizations

- Granting permission to execute **set proxy** or **set session authorization** allows the grantee to impersonate another login in Adaptive Server. **set proxy** and **set session authorization** are identical with one exception: **set session authorization** follows the SQL standard, and **set proxy** is a Transact-SQL extension.
- To grant **set proxy** or **set session authorization** permission, you must be a System Security Officer, and you must be in the *master* database.
- The name you specify in the **grant set proxy** command must be a valid user in the database; that is, the name must be in the *sysusers* table in the database.
- **grant all** does **not** include the **set proxy** or **set session authorization** permissions.

Granting Permission to Roles

- You can use the **grant** command to grant permissions to all users who have been granted a specified role. The role can be either a system role, like **sso_role** or **sa_role**, or a user-defined role. For a user-defined role, the System Security Officer must create the role with a **create role** command.

  However, **grant execute** permission does not prevent users who do not have a specified role from being individually granted permission to execute a stored procedure. If you want to ensure, for example, that only System Security Officers can ever be granted permission to execute a stored procedure, use the **proc_role** system function within the stored procedure itself. It checks to see whether the invoking user has the correct role to execute the procedure. See **proc_role** for more information.

- Permissions that are granted to roles override permissions that are granted to users or groups. For example, say John has been granted the System Security Officer role, and **sso_role** has been granted permission on the *sales* table. If John's individual permission on *sales* is revoked, he can still access *sales* because his role permissions override his individual permissions.

Users and User Groups

- User groups allow you to **grant** or **revoke** permissions to more than one user with a single statement. Each user can be a member of only one group and is always a member of "public."
- The Database Owner or System Administrator can add new users with **sp_adduser** and create groups with **sp_addgroup**. To allow users with logins on Adaptive Server to use the database with limited privileges, you can add a "guest" user with **sp_adduser** and assign limited permissions to "guest". All users with logins can access the database as "guest".
- To remove a user, use **sp_dropuser**. To remove a group, use **sp_dropgroup**.

  To add a new user to a group other than "public," use **sp_adduser.** To change an established user's group, use **sp_changegroup**.

  To display the members of a group, use **sp_helpgroup**.
- When **sp_changegroup** is executed to change group membership, it clears the in-memory protection cache by executing:

  ```
  grant all to null
  ```

  so that the cache can be refreshed with updated information from the *sysprotects* table. If you need to modify *sysprotects* directly, contact Sybase Technical Support.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	Granting permissions to groups and granting **set proxy** are Transact-SQL extensions. Granting **set session authorization** (identical in function to **set proxy**) follows the ANSI standard.

Permissions

Database Object Access Permissions

grant permission for database objects defaults to object owners. An object owner can grant permission to other users on his or her own database objects.

Command Execution Permissions

Only a System Administrator can grant **create database** permission, and only from the *master* database. Only a System Security Officer can grant **create trigger** permission.

Proxy and Session Authorization Permissions

Only a System Security Officer can grant **set proxy** or **set session authorization**, and only from the *master* database.

Role Permissions

You can grant roles only from the *master* database. Only a System Security Officer can grant **sso_role**, **oper_role** or a user-defined role to a user or a role. Only System Administrators can grant **sa_role** to a user or a role. Only a user who has both **sa_role** and **sso_role** can grant a role which includes **sa_role**.

See Also

Catalog stored procedures	**sp_column_privileges**
Commands	**revoke, setuser, set**
Functions	**proc_role**
System procedures	**sp_addgroup, sp_adduser, sp_changedbowner, sp_changegroup, sp_dropgroup, sp_dropuser, sp_helpgroup, sp_helprotect, sp_helpuser, sp_role**

group by and having Clauses

Function

Used in **select** statements to divide a table into groups and to return only groups that match conditions in the **having** clause.

Syntax

```
Start of select statement
[group by [all] aggregate_free_expression
   [, aggregate_free_expression]...]
[having search_conditions]
End of select statement
```

Keywords and Options

group by – specifies the groups into which the table will be divided, and if aggregate functions are included in the select list, finds a summary value for each group. These summary values appear as columns in the results, one for each group. You can refer to these summary columns in the **having** clause.

You can use the **avg**, **count**, **max**, **min**, and **sum** aggregate functions in the select list before **group by** (the expression is usually a column name). For more information, see "Aggregate Functions" in Chapter 2, "Transact-SQL Functions".

A table can be grouped by any combination of columns—that is, groups can be nested within each other, as in example 2.

all – is a Transact-SQL extension that includes all groups in the results, even those excluded by a **where** clause. For example:

```
select type, avg(price)
from titles
where advance > 7000
group by all type
```

```
type
-----------------  ----------
UNDECIDED                NULL
business                 2.99
mod_cook                 2.99
popular_comp            20.00
psychology               NULL
trad_cook               14.99

(6 rows affected)
```

"NULL" in the aggregate column indicates groups that would be excluded by the **where** clause. A **having** clause negates the meaning of **all**.

aggregate_free_expression – is an expression that includes no aggregates. A Transact-SQL extension allows grouping by an aggregate-free expression as well as by a column name.

You cannot group by column heading or alias. This example is correct:

```
select Price=avg(price), Pay=avg(advance),
Total=price * $1.15
from titles
group by price * $1.15
```

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

having search conditions can include aggregate expressions; otherwise, **having** search conditions are identical to **where** search conditions. Following is an example of a **having** clause with aggregates:

```
select pub_id, total = sum(total_sales)
from titles
where total_sales is not null
group by pub_id
having count(*)>5
```

When Adaptive Server optimizes queries, it evaluates the search conditions in **where** and **having** clauses, and determines which conditions are search arguments (SARGs) that can be used to choose the best indexes and query plan. For each table in a query, a maximum of 128 search arguments can be used to optimize the query. All of the search conditions, however, are used to qualify the rows. For more information on search arguments, see Chapter 8, "Search Arguments and Using Indexes," in the *Performance and Tuning Guide.*

Examples

1.
```
select type, avg(advance), sum(total_sales)
from titles
group by type
```
   Calculates the average advance and the sum of the sales for each type of book.

2.
```
select type, pub_id, avg(advance), sum(total_sales)
from titles
group by type, pub_id
```
   Groups the results by type and then by *pub_id* within each type.

3.
```
select type, avg(price)
from titles
group by type
having type like 'p%'
```
   Calculates results for all groups, but displays only groups whose type begins with "p".

4.
```
select pub_id, sum(advance), avg(price)
from titles
group by pub_id
having sum(advance) > $15000
and avg(price) < $10
and pub_id > "0700"
```
   Calculates results for all groups, but displays results for groups matching the mulitple conditions in the **having** clause.

5.
```
select p.pub_id, sum(t.total_sales)
from publishers p, titles t
where p.pub_id = t.pub_id
group by p.pub_id
```
   Calculates the total sales for each group (publisher) after joining the *titles* and *publishers* tables.

6.
```
select title_id, advance, price
from titles
where advance > 1000
having price > avg(price)
```
   Displays the titles that have an advance of more than $1000 and a price that is more than the average price of all titles.

Comments

- You can use a column name or any expression (except a column heading or alias) after **group by.** You can use **group by** to calculate results or display a column or an expression that does not appear in the select list (a Transact-SQL extension described in "Transact-SQL Extensions to group by and having").

- The maximum number of columns or expressions allowed in a **group by** clause is 16.
- The sum of the maximum lengths of all the columns specified by the **group by** clause cannot exceed 256 bytes.
- Null values in the **group by** column are put into a single group.
- You cannot name *text* or *image* columns in **group by** and **having** clauses.
- You cannot use a **group by** clause in the **select** statement of an updatable cursor.
- Aggregate functions can be used only in the select list or in a **having** clause. They cannot be used in a **where** or **group by** clause.

  Aggregate functions are of two types. Aggregates applied to **all the qualifying rows in a table** (producing a single value for the whole table per function) are called **scalar aggregates**. An aggregate function in the select list with no **group by** clause applies to the whole table; it is one example of a scalar aggregate.

  Aggregates applied to **a group of rows in a specified column or expression** (producing a value for each group per function) are called **vector aggregates**. For either aggregate type, the results of the aggregate operations are shown as new columns that the **having** clause can refer to.

  You can nest a vector aggregate inside a scalar aggregate. See "Aggregate Functions" in Chapter 2, "Transact-SQL Functions" for more information.

How *group by* and *having* Queries with Aggregates Work

- The **where** clause excludes rows that do not meet its search conditions; its function remains the same for grouped or non-grouped queries.
- The **group by** clause collects the remaining rows into one group for each unique value in the **group by** expression. Omitting **group by** creates a single group for the whole table.
- Aggregate functions specified in the select list calculate summary values for each group. For scalar aggregates, there is only one value for the table. Vector aggregates calculate values for the distinct groups.

- The **having** clause excludes groups from the results that do not meet its search conditions. Even though the **having** clause tests only rows, the presence or absence of a **group by** clause may make it appear to be operating on groups:
  - When the query includes **group by, having** excludes result group rows. This is why **having** seems to operate on groups.
  - When the query has no **group by, having** excludes result rows from the (single-group) table. This is why **having** seems to operate on rows (the results are similar to **where** clause results).

Standard *group by* and *having* Queries

- All **group by** and **having** queries in the "Examples" section adhere to the SQL standard. It dictates that queries using **group by, having**, and vector aggregate functions produce one row and one summary value per group, using these guidelines:
  - Columns in a select list must also be in the **group by** expression, or they must be arguments of aggregate functions.
  - A **group by** expression can contain only column names that are in the select list. However, columns used only as arguments of aggregate functions in the select list do not qualify.
  - Columns in a **having** expression must be single-valued—arguments of aggregates, for instance—and they must be in the select list or **group by** clause. Queries with a select list aggregate and a **having** clause **must** have a **group by** clause. If you omit the **group by** for a query without a select list aggregate, all the rows not excluded by the **where** clause are considered to be a single group (see example 6).

  In non-grouped queries, the principle that "**where** excludes rows" seems straightforward. In grouped queries, the principle expands to "**where** excludes rows before **group by**, and **having** excludes rows from the display of results."
- The SQL standard allows queries that join two or more tables to use **group by** and **having**, if they also adhere to the above guidelines. When specifying joins or other complex queries, use the standard syntax of **group by** and **having** until you fully comprehend the effect of the Transact-SQL extensions to both clauses, as described in "Transact-SQL Extensions to group by and having."

  To help you avoid problems with extensions, Adaptive Server provides the **fipsflagger** option to the **set** command that issues a non-fatal warning for each occurrence of a Transact-SQL extension in a query. See **set** for more information.

Transact-SQL Extensions to *group by* and *having*

- Transact-SQL extensions to standard SQL make displaying data more flexible, by allowing references to columns and expressions that are not used for creating groups or summary calculations:
  - A select list that includes aggregates can include **extended** columns that are not arguments of aggregate functions and are not included in the **group by** clause. An extended column affects the display of final results, since additional rows are displayed.
  - The **group by** clause can include columns or expressions that are not in the select list.
  - The **group by all** clause displays all groups, even those excluded from calculations by a **where** clause. See the example for the keyword **all** in the "Keywords and Options" section.
  - The **having** clause can include columns or expressions that are not in the select list and not in the **group by** clause.

  When the Transact-SQL extensions add rows and columns to a display, or if **group by** is omitted, query results can be hard to interpret. The examples that follow can help you understand how Transact-SQL extensions can affect query results.
- The following examples illustrate the differences between queries that use standard **group by** and **having** clauses and queries that use the Transact-SQL extensions:

1.
```
select type, avg(price)
from titles
group by type
```

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

(6 rows affected)
```

An example of a standard grouping query.

2. **select type, price, avg(price)**
   from titles
   group by type

```
type         price
------------ ---------------- ----------------
business                19.99            13.73
business                11.95            13.73
business                 2.99            13.73
business                19.99            13.73
mod_cook                19.99            11.49
mod_cook                 2.99            11.49
UNDECIDED                NULL             NULL
popular_comp            22.95            21.48
popular_comp            20.00            21.48
popular_comp             NULL            21.48
psychology              21.59            13.50
psychology              10.95            13.50
psychology               7.00            13.50
psychology              19.99            13.50
psychology               7.99            13.50
trad_cook               20.95            15.96
trad_cook               11.95            15.96
trad_cook               14.99            15.96

(18 rows affected)
```

The Transact-SQL extended column, *price* (in the select list, but not an aggregate and not in the **group by** clause), causes all qualified rows to display in each qualified group, even though a standard **group by** clause produces a single row per group. The **group by** still affects the vector aggregate, which computes the average price per group displayed on each row of each group (they are the same values that were computed for example 1).

3. **select type, price, avg(price)**
   from titles
   where price > 10.00
   group by type

```
type         price
------------ ---------------- ----------------
business                19.99            17.31
business                11.95            17.31
business                 2.99            17.31
business                19.99            17.31
mod_cook                19.99            19.99
mod_cook                 2.99            19.99
popular_comp            22.95            21.48
popular_comp            20.00            21.48
popular_comp             NULL            21.48
psychology              21.59            17.51
psychology              10.95            17.51
psychology               7.00            17.51
psychology              19.99            17.51
psychology               7.99            17.51
trad_cook               20.95            15.96
trad_cook               11.95            15.96
trad_cook               14.99            15.96

(17 rows affected)
```

The way Transact-SQL extended columns are handled can make it look as if a query is ignoring a **where** clause. This query computes the average prices using only those rows that satisfy the **where** clause, but it also displays rows that do not match the **where** clause.

Adaptive Server first builds a worktable containing only the type and aggregate values using the **where** clause. This worktable is joined back to the *titles* table in the grouping column *type* to include the *price* column in the results, but the **where** clause is **not** used in the join.

The only row in *titles* that is not in the results is the lone row with *type* = "UNDECIDED" and a NULL price, that is, a row for which there were no results in the worktable. If you also want to eliminate the rows from the displayed results that have prices of less than $10.00, you must add a **having** clause that repeats the **where** clause, as shown in example 4.

4. **`select type, price, avg(price)`**
   `from titles`
   `where price > 10.00`
   `group by type`
   `having price > 10.00`

```
type        price
----------- ----------------  ----------------
business                19.99            17.31
business                11.95            17.31
business                19.99            17.31
mod_cook                19.99            19.99
popular_comp            22.95            21.48
popular_comp            20.00            21.48
psychology              21.59            17.51
psychology              10.95            17.51
psychology              19.99            17.51
trad_cook               20.95            15.96
trad_cook               11.95            15.96
trad_cook               14.99            15.96

(12 rows affected)
```

   If you are specifying additional conditions, such as aggregates, in the **having** clause, be sure to also include all conditions specified in the **where** clause. Adaptive Server will appear to ignore any **where** clause conditions that are missing from the **having** clause.

5. **`select p.pub_id, t.type, sum(t.total_sales)`**
   `from publishers p, titles t`
   `where p.pub_id = t.pub_id`
   `group by p.pub_id, t.type`

```
pub_id  type
------  ------------  --------
0736    business         18722
0736    psychology        9564
0877    UNDECIDED         NULL
0877    mod_cook         24278
0877    psychology         375
0877    trad_cook        19566
1389    business         12066
1389    popular_comp     12875

(8 rows affected)
```

   This is an example of a standard grouping query using a join between two tables. It groups by *pub_id* and then by *type* within each publisher ID, to calculate the vector aggregate for each row.

It may seem that it is only necessary to specify **group by** for the *pub_id* and *type* columns to produce the results, and add extended columns as follows:

```
select p.pub_id, p.pub_name, t.type,
    sum(t.total_sales)
from publishers p, titles t
where p.pub_id = t.pub_id
group by p.pub_id, t.type
```

However, the results for the above query are much different from the results for the first query in this example. After joining the two tables to determine the vector aggregate in a worktable, Adaptive Server joins the worktable to the table (*publishers*) of the extended column for the final results. Each extended column from a different table invokes an additional join.

As you can see, using the extended column extension in queries that join tables can easily produce results that are difficult to comprehend. In most cases, you should use the standard **group by** to join tables in your queries.

6.
```
select p.pub_id, sum(t.total_sales)
from publishers p, titles t
where p.pub_id = t.pub_id
group by p.pub_id, t.type
```

```
pub_id
------  --------
0736       18722
0736        9564
0877        NULL
0877       24278
0877         375
0877       19566
1389       12066
1389       12875

(8 rows affected)
```

This example uses the Transact-SQL extension to **group by** to include columns that are not in the select list. Both the *pub_id* and *type* columns are used to group the results for the vector aggregate. However, the final results do not include the type within each publisher. In this case, you may only want to know how many distinct title types are sold for each publisher.

7. **`select pub_id, count(pub_id)`**
   `from publishers`

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

(3 rows affected)
```

   This example combines two Transact-SQL extension effects. First, it omits the **group by** clause while including an aggregate in the select list. Second, it includes an extended column. By omitting the **group by** clause:

   - The table becomes a single group. The scalar aggregate counts three qualified rows.
   - *pub_id* becomes a Transact-SQL extended column because it does not appear in a **group by** clause. No **having** clause is present, so all rows in the group are qualified to be displayed.

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

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

(3 rows affected)
```

   The **where** clause excludes publishers with a *pub_id* of 1000 or more from the single group, so the scalar aggregate counts two qualified rows. The extended column *pub_id* displays all qualified rows from the *publishers* table.

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

```
pub_id
-------------- ---------
0736                   3
0877                   3
(2 rows affected)
```

This example illustrates an effect of a **having** clause used without a **group by** clause.

- The table is considered a single group. No **where** clause excludes rows, so all the rows in the group (table) are qualified to be counted.
- The rows in this single-group table are tested by the **having** clause.
- These combined effects display the two qualified rows.

10.
```
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)
```

This example uses the extension to **having** that allows columns or expressions not in the select list and not in the **group by** clause. 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.

group by and *having* and Sort Orders

- If your server has a case-insensitive sort order, **group by** ignores the case of the grouping columns. For example, given this data on a case-insensitive server:

```
select lname, amount
from groupdemo
```

```
lname      amount
---------- ------------------
Smith                   10.00
smith                    5.00
SMITH                    7.00
Levi                     9.00
Lévi                    20.00
```

grouping by *lname* produces these results:

```
select lname, sum(amount)
from groupdemo
group by lname
```

```
lname
---------- -----------------
Levi                    9.00
Lévi                   20.00
Smith                  22.00
```

The same query on a case- and accent-insensitive server produces these results:

```
 lname
 ---------- -----------------
 Levi                    29.00
 Smith                   22.00
```

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The use of columns within the **select** list that are not in the **group by** list and have no aggregate functions is a Transact-SQL extension. The use of the **all** keyword is a Transact-SQL extension.

See Also

Commands	**compute Clause**, **declare**, **select**, **where Clause**
Functions	Aggregate Functions

if...else

Function

Imposes conditions on the execution of a SQL statement.

Syntax

```
if logical_expression
   statements
[else
   [if logical_expression]
   statement]
```

Keywords and Options

logical_expression – is an expression (a column name, a constant, any combination of column names and constants connected by arithmetic or bitwise operators, or a subquery) that returns TRUE, FALSE, or NULL. If the expression contains a select statement, the select statement must be enclosed in parentheses.

statements – is either a single SQL statement or a block of statements delimited by **begin** and **end**.

Examples

1.
```
if 3 > 2
 print "yes"
```

2.
```
if exists (select postalcode from authors
 where postalcode = "94705")
 print "Berkeley author"
```

3.
```
if (select max(id) from sysobjects) < 100
   print "No user-created objects in this
database" else
 begin
   print "These are the user-created objects"
   select name, type, id
   from sysobjects
   where id > 100
 end
```

The **if...else** condition tests for the presence of user-created objects (all of which have ID numbers greater than 100) in a database. Where user tables exist, the **else** clause prints a message and selects their names, types, and ID numbers.

4.
```
if (select total_sales
    from titles
    where title_id = "PC9999") > 100
select "true"
else
select "false"
```

Since the value for total sales for PC9999 in the *titles* table is NULL, this query returns FALSE. The **else** portion of the query is performed when the **if** portion returns FALSE or NULL. See "Expressions" in Appendix A, "Expressions, Identifiers, and Wildcard Characters," for more information on truth values and logical expressions.

Comments

- The statement following an **if** keyword and its condition is executed if the condition is satisfied (when the logical expression returns TRUE). The optional **else** keyword introduces an alternate SQL statement that executes when the **if** condition is not satisfied (when the logical expression returns FALSE).
- The **if** or **else** condition affects the performance of only a single SQL statement, unless statements are grouped into a block between the keywords **begin** and **end** (see example 3).

  The statement clause could be an **execute** stored procedure command or any other legal SQL statement or statement block.
- If a **select** statement is used as part of the boolean expression, it must return a single value.
- **if...else** constructs can be used either in a stored procedure (where they are often used to test for the existence of some parameter) or in *ad hoc* queries (see examples 1 and 2).
- **if** tests can be nested either within another **if** or following an **else**. The maximum number of **if** tests you can nest varies with the complexity of any **select** statements (or other language constructs) that you include with each **if...else** construct.

➤ ***Note***

When a **create table** or **create view** command occurs within an **if...else** block, Adaptive Server creates the schema for the table or view before determining whether the condition is true. This may lead to errors if the table or view already exists.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

if...else permission defaults to all users. No permission is required to use it.

See Also

Commands	**begin...end**, **create procedure**

insert

Function

Adds new rows to a table or view.

Syntax

```
insert [into]
   [database.[owner.]]{table_name|view_name}
   [(column_list)]
   {values (expression [, expression]...)
       |select_statement }
```

Keywords and Options

into – is optional.

table_name | *view_name* – is the name of the table or view from which you want to remove rows. Specify the database name if the table or view is in another database, and specify the owner's name if more than one table or view of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

column_list – is a list of one or more columns to which data is to be added. Enclose the list in parentheses. The columns can be listed in any order, but the incoming data (whether in a **values** clause or a **select** clause) must be in the same order. If a column has the IDENTITY property, you can substitute the **syb_identity** keyword for the actual column name.

The column list is necessary when some, but not all, of the columns in the table are to receive data. If no column list is given, Adaptive Server assumes that the **insert** affects all columns in the receiving table (in **create table** order).

See "The Column List" for more information.

values – is a keyword that introduces a list of expressions.

expression – specifies constant expressions, variables, parameters or null values for the indicated columns. Enclose character and datetime constants in single or double quotes.

You cannot use a subquery as an *expression*.

The values list must be enclosed in parentheses and must match the explicit or implicit column list. See “Datatypes” for more information about data entry rules.

select_statement – is a standard **select** statement used to retrieve the values to be inserted.

Examples

1.
```
insert titles
values("BU2222", "Faster!", "business", "1389",
    null, null, null, "ok", "06/17/87", 0)
```

2.
```
insert titles
(title_id, title, type, pub_id, notes, pubdate,
    contract)
values ('BU1237', 'Get Going!', 'business',
     '1389', 'great', '06/18/86', 1)
```

3.
```
insert newauthors
  select *
  from authors
  where city = "San Francisco"
```

4.
```
insert test
  select *
  from test
  where city = "San Francisco"
```

Comments

- Use **insert** only to add new rows. Use **update** to modify column values in a row you have already inserted.

The Column List

- The column list determines the order in which values are entered. For example, suppose that you have a table called *newpublishers* that is identical in structure and content to the *publishers* table in *pubs2*. In the example below, the columns in the column list of the *newpublishers* table match the columns of the select list in the *publishers* table.

```
insert newpublishers (pub_id, pub_name)
select pub_id, pub_name
    from publishers
    where pub_name="New Age Data"
```

The *pub_id* and *pub_name* for “New Age Data” are stored in the *pub_id* and *pub_name* columns of *newpublishers*.

In the next example, the order of the columns in the column list of the *newpublishers* table does not match the order of the columns of the select list of the *publishers* table.

```
insert newpublishers (pub_id, pub_name)
    select pub_name, pub_id
    from publishers
    where pub_name="New Age Data"
```

The result is that the *pub_id* for "New Age Data" is stored in the *pub_name* column of the *newpublishers* table, and the *pub_name* for "New Age Data" is stored in the *pub_id* column of the *newpublishers* table.

- You can omit items from the column and values lists as long as the omitted columns allow null values (see example 2).

Validating Column Values

- **insert** interacts with the **ignore_dup_key**, **ignore_dup_row**, and **allow_dup_row** options, which are set with the **create index** command. (See **create index** for more information.)
- A rule or **check** constraint can restrict the domain of legal values that can be entered into a column. Rules are created with the **create rule** command and bound with the system procedure **sp_bindrule**. **check** constraints are declared with the **create table** statement.
- A default can supply a value if you do not explicitly enter one. Defaults are created with the **create default** command and bound with the system procedure **sp_bindefault**, or they are declared with the **create table** statement.
- If an **insert** statement violates domain or integrity rules (see **create rule** and **create trigger**), or if it is the wrong datatype (see **create table** and "System and User-Defined Datatypes"), the statement fails, and Adaptive Server displays an error message.

Treatment of Blanks

- Inserting an empty string ("") into a variable character type or *text* column inserts a single space. *char* columns are padded to the defined length.
- All trailing spaces are removed from data that is inserted into *varchar* columns, except in the case of a string that contains only spaces. Strings that contain only spaces are truncated to a single space. Strings that are longer than the specified length of a *char*,

nchar, *varchar*, or *nvarchar* column are silently truncated unless the **string_rtruncation** option is set to **on**.

Inserting into *text* and *image* Columns

- An **insert** of a NULL into a *text* or an *image* column does not create a valid text pointer, nor does it preallocate 2K per value as would otherwise occur. Use **update** to get a valid text pointer for that column.

Insert Triggers

- You can define a trigger that takes a specified action when an **insert** command is issued on a specified table.

Using *insert* When Component Integration Services Is Enabled

- You can send an **insert** as a language event or as a parameterized dynamic statement to remote servers.

Inserting Rows Selected from Another Table

- You can select rows from a table and insert them into the same table in a single statement (see example 4).
- 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, substitute 0 for the NULL fields:

  ```
  insert advances
  select pub_id, isnull(advance, 0) from titles
  ```

  Without the **isnull** function, this command would insert all the rows with non-null values into the *advances* table, which would produce error messages for all the rows where the *advance* column in the *titles* table contained NULL.

  If you cannot make this kind of substitution for your data, you cannot insert data containing null values into the columns that have a NOT NULL specification.

  Two tables can be identically structured, and yet be different as to whether null values are permitted in some fields. You can use **sp_help** to see the null types of the columns in your table.

Transactions and *insert*

- When you set chained transaction mode, Adaptive Server implicitly begins a transaction with the **insert** statement if no transaction is currently active. To complete any inserts, you must commit the transaction, or roll back the changes. For example:

  ```
  insert stores (stor_id, stor_name, city, state)
      values ('9999', 'Books-R-Us', 'Fremont', 'AZ')
  if exists (select t1.city, t2.city
      from stores t1, stores t2
      where t1.city = t2.city
      and t1.state = t2.state
      and t1.stor_id < t2.stor_id)
          rollback transaction
  else
          commit transaction
  ```

  In chained transaction mode, this batch begins a transaction and inserts a new row into the *stores* table. If it inserts a row containing the same city and state information as another store in the table, it rolls back the changes to *stores* and ends the transaction. Otherwise, it commits the insertions and ends the transaction. For more information about chained transaction mode, see Chapter 18, "Transactions: Maintaining Data Consistency and Recovery," in the *Transact-SQL User's Guide*.

Inserting Values into IDENTITY Columns

- When inserting a row into a table, do not include the name of the IDENTITY column in the column list or its value in the values list. If the table consists of only one column, an IDENTITY column, omit the column list and leave the values list empty as follows:

  ```
  insert id_table values()
  ```

- The first time you insert a row into a table, Adaptive Server assigns the IDENTITY column a value of 1. Each new row gets a column value that is one higher than the last. This value takes precedence over any defaults declared for the column in the **create table** or **alter table** statement or defaults bound to the column with the **sp_bindefault** system procedure.

  Server failures can create gaps in IDENTITY column values. The maximum size of the gap depends on the setting of the **identity burning set factor** configuration parameter. Gaps can also result from manual insertion of data into the IDENTITY column, deletion of rows, and transaction rollbacks.

- Only the table owner, Database Owner, or System Administrator can explicitly insert a value into an IDENTITY column after setting **identity_insert** *table_name* **on** for the column's base table. A user can set **identity_insert** *table_name* **on** for one table at a time in a database. When **identity_insert** is **on**, each **insert** statement must include a column list and must specify an explicit value for the IDENTITY column.

  Inserting a value into the IDENTITY column allows you to specify a seed value for the column or to restore a row that was deleted in error. Unless you have created a unique index on the IDENTITY column, Adaptive Server does not verify the uniqueness of the value; you can insert any positive integer.

  To insert an explicit value into an IDENTITY column, the table owner, Database Owner, or System Administrator must set **identity_insert** *table_name* **on** for the column's base table, not for the view through which it is being inserted.

- The maximum value that can be inserted into an IDENTITY column is $10^{\text{PRECISION}} - 1$. Once an IDENTITY column reaches this value, any additional **insert** statements return an error that aborts the current transaction.

  When this happens, use the **create table** statement to create a new table that is identical to the old one, but that has a larger precision for the IDENTITY column. Once you have created the new table, use either the **insert** statement or the **bcp** utility to copy the data from the old table to the new one.

- Use the *@@identity* global variable to retrieve the last value that you inserted into an IDENTITY column. If the last **insert** or **select into** statement affected a table with no IDENTITY column, *@@identity* returns the value 0.

- An IDENTITY column selected into a result table observes the following rules with regard to inheritance of the IDENTITY property:

  - 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.

  - If an IDENTITY column is selected as part of an expression, the resulting column does not inherit the IDENTITY property. It is created as NULL if any column in the expression allows nulls; otherwise, it is created as NOT NULL.

- If the select statement contains a **group by** clause or aggregate function, the resulting column does not inherit the IDENTITY property. Columns that include an aggregate of the IDENTITY column are created NULL; others are created NOT NULL.
- 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 defined as NOT NULL.

Inserting Data Through Views

- If a view is created **with check option**, each row that is inserted through the view must meet the selection criteria of the view.

  For example, the *stores_cal* view includes all rows of the *stores* table for which *state* has a value of "CA":

  ```
  create view stores_cal
  as select * from stores
  where state = "CA"
  with check option
  ```

  The **with check option** clause checks each **insert** statement against the view's selection criteria. Rows for which *state* has a value other than "CA" are rejected.

- If a view is created **with check option**, all views derived from the **base** view must satisfy the view's selection criteria. Each new row inserted through a derived view must be visible through the base view.

  Consider the view *stores_cal30*, which is derived from *stores_cal*. The new view includes information about stores in California with payment terms of "Net 30":

  ```
  create view stores_cal30
  as select * from stores_cal
  where payterms = "Net 30"
  ```

  Because *stores_cal* was created **with check option**, all rows inserted or updated through *stores_cal30* must be visible through *stores_cal*. Any row with a *state* value other than "CA" is rejected.

  Notice that *stores_cal30* does not have a **with check option** clause of its own. This means that it is possible to insert or update a row with a *payterms* value other than "Net 30" through *stores_cal30*.

The following **update** statement would be successful, even though the row would no longer be visible through *stores_cal30*:

```
update stores_cal30
set payterms = "Net 60"
where stor_id = "7067"
```

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

Partitioning Tables for Improved Insert Performance

- An unpartitioned table with no clustered index consists of a single doubly linked chain of database pages, so each insertion into the table uses the last page of the chain. Adaptive Server holds an exclusive lock on the last page while it inserts the rows, blocking other concurrent transactions from inserting data into the table.

  Partitioning a table with the **partition** clause of the **alter table** command creates additional page chains. Each chain has its own last page, which can be used for concurrent insert operations. This improves insert performance by reducing page contention. If the table is spread over multiple physical devices, partitioning also improves insert performance by reducing I/O contention while the server flushes data from cache to disk.

  For more information about partitioning tables for insert performance, see “Partitioning Tables for Performance” in Chapter 17, “Controlling Physical Data Placement,” in the *Performance and Tuning Guide.*

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The following are Transact-SQL extensions: • A **union** operator in the select portion of an **insert** statement • Qualification of a table or column name by a database name • Insertion through a view that contains a join **Note:** Insertion through a view that contains a join is not detected by the FIPS flagger.)

Permissions

insert permission defaults to the table or view owner, who can transfer it to other users.

insert permission for a table's IDENTITY column is limited to the table owner, Database Owner, and System Administrator.

See Also

Commands	**alter table, create default, create index, create rule, create table, create trigger, dbcc, delete, select, update**
Datatypes	"System and User-Defined Datatypes"
System procedures	**sp_bindefault, sp_bindrule, sp_help, sp_helppartition, sp_unbindefault, sp_unbindrule**
Utility programs	**bcp**

kill

Function

Kills a process.

Syntax

```
kill spid
```

Keywords and Options

spid – is the identification number of the process you want to kill. *spid* must be a constant; it cannot be passed as a parameter to a stored procedure or used as a local variable. Use **sp_who** to see a list of processes and other information.

Examples

1. **`kill 1378`**

Comments

- To get a report on the current processes, execute the system procedure **sp_who**. Following is a typical report:

```
fid spid  status    loginame origname hostname blk  dbname
       cmd
--- ----- --------   -------- -------- -------- --- ------
       --------------
  0    1  recv sleep  bird     bird     jazzy    0   master
       AWAITING COMMAND
  0    2  sleeping    NULL     NULL              0   master
       NETWORK HANDLER
  0    3  sleeping    NULL     NULL              0   master
       MIRROR HANDLER
  0    4  sleeping    NULL     NULL              0   master
       AUDIT PROCESS
  0    5  sleeping    NULL     NULL              0   master
       CHECKPOINT SLEEP
```

```
0    6  recv sleep  rose     rose     petal    0   master
     AWAITING COMMAND
0    7  running     robert   sa       helos    0   master
     SELECT
0    8  send sleep  daisy    daisy    chain    0   pubs2
     SELECT
0    9  alarm sleep lily     lily     pond     0   master
     WAITFOR
0   10  lock sleep  viola    viola    cello    7   pubs2
     SELECT
```

The *spid* column contains the process identification numbers used in the Transact-SQL **kill** command. The *blk* column contains the process ID of a blocking process, if there is one. A blocking process (which may have an exclusive lock) is one that is holding resources that are needed by another process. In this example, process 10 (a **select** on a table) is blocked by process 7 (a **begin transaction** followed by an **insert** on the same table).

- The *status* column reports the state of the command. The following table shows the status values and the effects of **sp_who**:

Table 1-16: Status values reported by sp_who

Status		Effect of *kill* Command
`recv sleep`	Waiting on a network read	Immediate.
`send sleep`	Waiting on a network send	Immediate.
`alarm sleep`	Waiting on an alarm, such as **waitfor delay "10:00"**	Immediate.
`lock sleep`	Waiting on a lock acquisition	Immediate.
`sleeping`	Waiting on disk I/O or some other resource. Probably indicates a process that is running, but doing extensive disk I/O	Killed when it "wakes up", usually immediate. A few sleeping processes do not wake up, and require a Adaptive Server reboot to clear.
`runnable`	In the queue of runnable processes	Immediate.
`running`	Actively running on one of the server engines	Immediate.
`infected`	Adaptive Server has detected a serious error condition; extremely rare	**kill** command not recommended. Adaptive Server restart probably required to clear process.

Table 1-16: Status values reported by sp_who (continued)

Status		Effect of *kill* Command
`background`	A process, such as a threshold procedure, run by Adaptive Server rather than by a user process	Immediate; use **kill** with extreme care. Recommend a careful check of *sysprocesses* before killing a background process.
`log suspend`	Processes suspended by reaching the last-chance threshold on the log	Immediate.

- To get a report on the current locks and the *spid*s of the processes holding them, use **sp_lock**.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

kill permission defaults to System Administrators and is not transferable.

See Also

Commands	**shutdown**
System procedures	**sp_lock, sp_who**

load database

Function

Loads a backup copy of a user database, including its transaction log, that was created with **dump database**.

Syntax

```
load database database_name
   from stripe_device [at backup_server_name ]
       [density = density_value,
       blocksize = number_bytes,
       dumpvolume = volume_name,
       file = file_name]
   [stripe on stripe_device [at backup_server_name ]
            [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]
   [[stripe on stripe_device [at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]]...]
   [with {
       density = density_value,
       blocksize = number_bytes,
       dumpvolume = volume_name,
       file = file_name,
       [dismount | nodismount],
       [nounload | unload],
       listonly [= full],
       headeronly,
       notify = {client | operator_console}
       }]]
```

Keywords and Options

database_name – is the name of the database that will receive the backup copy. It can be either a database created with the **for load** option, or an existing database. Loading dumped data to an existing database overwrites all existing data. The receiving database must be at least as large as the dumped database. The database name can be specified as a literal, a local variable, or a stored procedure parameter.

from *stripe_device* – is the device from which data is being loaded. See "Specifying Dump Devices" for information about what form to use when specifying a dump device. See the Adaptive Server installation and configuration guide for a list of supported dump devices.

at *backup_server_name* – is the name of a remote Backup Server running on the machine to which the dump device is attached. For platforms that use interfaces files, the *backup_server_name* must appear in the interfaces file.

density = *density_value* – is ignored.

blocksize = *number_bytes* – overrides the default block size for a dump device. Do not specify a block size on OpenVMS systems. If you specify a block size on UNIX systems, it should be identical to that used to make the dump.

dumpvolume = *volume_name* – is the volume name field of the ANSI tape label. **load database** checks this label when the tape is opened and generates an error message if the wrong volume is loaded.

stripe on *stripe_device* – is an additional dump device. You can use up to 32 devices, including the device named in the **to** *stripe_device* clause. The Backup Server loads data from all devices concurrently, reducing the time and the number of volume changes required. See "Specifying Dump Devices" for information about how to specify a dump device.

dismount | nodismount – **on platforms that support logical dismount** (such as OpenVMS), determines whether tapes remain mounted. By default, all tapes used for a load are dismounted when the load completes. Use **nodismount** to keep tapes available for additional loads or dumps.

nounload | unload – determines whether tapes rewind after the load completes. By default, tapes do not rewind, allowing you to make additional loads from the same tape volume. Specify **unload** for the last dump file to be loaded from a multi-dump volume. This rewinds and unloads the tape when the load completes.

file = *file_name* – is the name of a particular database dump on the tape volume. If you did not record the dump file names at the time you made the dump, use **listonly** to display information about all dump files.

listonly [**= full**] – displays information about all dump files on a tape volume, but **does not load the database**. **listonly** identifies the database and device, the date and time the dump was made, and the date and time it can be overwritten. **listonly = full** provides additional details about the dump. Both reports are sorted by ANSI tape label.

After listing the files on a volume, the Backup Server sends a volume change request. The operator can either mount another tape volume or terminate the list operation for all dump devices.

Due to current implementation, the **listonly** option overrides the **headeronly** option.

◆ ***WARNING!***

Do not use load database with listonly on 1/4-inch cartridge tape.

headeronly – displays header information for a single dump file, but **does not load the database**. **headeronly** displays information about the first file on the tape unless you use the **file =** *file_name* option to specify another file name. The dump header indicates:

- Type of dump (database or transaction log)
- Database ID
- File name
- Date the dump was made
- Character set
- Sort order
- Page count
- Next object ID

notify = {**client** | **operator_console**} – overrides the default message destination.

- On operating systems (such as OpenVMS) that offer an operator terminal feature, volume change messages are always sent to the operator terminal on the machine on which the Backup Server is running. Use **client** to route other Backup Server messages to the terminal session that initiated the **dump database**.
- On operating systems (such as UNIX) that do not offer an operator terminal feature, messages are sent to the client that

initiated the **dump database**. Use **operator_console** to route messages to the terminal on which the Backup Server is running.

Examples

1. **For UNIX:**
```
load database pubs2
  from "/dev/nrmt0"
```
   For OpenVMS:
```
load database pubs2
  from "MTA0:"
```
   Reloads the database *pubs2* from a tape device.

2. **For UNIX:**
```
load database pubs2
    from "/dev/nrmt4" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt5" at REMOTE_BKP_SERVER
      stripe on "/dev/nrmt0" at REMOTE_BKP_SERVER
```
   For OpenVMS:
```
load database pubs2
    from "MTA0:" at REMOTE_BKP_SERVER
    stripe on "MTA1:" at REMOTE_BKP_SERVER
  stripe on "MTA2:" at REMOTE_BKP_SERVER
```
   Loads the *pubs2* database, using the Backup Server REMOTE_BKP_SERVER. This command names three devices.

Comments

- The **listonly** and **headeronly** options display information about the dump files without loading them.
- Dumps and loads are performed through Backup Server.
- Table 1-17 describes the commands and system procedures used to restore databases from backups:

Table 1-17: Commands used to restore databases from dumps

Use This Command	To Do This
create database for load	Create a database for the purpose of loading a dump.
load database	Restore a database from a dump.
load transaction	Apply recent transactions to a restored database.

Table 1-17: Commands used to restore databases from dumps (continued)

Use This Command	To Do This
online database	Make a database available for public use after a normal load sequence or after upgrading the database to the current version of Adaptive Server.
load { database | transaction } with {headeronly | listonly}	Identify the dump files on a tape.
sp_volchanged	Respond to Backup Server's volume change messages.

load database Restrictions

- You cannot load a dump that was made on a different platform.
- You cannot load a dump that was generated on a pre-release 10.0 server.
- If a database has cross-database referential integrity constraints, the *sysreferences* system table stores the **name**—not the ID number—of the external database. Adaptive Server cannot guarantee referential integrity if you use **load database** to change the database name or to load it onto a different server.
- Each time you add or remove a cross-database constraint or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases can cause database corruption. Before dumping a database in order to load it with a different name or move it to another Adaptive Server, use alter table to drop all external referential integrity constraints.

- **load database** clears the suspect page entries pertaining to the loaded database from *master..sysattributes.*
- **load database** overwrites any existing data in the database.
- The receiving database must be as large as or larger than the database to be loaded. If the receiving database is too small, Adaptive Server displays an error message that gives the required size.

- You cannot load from the null device (on UNIX, */dev/null*; on OpenVMS, any device name beginning with "NL").
- You cannot use the **load database** command in a user-defined transaction.

Locking Out Users During Loads

- While you are loading a database, it cannot be in use. The **load database** command sets the status of the database to "offline." No one can use the database while its status is "offline". The "offline" status prevents users from accessing and changing the database during a load sequence.
- A database loaded by **load database** remains inaccessible until the **online database** command is issued.

Upgrading Database and Transaction Log Dumps

- To restore and upgrade a user database dump from a release 10.0 or later server to the current release of Adaptive Server:
  1. Load the most recent database dump.
  2. Load, **in order,** all transaction log dumps made since the last database dump.

     Adaptive Server checks the timestamp on each dump to make sure that it is being loaded to the correct database and in the correct sequence.
  3. Issue the **online database** command to do the upgrade and make the database available for public use.
  4. Dump the newly upgraded database immediately after upgrade, to create a dump consistent with the current release of Adaptive Server.

Specifying Dump Devices

- You can specify the dump device as a literal, a local variable, or a parameter to a stored procedure.
- You can specify a local device as:
  - A logical device name from the *sysdevices* system table
  - An absolute path name
  - A relative path name

The Backup Server resolves relative path names using Adaptive Server's current working directory.

- When loading across the network, specify the absolute path name of the dump device. The path name must be valid on the machine on which the Backup Server is running. If the name includes characters other than letters, numbers, or the underscore (_), enclose the entire name in quotes.
- Ownership and permissions problems on the dump device may interfere with use of load commands.
- You can run more than one load (or dump) at the same time, as long as each load uses a different physical device.

Backup Servers

- You must have a Backup Server running on the same machine as Adaptive Server. (On OpenVMS systems, the Backup Server can be running in the same cluster as the Adaptive Server, as long as all database devices are visible to both servers.) The Backup Server must be listed in the *master..sysservers* table. This entry is created during installation or upgrade and should not be deleted.
- If your backup devices are located on another machine, so that you load across a network, you must also have a Backup Server installed on the remote machine.

Volume Names

- Dump volumes are labeled according to the ANSI tape labeling standard. The label includes the logical volume number and the position of the device within the stripe set.
- During loads, Backup Server uses the tape label to verify that volumes are mounted in the correct order. This allows you to load from a smaller number of devices than you used at dump time.

➤ ***Note***

When dumping and loading across the network, you must specify the same number of stripe devices for each operation.

Changing Dump Volumes

- If the Backup Server detects a problem with the currently mounted volume, it requests a volume change by sending messages to either the client or its operator console. After mounting another volume, the operator notifies the Backup Server by executing the **sp_volchanged** system procedure on any Adaptive Server that can communicate with the Backup Server.

- On OpenVMS systems, the operating system requests a volume change when the specified drive is offline. After mounting another volume, the operator uses the REPLY command to reply to volume change messages.

Restoring the System Databases

- See Chapter 22, "Restoring the System Databases," in the *System Administration Guide* for step-by-step instructions for restoring the system databases from dumps.

Disk Mirroring

- At the beginning of a load, Adaptive Server passes Backup Server the primary device name of each logical database and log device. If the primary device has been unmirrored, Adaptive Server passes the name of the secondary device instead. If any named device fails before Backup Server completes its data transfer, Adaptive Server aborts the load.
- If you attempt to unmirror any named device while a **load database** is in progress, Adaptive Server displays a message. The user executing the **disk unmirror** command can abort the load or defer the **disk unmirror** until after the load completes.
- Backup Server loads the data onto the primary device, and then **load database** copies it to the secondary device. **load database** takes longer to complete if any database device is mirrored.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Only a System Administrator, Database Owner, or user with the Operator role can execute **load database**.

See Also

Commands	**dbcc, dump database, dump transaction, load transaction, online database**
System procedures	**sp_helpdevice, sp_volchanged, sp_helpdb**

load transaction

Function

Loads a backup copy of the transaction log that was created with the **dump transaction** command.

Syntax

```
load tran[saction] database_name
   from stripe_device [at backup_server_name]
       [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [at backup_server_name]
       [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]
   [[stripe on stripe_device [at backup_server_name]
       [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]]...]
   [with {
       density = density_value,
       blocksize = number_bytes,
       dumpvolume = volume_name,
       file = file_name,
       [dismount | nodismount],
       [nounload | unload],
       listonly [= full],
       headeronly,
       notify = {client | operator_console}
       until_time = datetime}]]
```

Keywords and Options

database_name – is the name of the database that will receive data from a dumped backup copy of the transaction log. The log segment of the receiving database must be at least as large as the log segment of the dumped database. The database name can be specified as a literal, a local variable, or a parameter of a stored procedure.

from *stripe_device* – is the name of the dump device from which you are loading the transaction log. See "Specifying Dump Devices" for information about what form to use when specifying a dump device. See the Adaptive Server installation and configuration guide for a list of supported dump devices.

at *backup_server_name* – is the name of a remote Backup Server running on the machine to which the dump device is attached. For platforms that use interfaces files, the *backup_server_name* must appear in the interfaces file.

density = *density_value* – overrides the default density for a tape device. **This option is ignored.**

blocksize = *number_bytes* – overrides the default block size for a dump device. Do not specify a block size on OpenVMS systems. If you specify a block size on UNIX systems, it should be identical to that used to make the dump.

dumpvolume = *volume_name* – is the volume name field of the ANSI tape label. **load transaction** checks this label when the tape is opened and generates an error message if the wrong volume is loaded.

stripe on *stripe_device* – is an additional dump device. You can use up to 32 devices, including the device named in the **to** *stripe_device* clause. The Backup Server loads data from all devices concurrently, reducing the time and the number of volume changes required. See "Specifying Dump Devices" for information about how to specify a dump device.

dismount | **nodismount** – **on platforms that support logical dismount** (such as OpenVMS), determines whether tapes remain mounted. By default, all tapes used for a load are dismounted when the load completes. Use **nodismount** to keep tapes available for additional loads or dumps.

nounload | **unload** – determines whether tapes rewind after the load completes. By default, tapes do not rewind, allowing you to make additional loads from the same tape volume. Specify **unload** for the last dump file to be loaded from a multidump volume. This rewinds and unloads the tape when the load completes.

file = *file_name* – is the name of a particular database dump on the tape volume. If you did not record the dump file names at the time you made the dump, use **listonly** to display information about all the dump files.

listonly [**= full**] – displays information about all the dump files on a tape volume, but **does not load the transaction log**. **listonly** identifies the database and device, the date and time the dump was made, and the date and time it can be overwritten. **listonly = full** provides additional details about the dump. Both reports are sorted by ANSI tape label.

After listing the files on a volume, the Backup Server sends a volume change request. The operator can either mount another tape volume or terminate the list operation for all dump devices.

Due to current implementation, the **listonly** option overrides the **headeronly** option.

◆ ***WARNING!***

Do not use load transaction with listonly on 1/4-inch cartridge tape.

headeronly – displays header information for a single dump file, but **does not load the database**. **headeronly** displays information about the first file on the tape unless you use the **file =** *file_name* option to specify another file name. The dump header indicates:

- Type of dump (database or transaction log)
- Database ID
- File name
- Date the dump was made
- Character set
- Sort order
- Page count
- Next object ID
- Checkpoint location in the log
- Location of the oldest **begin transaction** record
- Old and new sequence dates

notify = {**client** | **operator_console**} – overrides the default message destination.

- On operating systems (such as OpenVMS) that offer an operator terminal feature, volume change messages are always sent to the operator terminal on the machine on which the Backup Server is running. Use **client** to route other Backup

Server messages to the terminal session that initiated the **dump database**.

- On operating systems (such as UNIX) that do not offer an operator terminal feature, messages are sent to the client that initiated the **dump database**. Use **operator_console** to route messages to the terminal on which the Backup Server is running.

until_time – loads the transaction log up to a specified time in the transaction log. Only transactions committed before the specified time are saved to the database.

Examples

1. For UNIX:

```
load transaction pubs2
  from "/dev/nrmt0"
```

For OpenVMS:

```
load transaction pubs2
  from "MTA0:"
```

Loads the transaction log for the database *pubs2* tape.

2. For UNIX:

```
load transaction pubs2
      from "/dev/nrmt4" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt5" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt0" at REMOTE_BKP_SERVER
```

For OpenVMS:

```
load transaction pubs2
  from "MTA0:" at REMOTE_BKP_SERVER
  stripe on "MTA1:" at REMOTE_BKP_SERVER
  stripe on "MTA2:" at REMOTE_BKP_SERVER
```

Loads the transaction log for the *pubs2* database, using the Backup Server REMOTE_BKP_SERVER.

3.
```
load transaction pubs2
from "/dev/ntmt0"
with until_time = "mar 20, 1997 10:51:43:866am"
```

Loads the transaction log for *pubs2*, up to March 20, 1997, at 10:51:43:866 a.m.

Comments

- The **listonly** and **headeronly** options display information about the dump files without loading them.
- Dumps and loads are performed through Backup Server.

- Table 1-18 describes the commands and system procedures used to restore databases from backups:

Table 1-18: Commands used to restore databases

Use This Command	To Do This
create database for load	Create a database for the purpose of loading a dump.
load database	Restore a database from a dump.
load transaction	Apply recent transactions to a restored database.
online database	Make a database available for public use after a normal load sequence or after upgrading the database to the current version of Adaptive Server.
load { database | transaction } with {headeronly | listonly}	Identify the dump files on a tape.
sp_volchanged	Respond to the Backup Server's volume change messages.

load transaction Restrictions

- You cannot load a dump that was made on a different platform.
- You cannot load a dump that was generated on a pre-release 10.0 server.
- The database and transaction logs must be at the same release level.
- Load transaction logs in chronological order.
- You cannot load from the null device (on UNIX, */dev/null*; on OpenVMS, any device name beginning with "NL").
- You cannot use **load transaction** after an **online database** command that does an upgrade. The following sequence is **incorrect** for upgrading a database: **load database**, **online database**, **load transaction**. The correct sequence for upgrading a database is **load database**, **load transaction**, **online database**.
- You can use **load transaction** after **online database** if there was no upgrade or version change.
- You cannot use the **load transaction** command in a user-defined transaction.

Restoring a Database

- To restore a database:
  - Load the most recent database dump
  - Load, **in order**, all transaction log dumps made since the last database dump
  - Issue the **online database** command to make the database available for public use.
- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases can cause database corruption.

- For more information on backup and recovery of Adaptive Server databases, see the *System Administration Guide*.

Recovering a Database to a Specified Time

- You can use the **until_time** option for most databases that can be loaded or dumped. It does not apply to databases such as *master*, in which the data and logs are on the same device. Also, you cannot use it on any database that has had a truncated log since the last **dump database**, such as *tempdb*.
- The **until_time** option is useful for the following reasons:
  - It enables you to have a database consistent to a particular time. For example, in an environment with a Decision Support System (DSS) database and an Online Transaction Processing (OLTP) database, the System Administrator can roll the DSS database to an earlier specified time to compare data between the earlier version and the current version.
  - If a user inadvertently destroys data, such as dropping an important table, you can use the **until_time** option to back out the errant command by rolling forward the database to a point just before the data was destroyed.
- To effectively use the **until_time** option after data has been destroyed, you must know the exact time the error took place. You can find out by executing a **select getdate()** command

immediately after the error. For a more precise time using milliseconds, use the **convert** function, for example:

```
select convert(char(26), getdate(), 109)
--------------------------
Feb 26 1997 12:45:59:650PM
```

- After you load a transaction log using **until_time**, Adaptive Server restarts the database's log sequence. This means that until you dump the database again, you cannot load subsequent transaction logs after the **load transaction** using **until_time**. You will need to dump the database before you can dump another transaction log.
- Only transactions that committed before the specified time are saved to the database. However, in some cases, transactions committed shortly after the **until_time** specification are applied to the database data. This may occur when several transactions are committing at the same time. The ordering of transactions may not be written to the transaction log in time-ordered sequence. In this case, the transactions that are out of time sequence will be reflected in the data that has been recovered. The time should be less than a second.
- For more information on recovering a database to a specified time, see the *System Administration Guide*.

Locking Users out During Loads

- While you are loading a database, it cannot be in use. The **load transaction** command, unlike **load database**, does not change the offline / online status of the database. **load transaction** leaves the status of the database the way it found it. The **load database** command sets the status of the database to "offline". No one can use the database while it is "offline". The "offline" status prevents users from accessing and changing the database during a load sequence.
- A database loaded by **load database** remains inaccessible until the **online database** command is issued.

Upgrading Database and Transaction Log Dumps

- To restore and upgrade a user database dump from a release 10.0 or later server to the current release of Adaptive Server:

  1. Load the most recent database dump.

2. Load, **in order**, all transaction logs generated after the last database dump.
3. Use **online database** to do the upgrade.
4. Dump the newly upgraded database immediately after upgrade, to create a dump that is consistent with the current release of Adaptive Server.

Specifying Dump Devices

- You can specify the dump device as a literal, a local variable, or a parameter to a stored procedure.
- When loading from a local device, you can specify the dump device as:
  - An absolute path name
  - A relative path name
  - A logical device name from the *sysdevices* system table

  Backup Server resolves relative path names, using Adaptive Server's current working directory.
- When loading across the network, you must specify the absolute path name of the dump device. (You cannot use a relative path name or a logical device name from the *sysdevices* system table.) The path name must be valid on the machine on which the Backup Server is running. If the name includes any characters other than letters, numbers or the underscore (_), you must enclose it in quotes.
- Ownership and permissions problems on the dump device may interfere with use of load commands. The **sp_addumpdevice** procedure adds the device to the system tables, but does not guarantee that you can load from that device or create a file as a dump device.
- You can run more than one load (or dump) at the same time, as long as each one uses a different physical device.

Backup Servers

- You must have a Backup Server running on the same machine as your Adaptive Server. (On OpenVMS systems, Backup Server can be running in the same cluster as Adaptive Server, as long as all database devices are visible to both servers.) The Backup Server must be listed in the *master..sysservers* table. This entry is created during installation or upgrade and should not be deleted.

- If your backup devices are located on another machine so that you load across a network, you must also have a Backup Server installed on the remote machine.

Volume Names

- Dump volumes are labeled according to the ANSI tape-labeling standard. The label includes the logical volume number and the position of the device within the stripe set.
- During loads, Backup Server uses the tape label to verify that volumes are mounted in the correct order. This allows you to load from a smaller number of devices than you used at dump time.

➤ ***Note***

When dumping and loading across a network, you must specify the same number of stripe devices for each operation.

Changing Dump Volumes

- If Backup Server detects a problem with the currently mounted volume, it requests a volume change by sending messages to either the client or its operator console. After mounting another volume, the operator notifies Backup Server by executing **sp_volchanged** on any Adaptive Server that can communicate with Backup Server.
- On OpenVMS systems, the operating system requests a volume change when the specified drive is offline. After mounting another volume, the operator uses the **REPLY** command to reply to volume change messages.

Restoring the System Databases

- See Chapter 22, “Restoring the System Databases,” in the *System Administration Guide* for step-by-step instructions for restoring the system databases from dumps.

Disk Mirroring

- At the beginning of a load, Adaptive Server passes the primary device name of each logical database device and each logical log device to the Backup Server. If the primary device has been unmirrored, Adaptive Server passes the name of the secondary device instead. If any named device fails before the Backup

Server completes its data transfer, Adaptive Server aborts the load.

- If you attempt to unmirror any of the named devices while a **load transaction** is in progress, Adaptive Server displays a message. The user executing the **disk unmirror** command can abort the load or defer the **disk unmirror** until after the load completes.
- The Backup Server loads the data onto the primary device, and then **load transaction** copies it to the secondary device. **load transaction** takes longer to complete if any database device is mirrored.

Standards and Compliance

Standard	**Compliance Level**
SQL92	Transact-SQL extension

Permissions

load transaction permission defaults to the Database Owner and Operators. It is not transferable.

See Also

Commands	**disk unmirror, dump database, dump transaction, load database, online database**
System procedures	**sp_dboption, sp_helpdb, sp_helpdevice, sp_volchanged**

online database

Function

Marks a database available for public use after a normal load sequence and, if needed, upgrades a loaded database to the current release of Adaptive Server.

Syntax

```
online database database_name
```

Parameters

database_name – is the name of the database.

Examples

1. **`online database pubs2`**

   Makes the *pubs2* database available for public use after a load sequence completes.

Comments

- The **online database** command brings a database online for general use after a normal database or transaction log load sequence.
- When a **load database** command is issued, the database's status is set to "offline." The offline status is set in the *sysdatabases* system table and remains set until the **online database** command completes.
- Do **not** issue the **online database** command until all transaction logs are loaded. The command sequence is:
  - **load database**
  - **load transaction** (there may be more than one **load transaction**)
  - **online database**
- If you execute **online database** against a currently online database, no processing occurs and no error messages are generated.
- **sp_helpdb** displays the offline / online status of a database.

Upgrading Databases

- **online database** initiates, if needed, the upgrade of a loaded database and transaction log dumps to make the database compatible with the current release of Adaptive Server. After the upgrade completes, the database is made available for public use. If errors occur during processing, the database remains offline.

- **online database** is required only after a database or transaction log load sequence. It is not required for new installations or upgrades. When Adaptive Server is upgraded to a new release, all databases associated with that server are automatically upgraded.
- **online database** only upgrades release 10.0 or later user databases.
- After you upgrade a database with **online database**, dump the newly upgraded database to create a dump that is consistent with the current release of Adaptive Server. You must dump the upgraded database before you can issue a **dump transaction** command.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Only a System Administrator, Database Owner, or user with the Operator role can execute **online database**.

See Also

Commands	**dump database, dump transaction, load database, load transaction**
System Procedures	**sp_helpdb**

open

Function

Opens a cursor for processing.

Syntax

```
open cursor_name
```

Parameters

cursor_name – is the name of the cursor to open.

Examples

1. `open authors_crsr`

   Opens the cursor named *authors_crsr*.

Comments

- **open** opens a cursor. Cursors allow you to modify or delete rows on an individual basis. You must first open a cursor to use the **fetch**, **update**, and **delete** statements. For more information about cursors, see Chapter 17, "Cursors: Accessing Data Row by Row," in the *Transact-SQL User's Guide*.
- Adaptive Server returns an error message if the cursor is already open or if the cursor has not been created with the **declare cursor** statement.
- Opening the cursor causes Adaptive Server to evaluate the **select** statement that defines the cursor (specified in the **declare cursor** statement) and makes the cursor result set available for processing.
- When the cursor is first opened, it is positioned before the first row of the cursor result set.
- When you set the chained transaction mode, Adaptive Server implicitly begins a transaction with the **open** statement if no transaction is currently active.

Permissions

open permission defaults to all users.

Standards and Compliance

Standard	Compliance Level
SQL92	Entry level compliant

See Also

Commands	**close, declare cursor, fetch**

order by Clause

Function

Returns query results in the specified column(s) in sorted order.

Syntax

```
[Start of select statement]

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

[End of select statement]
```

Keywords and Options

order by – sorts the results by columns.

asc – sorts the results in ascending order. If you do not specify **asc** or **desc**, **asc** is assumed.

desc – sorts the results in descending order.

Examples

1. **select title, type, price**
   from titles
   where price > $19.99
   order by title

```
title
        type         price
------------------------------------------------------------
        ------------ ------------------------
But Is It User Friendly?
         popular_comp                    22.95
Computer Phobic and Non-Phobic Individuals: Behavior Variations
          psychology                      21.59
Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
         trad_cook                       20.95
Secrets of Silicon Valley
         popular_comp                    20.00
```

Selects the titles whose price is greater than $19.99 and lists them with the titles in alphabetical order.

2. **select type, price, advance**
   from titles
   order by type desc
   compute avg(price), avg(advance) by type

   Lists the books from the *titles* table, in descending alphabetical order of the type, and calculates the average price and advance for each type.

3. **select title_id, advance/total_sales**
   from titles
   order by advance/total_sales

```
title_id
-------- ------------------------
MC3026                       NULL
PC9999                       NULL
MC2222                       0.00
TC4203                       0.26
PS3333                       0.49
BU2075                       0.54
MC3021                       0.67
PC1035                       0.80
PS2091                       1.11
PS7777                       1.20
BU1032                       1.22
BU7832                       1.22
BU1111                       1.29
PC8888                       1.95
TC7777                       1.95
PS1372                      18.67
TC3218                      18.67
PS2106                      54.05
```

   Lists the title IDs from the *titles* table, with the advances divided by the total sales, ordered from the lowest calculated amount to the highest.

4. **select title as BookName, type as Type**
   from titles
   order by Type

   Lists book titles and types in order by the type, renaming the columns in the output.

Comments

- **order by** returns query results in the specified column(s) in sorted order. **order by** is part of the **select** command.

- In Transact-SQL, you can use **order by** to sort items that do not appear in the select list. You can sort by a column heading, a column name, an expression, an alias name (if specified in the select list), or a number representing the position of the item in the select list (*select_list_number*).
- If you sort by *select_list_number*, the columns to which the *order by* clause refers must be included in the **select** list, and the **select** list cannot be * (asterisk).
- Use **order by** to display your query results in a meaningful order. Without an **order by** clause, you cannot control the order in which Adaptive Server returns results.

Restrictions

- The maximum number of columns allowed in an **order by** clause is 16.
- The sum of the maximum lengths of all the columns specified by the **order by** clause cannot exceed 2014 bytes.
- **order by** cannot be used on *text* or *image* datatype columns.
- Subqueries and view definitions cannot include an **order by** clause (or a **compute** clause or the keyword **into**). Conversely, you cannot use a subquery in an **order by** list.
- You cannot update the result set of a server- or language- type cursor if it contains an **order by** clause in its **select** statement. See Chapter 17, "Cursors: Accessing Data Row by Row," in the *Transact-SQL User's Guide* for more information about the restrictions applied to updatable cursors.
- If you use **compute by**, you must also use an **order by** clause. The expressions listed after **compute by** must be identical to or a subset of those listed after **order by**, must be in the same left-to-right order, must start with the same expression, and must not skip any expressions. For example, if the **order by** clause is:

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

  the **compute by** clause can be any (or all) of these:

  ```
  compute by a, b, c
  compute by a, b
  compute by a
  ```

  The keyword **compute** can be used without **by** to generate grand totals, grand counts, and so on. In this case, **order by** is optional.

Collating Sequences

- With **order by**, null values precede all others.
- The sort order (collating sequence) on your Adaptive Server determines how your data is sorted. The sort order choices are binary, dictionary, case-insensitive, case-insensitive with preference, and case- and accent-insensitive. Sort orders that are specific to specific national languages may also be provided.

Table 1-19: Effect of sort order choices

Adaptive Server Sort Order	Effects on *order by* Results
Binary order	Sorts all data according to the numeric byte-value of each character in the character set. Binary order sorts all uppercase letters before lowercase letters. Binary sort order is the only option for multibyte character sets.
Dictionary order	Sorts uppercase letters before their lowercase counterparts (case-sensitive). Dictionary order recognizes the various accented forms of a letter and sorts them after the unaccented form.
Dictionary order, case-insensitive	Sorts data in dictionary order but does not recognize case differences. Uppercase letters are equivalent to their lowercase counterparts and are sorted as described in "Sort Rules".
Dictionary order, case-insensitive with preference	Sorts an uppercase letter in the preferred position, before its lowercase version. It does not recognize case difference when performing comparisons (for example, in where clauses).
Dictionary order, case- and accent-insensitive	Sorts data in dictionary order, but does not recognize case differences; treats accented forms of a letter as equivalent to the associated unaccented letter. It intermingles accented and unaccented letters in sorting results.

- The system procedure **sp_helpsort** reports the sort order installed on Adaptive Server.

Sort Rules

- When two rows have equivalent values in Adaptive Server's sort order, the following rules are used to order the rows:
  - The values in the columns named in the **order by** clause are compared.
  - If two rows have equivalent column values, the binary value of the entire rows is compared byte by byte. This comparison is performed on the row in the order in which the columns are stored internally, not the order of the columns as they are named in the query or in the original **create table** clause. (In brief, data is stored with all the fixed-length columns, in order, followed by all the variable length columns, in order.)
  - If rows are equal, row IDs are compared.

  Given this table:

  ```
  create table sortdemo (lname varchar(20),
                         init char(1) not null)
  ```

  and this data:

  ```
  lname      init
  ---------- ----
  Smith      B
  SMITH      C
  smith      A
  ```

  you get these results when you order by *lname*:

  ```
  lname      init
  ---------- ----
  smith      A
  Smith      B
  SMITH      C
  ```

  Since the fixed-length *char* data (the *init* column) is stored first internally, the **order by** sorts these rows based on the binary values "Asmith", "BSmith" and "CSMITH".

  However, if the *init* is of type *varchar*, the *lname* column is stored first, ane then the *init* column. The comparison takes place on the binary values "SMITHC", "SmithB", and "smithA", and the rows are returned in that order.

Descending Scans

- Use of the keyword **desc** in an **order by** clause allows the query optimizer to choose a strategy that eliminates the need for a worktable and a sort step to return results in descending order. This optimization scans the page chain of the index in reverse order, following the previous page pointers on each index page.

In order to use this optimization, the columns in the **order by** clause must match the index order. They can be a subset of the keys, but must be a prefix subset, that is, they must include the first key(s). The descending scan optimization cannot be used if the columns named in the **order by** clause are a superset of the index keys.

If the query involves a join, all tables can be scanned in descending key order, as long as the requirements for a prefix subset of keys are met. Descending scan optimization can also be used for one or more tables in a join, while other tables are scanned in ascending order.

- If other user processes are scanning forward to perform updates or deletes, performing descending scans can cause deadlocks. Deadlocks may also be encountered during page splits and shrinks. You can use the system procedure **sp_sysmon** to track deadlocks on your server, or you can use the configuration parameter **print deadlock information** to send deadlock information to the error log.
- If your applications need to return results in descending order, but the descending scans optimization creates deadlock problems, some possible workarounds are:
  - Use transaction isolation level 0 scans for descending scans. See "Isolation Level 0" in Chapter 5, "Locking in Adaptive Server," of the *Performance and Tuning Guide* for more information on the effect of isolation level 0 reads.
  - Disable descending scan optimization with the configuration parameter **allow backward scans** so that all queries that use **desc** will scan the table in ascending order and sort the result set into descending order. See "allow backward scans" in Chapter 11, "Setting Configuration Parameters," of the *System Administration Guide.*
  - Break problematical descending scans into two steps, selecting the required rows into a temporary table in ascending order in the first step, and selecting from the temporary table in descending order in the second step.
- If a backward scan uses a clustered index that contains overflow pages because duplicate key values are present, the result set returned by the descending scan may not be in exact reverse

order of the result set that is returned with an ascending scan. The specified key values are returned in order, but the order of the rows for the identical keys on the overflow pages may be different. For an explanation of how overflow pages in clustered indexes are stored, see "Overflow Pages" in Chapter 4, "How Indexes Work," of the *Performance and Tuning Guide.*

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	Specifying new column headings in the **order by** clause of a select statement when the **union** operator is used is a Transact-SQL extension.

See Also

Commands	**compute Clause**, **declare**, **group by and having Clauses**, **select**, **where Clause**
System procedures	**sp_configure**, **sp_helpsort**, **sp_lock**, **sp_sysmon**

prepare transaction

Function

Used by DB-Library in a two-phase commit application to see if a server is prepared to commit a transaction.

Syntax

```
prepare tran[saction]
```

Comments

- See the *Open Client DB-Library Reference Manual* for more information.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

See Also

Commands	**begin transaction**, **commit**, **rollback**, **save transaction**

print

Function

Prints a user-defined message on the user's screen.

Syntax

```
print
   {format_string | @local_variable |
   @@global_variable}
       [, arg_list]
```

Keywords and Options

format_string – can be either a variable or a string of characters. The maximum length of *format_string* is 255 bytes.

Format strings can 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*!"—a percent sign (%), followed by an integer from 1 to 20, followed by an exclamation point (!). The integer represents the argument number in the string in the argument list. "%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.

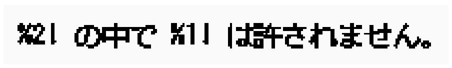
For example, assume the following is an English message:

```
%1! is not allowed in %2!.
```

The German version of this message is:

```
%1! ist in %2! nicht zulassig.
```

The Japanese version of this message is:



```
%2! の中で %1! は許されません。
```

In this example, "%1!" represents the same argument in all three languages, as does "%2!". This example shows the reordering of the arguments that is sometimes necessary in the translated form.

@local_variable – must be of type *char*, *nchar*, *varchar*, or *nvarchar*, and must be declared within the batch or procedure in which it is used.

@@global_variable – must be of type *char* or *varchar,* or be automatically convertible to these types, such as *@@version*. Currently, *@@version* is the only character-type global variable.

arg_list – may be a series of either variables or constants separated by commas. *arg_list* is optional unless a format string containing placeholders of the form "*%nn!*" is provided. In that case, the *arg_list* must have at least as many arguments as the highest numbered placeholder. An argument can be any datatype except *text* or *image*; it is converted to a character datatype before being included in the final message.

Examples

1.
```
if exists (select postalcode from authors
where postalcode = ’94705’)
print "Berkeley author"
```

Prints "Berkeley author" if any authors in the *authors* table live in the 94705 ZIP code.

2.
```
declare @msg char(50)
select @msg = "What’s up, doc?"
print @msg
```

```
What’s up, doc?
```

Delcares a variable, assigns a value to the variable, and prints the value.

3.
```
declare @tabname varchar(30)
select @tabname = "titles"

declare @username varchar(30)
select @username = "ezekiel"

print "The table ’%1!’ is not owned by the user
’%2!’.", @tabname, @username
```

```
The table ’titles’ is not owned
by the user ’ezekiel.’
```

Demonstrates the use of variables and placeholders in messages.

Comments

- The maximum output string length of *format_string* plus all arguments after substitution is 512 bytes.
- If you use placeholders in a format string, keep this in mind: for each placeholder *n* in the string, the placeholders 1 through *n*-1 must also exist in the same string, although they do not have to be in numerical order. For example, you cannot have placeholders 1 and 3 in a format string without having placeholder 2 in the same string. If you omit a number in a format string, an error message is generated when **print** is executed.
- The *arg_lis*t must include an argument for each placeholder in the *format_string*, or the transaction is aborted. It is permissible to have more arguments than placeholders.
- To include a literal percent sign as part of the error message, use two percent signs ("%%") in the *format_string*. If you include a single percent sign ("%") in the *format_string* that is not used as a placeholder, Adaptive Server returns an error message.
- If an argument evaluates to NULL, it is converted into a zero-length character string. If you do not want zero-length strings in the output, use the **isnull** function. For example, if *@arg* is null, the following:

  ```
  declare @arg varchar(30)
  select @arg = isnull(col1, "nothing") from
  table_a where ...
  ```

  ```
  print "I think we have %1! here", @arg
  ```

  prints:

  ```
  I think we have nothing here.
  ```

- User-defined messages can be added to the system table *sysusermessages* for use by any application. Use **sp_addmessage** to add messages to *sysusermessages*; use **sp_getmessage** to retrieve messages for use by **print** and **raiserror**.
- Use **raiserror** instead of **print** if you want to print a user-defined error message and have the error number stored in *@@error*.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

print permission defaults to all users. No permission is required to use it.

See Also

Commands	**declare, raiserror**
System procedures	**sp_addmessage, sp_getmessage**

raiserror

Function

Prints a user-defined error message on the user's screen and sets a system flag to record that an error condition has occurred.

Syntax

```
raiserror error_number
   [{format_string | @local_variable}] [, arg_list]
   [with errordata restricted_select_list]
```

Keywords and Options

error_number – is a local variable or an integer with a value 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.

If *error_number* is 20,000 or greater and *format_string* is missing or empty, **raiserror** retrieves the message text from the *sysusermessages* table in the database from which the query or stored procedure originates. Adaptive Server attempts to retrieve messages from either *sysmessages* or *sysusermessages* in the language defined by the current setting of *@@langid*.

format_string – is a string of characters with a maximum length of 255 bytes. Optionally, you can declare *format_string* in a local variable and use that variable with **raiserror** (see *@local_variable*).

raiserror recognizes placeholders in the character string that is to be printed out. Format strings can 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*!"—a percent sign (%), followed by an integer from 1 to 20, followed by an exclamation point (!). The integer represents the argument number in the string in the argument list. "%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.

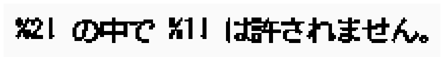
For example, assume the following is an English message:

```
%1! is not allowed in %2!.
```

The German version of this message is:

```
%1! ist in %2! nicht zulassig.
```

The Japanese version of this message is:



```
%2! の中で %1! は許されません。
```

In this example, "%1!" represents the same argument in all three languages, as does "%2!". This example shows the reordering of the arguments that is sometimes necessary in the translated form.

@local_variable – is a local variable containing the *format_string* value. It must be of type *char* or *varchar* and must be declared within the batch or procedure in which it is used.

arg_list – is a series of variables or constants separated by commas. *arg_list* is optional unless a format string containing placeholders of the form "*%nn*!" is provided. An argument can be any datatype except *text* or *image*; it is converted to the *char* datatype before being included in the final string.

If an argument evaluates to NULL, Adaptive Server converts it to a zero-length *char* string.

with errordata – supplies extended error data for Client-Library™ programs.

restricted_select_list – consists of one or more of the following items:

- "*", representing all columns in **create table** order.
- A list of column names in the order in which you want to see them. When selecting an existing IDENTITY column, you can substitute the **syb_identity** keyword, qualified by the table name, where necessary, for the actual column name.
- A specification to add a new IDENTITY column to the result table:

  column_name **= identity(*****precision*****)**

- A replacement for the default column heading (the column name), in the form:

  column_heading = column_name

  or:

  column_name column_heading

  or:

  column_name* as *column_heading

  The column heading may be enclosed in quotation marks for any of these forms. The heading must be enclosed in quotation marks if it is not a valid identifier (that is, if it is a reserved word, if it begins with a special character, or if it contains spaces or punctuation marks).

- An expression (a column name, constant, function, or any combination of column names, constants, and functions connected by arithmetic or bitwise operators, or a subquery).
- A built-in function or an aggregate
- Any combination of the items listed above

The *restricted_select_list* can also perform variable assignment, in the form:

```
@variable = expression
    [, @variable = expression ...]
```

Restrictions to *restricted_select_list* are:

- You cannot combine variable assignment with any of the other *restricted_select_list* options.
- You cannot use **from**, **where**, or other **select** clauses in *restricted_select_list*.
- You cannot use “*” to represent all columns in *restricted_select_list*.

For more information, see Appendix A, “Expressions, Identifiers, and Wildcard Characters,” and Chapter 2, “Transact-SQL Functions,” or Chapter 5, “Subqueries: Using Queries Within Other Queries,” in the *Transact-SQL User’s Guide*

Examples

1.
```
create procedure showtable_sp @tabname varchar(18)
as
if not exists (select name from sysobjects
    where name = @tabname)
    begin
      raiserror 99999 "Table %1! not found.",
        @tabname
    end
else
    begin
      select sysobjects.name, type, crdate, indid
      from sysindexes, sysobjects
      where sysobjects.name = @tabname
      and sysobjects.id = sysindexes.id
    end
```

   This stored procedure example returns an error if it does not find the table supplied with the *@tabname* parameter.

2.
```
sp_addmessage 25001,
"There is already a remote user named '%1!'
for remote server '%2!'."

raiserror 25001, jane, myserver
```

   This example adds a message to *sysusermessages* and then tests the message with **raiserror**, providing the substitution arguments.

3.
```
raiserror 20100 "Login must be at least 5
    characters long" with errordata "column" =
       "login", "server" = @@servername
```

   This example uses the **with errordata** option to return the extended error data *column* and *server* to a client application, to indicate which column was involved and which server was used.

Comments

- User-defined messages can be generated ad hoc, as in examples 1 and 3 above, or they can be added to the system table *sysusermessages* for use by any application, as shown in example 2. Use **sp_addmessage** to add messages to *sysusermessages*; use **sp_getmessage** to retrieve messages for use by **print** and **raiserror**.
- Error numbers for user-defined error messages must be greater than 20,000. The maximum value is 2,147,483,647 (2^{31} -1).
- The severity level of all user-defined error messages is 16. This level indicates that the user has made a a non-fatal error.

- The maximum output string length of *format_string* plus all arguments after substitution is 512 bytes.
- If you use placeholders in a format string, keep this in mind: for each placeholder *n* in the string, the placeholders *1* through *n-1* must exist in the same string, although they do not have to be in numerical order. For example, you cannot have placeholders 1 and 3 in a format string without having placeholder 2 in the same string. If you omit a number in a format string, an error message is generated when **raiserror** is executed.
- If there are too few arguments relative to the number of placeholders in *format_string*, an error message displays and the transaction is aborted. It is permissible to have more arguments than placeholders in *format_string*.
- To include a literal percent sign as part of the error message, use two percent signs (“%%”) in the *format_string*. If you include a single percent sign (“%”) in the *format_string* that is not used as a placeholder, Adaptive Server returns an error message.
- If an argument evaluates to NULL, it is converted into a zero-length *char* string. If you do not want zero-length strings in the output, use the **isnull** function.
- When **raiserror** is executed, the error number is placed in the global variable *@@error*, which stores the error number that was most recently generated by the system.
- Use **raiserror** instead of **print** if you want an error number stored in *@@error*.
- To include an *arg_list* with **raiserror**, put a comma after *error_number* or *format_string* before the first argument. To include extended error data, separate the first *extended_value* from *error_number*, *format_string*, or *arg_list* using a space (not a comma).

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

raiserror permission defaults to all users. No permission is required to use it.

See Also

Commands	**declare, print**
System procedures	**sp_addmessage, sp_getmessage**

readtext

Function

Reads *text* and *image* values, starting from a specified offset and reading a specified number of bytes or characters.

Syntax

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

Keywords and Options

table_name.column_name – is the name of the *text* or *image* column You must include the table name. Specify the database name if the table is in another database, and specify the owner's name if more than one table of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

text_pointer – is a *varbinary*(16) value that stores the pointer to the *text* or *image* data. Use the **textptr** function to determine this value (see example 1). *text* and *image* data is not stored in the same set of linked pages as other table columns. It is stored in a separate set of linked pages. A pointer to the actual location is stored with the data; **textptr** returns this pointer.

offset – specifies the number of bytes or characters to skip before starting to read *text* or *image* data.

size – specifies the number of bytes or characters of data to read.

holdlock – 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.

using – specifies whether **readtext** interprets the *offset* and *size* parameters as a number of bytes (**bytes**) or as a number of **textptr** characters (**chars** or **characters** are synonymous). This option has no effect when used with a single-byte character set or with *image* values (**readtext** reads *image* values byte by byte). If the **using** option is not given, **readtext** interprets the *size* and *offset* arguments as bytes.

at isolation – specifies the isolation level (0, 1, or 3) of the query. If you omit this clause, the query uses the isolation level of the session in which it executes (isolation level 1 by default). If you specify **holdlock** in a query that also specifies **at isolation read uncommitted**, Adaptive Server issues a warning and ignores the **at isolation** clause. For the other isolation levels, **holdlock** takes precedence over the **at isolation** clause.

read uncommitted – specifies isolation level 0 for the query. You can specify 0 instead of **read uncommitted** with the **at isolation** clause.

read committed – specifies isolation level 1 for the query. You can specify "1" instead of **read committed** with the **at isolation** clause.

serializable – specifies isolation level 3 for the query. You can specify "3" instead of **serializable** with the **at isolation** clause.

Examples

1.
```
create table texttest
(title_id varchar(6), blurb text null,
    pub_id char(4))

insert texttest values ("BU1032",
"The Busy Executive’s Database Guide is an
overview of available database systems with
emphasis on common business applications.
Illustrated.", "1389")

declare @val varbinary(16)
select @val = textptr(blurb) from texttest
 where title_id = "BU1032"
readtext texttest.blurb @val 1 5 using chars
```

After creating the table *texttest* and entering values into it, this example selects the second through the sixth character of the *blurb* column.

Comments

- The **textptr** function returns a 16-byte binary string (text pointer) to the *text* or *image* column in the specified row or to the *text* or *image* column in the last row returned by the query, if more than one row is returned. It is best to declare a local variable to hold the text pointer and then use the variable with **readtext**.
- The value in the global variable *@@textsize*, which is the limit on the number of bytes of data to be returned, supersedes the size specified for **readtext** if it is less than that size. Use **set textsize** to change the value of *@@textsize*.

- When using bytes as the offset and size, Adaptive Server may find partial characters at the beginning or end of the *text* data to be returned. If it does, and character set conversion is on, the server replaces each partial character with a question mark (?) before returning the text to the client.
- 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, the 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*. The **using characters** option is useful only when Adaptive Server is using a multibyte character set: this option ensures that **readtext** will not return partial characters.
- You cannot use **readtext** on *text* and *image* columns in views.
- If you attempt to use **readtext** on *text* values after changing to a multibyte character set, and you have not run **dbcc fix_text**, the command fails, and an error message instructs you to run **dbcc fix_text** on the table.

Standards and Compliance

Standard	Compliance level
SQL92	Transact-SQL extension

Permissions

readtext requires **select** permission on the table. **readtext** permission is transferred when **select** permission is transferred.

See Also

Commands	set, writetext
Datatypes	text and image Datatypes

reconfigure

Function

The **reconfigure** command currently has no effect; it is included to allow existing scripts to run without modification. In previous releases, **reconfigure** was required after the system procedure **sp_configure** to implement new configuration parameter settings.

Syntax

```
reconfigure
```

Comments

➤ ***Note***

If you have scripts that include **reconfigure**, change them at your earliest convenience. Although **reconfigure** is included in this release, it may not be supported in subsequent releases.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

reconfigure permission defaults to System Administrators and is not transferable.

See Also

System procedures	**sp_configure**

return

Function

Exits from a batch or procedure unconditionally and provides an optional return status. Statements following **return** are not executed.

Syntax

```
return [integer_expression]
```

Keywords and Options

integer_expression – is the integer value returned by the procedure. Stored procedures can return an integer value to a calling procedure or an application program.

Examples

1.
```
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, the **return** command 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 created by that user in the current database are retrieved from the appropriate system tables.

2.
```
print "Begin update batch"
update titles
    set price = price + $3
    where title_id = 'BU2075'
update titles
    set price = price + $3
    where title_id = 'BU1111'
if (select avg(price) from titles
        where title_id like 'BU%') > $15
begin
      print "Batch stopped; average price over $15"
    return
end
update titles
    set price = price + $2
      where title_id = 'BU1032'
```

   If the updates cause the average price of business titles to exceed $15, the **return** command terminates the batch before any more updates are performed on *titles*.

3.
```
create proc checkcontract @param varchar(11)
as
declare @status int
if (select contract from titles where title_id =
@param) = 1
   return 1
else
   return 2
```

   This procedure creates two user-defined status codes: a value of 1 is returned if the *contract* column contains a 1; a value of 2 is returned for any other condition (for example, a value of 0 on *contract* or a *title_id* that did not match a row).

Comments

- The return status value can be used in subsequent statements in the batch or procedure that executed the current procedure, but must be given in the form:

  execute @*retval* = *procedure_name*

  See **execute** for more information.

- Adaptive Server reserves 0 to indicate a successful return, and negative values in the range -1 to -99 to indicate different reasons for failure. If no user-defined return value is provided, the Adaptive Server value is used. User-defined return status values must not conflict with those reserved by Adaptive Server. Numbers 0 and -1 to -14 are currently in use:

Table 1-20: Adaptive Server error return values

Value	Meaning
0	Procedure executed without error
-1	Missing object
-2	Datatype error
-3	Process was chosen as deadlock victim
-4	Permission error
-5	Syntax error
-6	Miscellaneous user error
-7	Resource error, such as out of space
-8	Non-fatal internal problem
-9	System limit was reached
-10	Fatal internal inconsistency
-11	Fatal internal inconsistency
-12	Table or index is corrupt
-13	Database is corrupt
-14	Hardware error

Values -15 to -99 are reserved for future Adaptive Server use.

- If more than one error occurs during execution, the status with the highest absolute value is returned. User-defined return values always take precedence over Adaptive Server-supplied return values.
- The **return** command can be used at any point where you want to exit from a batch or procedure. Return is immediate and complete: statements after **return** are not executed.
- A stored procedure cannot return a NULL return status. If a procedure attempts to return a null value, for example, using **return** *@status* where *@status* is NULL, a warning message is generated, and a value in the range of 0 to -14 is returned.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

return permission defaults to all users. No permission is required to use it.

See Also

Commands	**begin...end, execute, if...else, while**

revoke

Function

Revokes permissions or roles from users or roles.

Syntax

To revoke permission to access database objects:

```
revoke [grant option for]
   {all [privileges] | permission_list}
   on { table_name [(column_list)]
       | view_name [(column_list)]
       | stored_procedure_name}
   from {public | name_list | role_name}
   [cascade]
```

To revoke permission to create database objects, execute **set proxy**, or execute **set session authorization**:

```
revoke {all [privileges] | command_list}
   from {public | name_list | role_name}
```

To revoke a role from a user or another role:

```
revoke role {role_name [, role_name ...]} from
   {grantee [, grantee ...]}
```

Keywords and Options

all – when used to revoke permission to access database objects (the first syntax format), **all** revokes all permissions applicable to the specified object. All object owners can use **revoke all** with an object name to revoke permissions on their own objects.

Only the System Administrator or the Database Owner can revoke permission to revoke **create** command permissions (the second syntax format). When used by the System Administrator, **revoke all** revokes all **create** permissions (**create database**, **create default**, **create procedure**, **create rule**, **create table**, and **create view**). When the Database Owner uses **revoke all**, Adaptive Server revokes all **create** permissions except **create database**, and prints an informational message.

all does not apply to **set proxy** or **set session authorization**.

permission_list – is a list of permissions to revoke. If more than one permission is listed, separate them with commas. The following table illustrates the access permissions that can be granted and revoked on each type of object:

Object	***permission_list* Can Include:**
Table	**select, insert, delete, update, references**
View	**select, insert, delete, update**
Column	**select, update, references** Column names can be specified in either *permission_list* or *column_list* (see example 2).
Stored procedure	**execute**

Permissions can be revoked only by the user who granted them.

command_list – is a list of commands. If more than one command is listed, separate them with commas. The command list can include **create database**, **create default**, **create procedure**, **create rule**, **create table**, **create view**, **set proxy**, or **set session authorization**. **create database** permission can be revoked only by a System Administrator and only from within the *master* database.

set proxy and **set session authorization** are identical; the only difference is that **set session authorization** follows the SQL standard, and **set proxy** is a Transact-SQL extension. Revoking permission to execute **set proxy** or **set session authorization** revokes permission to become another user in the server. Permissions for **set proxy** or **set session authorization** can be revoked only by a System Security Officer, and only from within the *master* database.

table_name – is the name of the table on which you are revoking permissions. The table must be in your current database. Only one object can be listed for each **revoke** statement.

column_list – is a list of columns, separated by commas, to which the privileges apply. If columns are specified, only **select** and **update** permissions can be revoked.

view_name – is the name of the view on which you are revoking permissions. The view must be in your current database. Only one object can be listed for each **revoke** statement.

stored _procedure_name – is the name of the stored procedure on which you are revoking permissions. The stored procedure must be in your current database. Only one object can be listed for each **revoke** statement.

public – is all users. For object access permissions, **public** excludes the object owner. For object creation permissions or **set proxy** authorizations, **public** excludes the Database Owner. You cannot **grant** permissions **with grant option** to "public" or to other groups or roles.

name_list – is a list of user and / or group names, separated by commas.

role – is the name of a system or user-defined role. Use **revoke role** to revoke granted roles from roles or users.

role_name – is the name of a system or user-defined role. This allows you to revoke permissions from all users who have been granted a specific role. The role name can be either a system role or a user-defined role created by a System Security Officer with **create role**. Either type of role can be granted to a user with the **grant role** command. In addition, the system procedure **sp_role** can be used to grant system roles.

grantee – is the name of a system role, user-defined role, or a user, from whom you are revoking a role.

grant option for – revokes **with grant option** permissions, so that the user(s) specified in *name_list* can no longer grant the specified permissions to other users. If those users have granted permissions to other users, you must use the **cascade** option to revoke permissions from those users. The user specified in *name_list* retains permission to access the object, but can no longer grant access to other users. **grant option for** applies only to object access permissions, not to object creation permissions.

cascade – revokes the specified object access permissions from all users to whom the revokee granted permissions. Applies only to object access permissions, not to object creation permissions. (When you use **revoke** without **grant option for**, permissions granted to other users by the revokee are also revoked: the cascade occurs automatically.)

Examples

1.
```
revoke insert, delete
on titles
from mary, sales
```

   Revokes **insert** and **delete** permissions on the *titles* table from Mary and the "sales" group.

2.
```
revoke update
on titles (price, advance)
from public
```

   or:

```
revoke update (price, advance)
on titles
from public
```

   Two ways to revoke **update** permission on the *price* and *advance* columns of the *titles* table from "public".

3.
```
revoke create database, create table
from mary, john
```

   Revokes permission from Mary and John to use the **create database** and **create table** commands. Because **create database** permission is being revoked, this command must be executed by a System Administrator from within the *master* database. Mary and John's **create table** permission will be revoked only within the *master* database.

4.
```
revoke set proxy from harry, billy
```

   Revokes permission from Harry and Billy to execute either **set proxy** or **set session authorization** to impersonate another user in the server.

5.
```
revoke set session authorization from sso_role
```

   Revokes permission from users with **sso_role** to execute either **set proxy** or **set session authorization.**

6.
```
revoke set proxy from vip_role
```

   Revokes permission from users with **vip_role** to impersonate another user in the server. **vip_role** must be a role defined by a System Security Officer with the **create role** command.

7.
```
revoke all
from mary
```

   Revokes all object creation permissions from Mary in the current database.

8.
```
revoke all
on titles
from mary
```

   Revokes all object access permissions on the *titles* table from Mary.

9.
```
revoke references
on titles (price, advance)
from tom
```

   or:

```
revoke references (price, advance)
on titles
from tom
```

   Two ways to revoke Tom's permission to create a referential integrity constraint on another table that refers to the *price* and *advance* columns in the *titles* table.

10.
```
revoke execute on new_sproc
from oper_role
```

   Revokes permission to execute the stored procedure *new_sproc* from all users who have been granted the Operator role.

11.
```
revoke grant option for
insert, update, delete
on authors
from john
cascade
```

   Revokes John's permission to grant **insert, update**, and **delete** permissions on the *authors* table to other users. Also revokes from other users any such permissions that John has granted.

12.
```
revoke role doctor_role from specialist_role
```

   Revokes doctor_role from specialist_role.

13.
```
revoke role doctor_role, surgeon_role from
specialist_role, intern_role, mary, tom
```

   Revokes "doctor_role" and "surgeon_role" from "specialist_role" and "intern_role", and from users Mary and Tom.

Comments

- See the **grant** command for more information about permissions.
- You can revoke permissions only on objects in your current database.

- You can only revoke permissions that were granted by you.
- You cannot revoke a role from a user while the user is logged in.
- **grant** and **revoke** commands are order sensitive. When there is a conflict, the command issued most recently takes effect.
- The word **to** can be substituted for the word **from** in the **revoke** syntax.
- If you do not specify **grant option for** in a **revoke** statement, **with grant option** permissions are revoked from the user along with the specified object access permissions. In addition, if the user has granted the specified permissions to any other users, all of those permissions are revoked. In other words, the **revoke** cascades.
- **revoke grant option** revokes the user's ability to grant the specified permission to other users, but does not revoke the permission itself from that user. If the user has granted that permission to others, you must use the **cascade** option; otherwise, you will receive an error message and the **revoke** will fail.

  For example, say you revoke the **with grant option** permissions from the user Bob on *titles*, with this statement:

  ```
  revoke grant option for all
  for select
  on titles
  from bob
  cascade
  ```

  - If Bob has not granted this permission to other users, this command revokes his ability to do so, but he retains **select** permission on the *titles* table.
  - If Bob has granted this permission to other users, you must use the **cascade** option. If you do not, you will receive an error message and the **revoke** will fail. **cascade** revokes this **select** permission from all users to whom Bob has granted it, as well as their ability to grant it to others.
- A **grant** statement adds one row to the *sysprotects* system table for each user, group, or role that receives the permission. If you subsequently **revoke** the permission from the user or group, Adaptive Server removes the row from *sysprotects*. If you revoke the permission from only selected group members, but not from the entire group to which it was granted, Adaptive Server retains the original row and adds a new row for the revoke.
- Permission to issue the **create trigger** command is granted to users by default. When you revoke permission for a user to create

triggers, a revoke row is added in the *sysprotects* table for that user. To grant permission to issue **create trigger**, you must issue two grant commands. The first command removes the revoke row from *sysprotects*; the second inserts a grant row. If you revoke permission to create triggers, the user cannot create triggers even on tables that the user owns. Revoking permission to create triggers from a user affects only the database where the revoke command was issued.

Revoking *set proxy* and *set session authorization*

- To revoke **set proxy** or **set session authorization** permission, or to revoke roles, you must be a System Security Officer, and you must be in the *master* database.
- **set proxy** and **set session authorization** are identical, with one exception: **set session authorization** follows the SQL standard. If you are concerned about using only SQL standard commands and syntax, use **set session authorization.**
- **revoke all** does **not** include **set proxy** or **set session authorization** permissions.

Revoking from Roles, Users and Groups

- Permissions granted to roles override permissions granted to individual users or groups. Therefore, if you revoke a permission from a user who has been granted a role, and the role has that same permission, the user will retain it. For example, say John has been granted the System Security Officer role, and **sso_role** has been granted permission on the *sales* table. If John's individual permission on *sales* is revoked, he can still access *sales* because his role permissions override his individual permissions.
- Revoking a specific permission from "public" or from a group also revokes it from users who were individually granted the permission.
- Database user groups allow you to **grant** or **revoke** permissions to more than one user at a time. A user is always a member of the default group, "public" and can be a member of only one other group. Adaptive Server's installation script assigns a set of permissions to "public."

  Create groups with the system procedure **sp_addgroup** and remove groups with **sp_dropgroup**. Add new users to a group with **sp_adduser.** Change a user's group membership with

sp_changegroup. To display the members of a group, use **sp_helpgroup**.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Database Object Access Permissions

revoke permission for database objects defaults to object owners. An object owner can revoke permission from other users on his or her own database objects.

Command Execution Permissions

Only a System Administrator can revoke **create database** permission, and only from the *master* database. Only a System Security Officer can revoke **create trigger** permission.

Proxy and Session Authorization Permissions

Only a System Security Officer can revoke **set proxy** or **set session authorization**, and only from the *master* database.

Role Permissions

You can revoke roles only from the *master* database. Only a System Security Officer can revoke **sso_role**, **oper_role** or a user-defined role from a user or a role. Only System Administrators can revoke **sa_role** from a user or a role. Only a user who has both **sa_role** and **sso_role** can revoke a role which includes **sa_role**.

See Also

Commands	grant, setuser, set
Functions	**proc_role**
System procedures	**sp_activeroles**, **sp_adduser**, **sp_changedbowner**, **sp_changegroup**, **sp_displaylogin**, **sp_displayroles**, **sp_dropgroup**, **sp_dropuser**, **sp_helpgroup**, **sp_helprotect**, **sp_helpuser**, **sp_modifylogin**, **sp_role**

rollback

Function

Rolls back a user-defined transaction to the named savepoint in the transaction or to the beginning of the transaction.

Syntax

```
rollback {tran[saction] | work}
   [transaction_name | savepoint_name]
```

Keywords and Options

transaction | **tran** | **work** – is optional.

transaction_name – is the name assigned to the outermost transaction. It must conform to the rules for identifiers.

savepoint_name – is the name assigned to the savepoint in the **save transaction** statement. The name must conform to the rules for identifiers.

Examples

1.
   ```
   begin transaction
   delete from publishers where pub_id = "9906"
   rollback transaction
   ```

   Rolls back the transaction.

Comments

- **rollback transaction** without a *transaction_name* or *savepoint_name* rolls back a user-defined transaction to the beginning of the outermost transaction.
- **rollback transaction** *transaction_name* rolls back a user-defined transaction to the beginning of the named transaction. Though you can nest transactions, you can roll back only the outermost transaction.
- **rollback transaction** *savepoint_name* rolls a user-defined transaction back to the matching **save transaction** *savepoint_name*. Savepoints apply only to the outermost transaction.

Restrictions

- If no transaction is currently active, the **commit** or **rollback** statement has no effect.
- The **rollback** command must appear within a transaction. You cannot roll back a transaction after **commit** has been entered.

Rolling Back an Entire Transaction

- **rollback** without a savepoint name cancels an entire transaction. All the transaction's statements or procedures are undone.
- If no *savepoint_name* or *transaction_name* is given with the **rollback** command, the transaction is rolled back to the first **begin transaction** in the batch. This also includes transactions that were started with an implicit **begin transaction** using the chained transaction mode.

Rolling Back to a Savepoint

- To cancel part of a transaction, use **rollback** with a *savepoint_name*. A savepoint is a marker set within a transaction by the user with the command **save transaction**. All statements or procedures between the savepoint and the **rollback** are undone.

  After a transaction is rolled back to a savepoint, it can proceed to completion (executing any SQL statements after that **rollback**) using **commit**, or it can be canceled altogether using **rollback** without a savepoint. There is no limit on the number of savepoints within a transaction.

Rollbacks Within Triggers and Stored Procedures

- In triggers or stored procedures, **rollback** statements without transaction or savepoint names roll back all statements to the first explicit or implicit **begin transaction** in the batch that called the procedure or fired the trigger.
- When a trigger contains a **rollback** command without a savepoint name, the rollback aborts the entire batch. Any statements in the batch following the rollback are not executed.
- A remote procedure call (RPC) is executed independently from any transaction in which it is included. In a standard transaction (that is, not using Open Client™ DB-Library two-phase commit), commands executed via an RPC by a remote server are not rolled back with **rollback** and do not depend on **commit** to be executed.

- See Chapter 18, "Transactions: Maintaining Data Consistency and Recovery," in the *Transact-SQL User's Guide* for complete information on using transaction management statements and on the effects of **rollback** on stored procedures, triggers, and batches.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The **rollback transaction** and **rollback tran** forms of the statement and the use of a transaction name are Transact-SQL extensions.

Permissions

rollback permission defaults to "public." No permission is required to use it.

See Also

Commands	**begin transaction, commit, create trigger, save transaction**

rollback trigger

Function

Rolls back the work done in a trigger, including the data modification that caused the trigger to fire, and issues an optional **raiserror** statement.

Syntax

```
rollback trigger
   [with raiserror_statement]
```

Keywords and Options

with *raiserror_statement* – specifies a **raiserror** statement, which prints a user-defined error message and sets a system flag to record that an error condition has occurred. This provides the ability to raise an error to the client when the rollback trigger is executed so that the transaction state in the error reflects the rollback. For information about the syntax and rules defining *raiserror_statement*, see the **raiserror** command.

Examples

```
1. rollback trigger with raiserror 25002
   "title_id does not exist in titles table."
```

Rolls back a trigger and issues the user-defined error message 25002.

Comments

- When **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 **rollback trigger** is nested within other triggers, Adaptive Server rolls back all work done in these triggers up to and including the update that caused the first trigger to fire.
- Adaptive Server ignores a **rollback trigger** statement that is executed outside a trigger and does not issue a **raiserror** associated with the statement. However, a **rollback trigger** statement 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.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

rollback trigger permission defaults to “public.” No permission is required to use it.

See Also

Commands	create trigger, raiserror, rollback

save transaction

Function

Sets a savepoint within a transaction.

Syntax

```
save transaction savepoint_name
```

Keywords and Options

savepoint_name – is the name assigned to the savepoint. It must conform to the rules for identifiers.

Examples

1.
```
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

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

rollback transaction percentchanged

commit transaction
```

After updating the *royaltyper* entries for the two authors, insert the savepoint *percentchanged*, then determine how a 10 percent increase in the book's price would affect the authors' royalty earnings. The transaction is rolled back to the savepoint with the **rollback transaction** command.

Comments

- See Chapter 18, "Transactions: Maintaining Data Consistency and Recovery," in the *Transact-SQL User's Guide* for complete information on using transaction statements.
- A savepoint is a user-defined marker within a transaction that allows portions of a transaction to be rolled back. The command **rollback** *savepoint_name* rolls back to the indicated savepoint; all statements or procedures between the savepoint and the **rollback** are undone.

  Statements preceding the savepoint are not undone—but neither are they committed. After rolling back to the savepoint, the transaction continues to execute statements. A **rollback** without a savepoint cancels the entire transaction. A **commit** allows it to proceed to completion.
- If you nest transactions, **save transaction** creates a savepoint only in the outermost transaction.
- There is no limit on the number of savepoints within a transaction.
- If no *savepoint_name* or t*ransaction_name* is given with the **rollback** command, all statements back to the first **begin transaction** in a batch are rolled back, and the entire transaction is canceled.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

save transaction permission defaults to “public.” No permission is required to use it.

See Also

Commands	**begin transaction, commit, rollback**

select

Function

Retrieves rows from database objects.

Syntax

```
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]
```

Keywords and Options

all – includes all rows in the results. **all** is the default.

distinct – includes only unique rows in the results. **distinct** must be the first word in the select list. **distinct** is ignored in browse mode.

Null values are considered equal for the purposes of the keyword **distinct**: only one NULL is selected, no matter how many are encountered.

Even when configured for case-insensitive sort order, **distinct** reports "smith" and "Smith" as two distinct rows.

select_list – consists of one or more of the following items:

- "*", representing all columns in **create table** order.
- A list of column names in the order in which you want to see them. When selecting an existing IDENTITY column, you can substitute the **syb_identity** keyword, qualified by the table name, where necessary, for the actual column name.
- A specification to add a new IDENTITY column to the result table:

  ```
  column_name = identity(precision)
  ```

- A replacement for the default column heading (the column name), in the form:

  ```
  column_heading = column_name
  ```

  or:

  ```
  column_name column_heading
  ```

  or:

  ```
  column_name as column_heading
  ```

  The column heading can be enclosed in quotation marks for any of these forms. The heading must be enclosed in quotation marks if it is not a valid identifier (that is, if it is a reserved word, if it begins with a special character, or if it contains spaces or punctuation marks).
- An expression (a column name, constant, function, or any combination of column names, constants, and functions connected by arithmetic or bitwise operators, or a subquery)
- A built-in function or an aggregate
- Any combination of the items listed above

The *select_list* can also assign values to variables, in the form:

```
    @variable = expression
        [, @variable = expression ...]
```

You cannot combine variable assignment with any other *select_list* option.

into – creates a new table based on the columns specified in the select list and the rows chosen in the **where** clause. See “Using select into” in this section.

from – indicates which tables and views to use in the **select** statement. It is required except when the select list contains no column names (that is, it contains constants and arithmetic expressions only):

```
select 5 x, 2 y, "the product is", 5*2 Result
x           y                          Result
----------- ----------- -------------- -----------
          5           2 the product is          10
```

At most, a query can reference 16 tables and 12 worktables (such as those created by aggregate functions). The 16-table limit includes:

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

view_name, *table_name* – lists tables and views used in the **select** statement. Specify the database name if the table or view is in another database, and specify the owner’s name if more than one table or view of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

If there is more than one table or view in the list, separate their names by commas. The order of the tables and views following the keyword **from** does not affect the results.

You can query tables in different databases in the same statement.

Table names and view names can be given correlation names (aliases), either for clarity or to distinguish the different roles that tables or views play in self-joins or subqueries. To assign a correlation name, give the table or view name, then a space, then the correlation name, like this:

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

All other references to that table or view (for example in a **where** clause) must use the correlation name. Correlation names cannot begin with a numeral.

index *index_name* – specifies the index to use to access *table_name*. You cannot use this option when you select from a view, but you can use it as part of a **select** clause in a **create view** statement.

parallel – specifies a parallel partition or index scan, if Adaptive Server is configured to allow parallel processing.

degree_of_parallelism – specifies the number of worker processes that will scan the table or index in parallel. If set to 1, the query executes serially.

prefetch *size* – specifies the I/O size, in kilobytes, for tables bound to caches with large I/Os configured. Valid values for size are 2, 4, 8, and 16. You cannot use this option when you select from a view, but you can use it as part of a **select** clause in a **create view** statement. The procedure **sp_helpcache** shows the valid sizes for the cache an object is bound to or for the default cache.

If Component Integration Services is enabled, you cannot use **prefetch** for remote servers.

lru | mru – specifies the buffer replacement strategy to use for the table. Use **lru** to force the optimizer to read the table into the cache on the MRU/LRU (most recently used/least recently used) chain. Use **mru** to discard the buffer from cache and replace it with the next buffer for the table. You cannot use this option when you select from a view, but you can use it as part of a **select** clause in a **create view** statement.

holdlock – makes a shared lock on a specified table or view more restrictive by holding it until the transaction completes (instead of releasing the shared lock as soon as the required data page is no longer needed, whether or not the transaction has completed).

The **holdlock** option applies only to the table or view for which it is specified, and only for the duration of the transaction defined by the statement in which it is used. Setting the **transaction isolation level 3** option of the **set** command implicitly applies a **holdlock** for each **select** statement within a transaction. The keyword **holdlock** is not permitted in a **select** statement that includes the **for browse** option. You cannot specify both a **holdlock** and a **noholdlock** option in a query.

If Component Integration Services is enabled, you cannot use **holdlock** for remote servers.

noholdlock – prevents the server from holding any locks acquired during the execution of this **select** statement, regardless of the transaction isolation level currently in effect. You cannot specify both a **holdlock** and a **noholdlock** option in a query.

shared – instructs Adaptive Server to use a shared lock (instead of an update lock) on a specified table or view. This allows other clients to obtain an update lock on that table or view. You can use the **shared** keyword only with a **select** clause included as part of a **declare cursor** statement. For example:

```
declare shared_crsr cursor
for select title, title_id
from titles shared
where title_id like "BU%"
```

You can use the **holdlock** keyword in conjunction with **shared** after each table or view name, but **holdlock** must precede **shared**.

search_conditions – used to set the conditions for the rows that are retrieved. A search condition can include column names, expressions, arithmetic operators, comparison operators, the keywords **not**, **like**, **is null**, **and**, **or**, **between**, **in**, **exists**, **any**, and **all**, subqueries, case expressions, or any combination of these items. See "where Clause" for more information.

group by – finds a value for each group. These values appear as new columns in the results, rather than as new rows.

When **group by** is used with standard SQL, each item in the select list must either have a fixed value in every row in the group or be used with aggregate functions, which produce a single value for each group. Transact-SQL has no such restrictions on the items in the select list. Also, Transact-SQL allows you to group by any expression (except by a column alias); with standard SQL, you can group by a column only.

You can use the aggregates listed in Table 1-21 with **group by** (*expression* is almost always a column name):

Table 1-21: Results of using aggregates with group by

Aggregate Function	Result
sum([all | distinct] *expression)*	Total of the values in the numeric column.
avg([all | distinct] *expression)*	Average of the values in the numeric column.
count([all | distinct] *expression)*	Number of (distinct) non-null values in the column.
count(*)	Number of selected rows.
max(*expression)*	Highest value in the column.
min*(expression)*	Lowest value in the column.

See “group by and having Clauses” for more information.

A table can be grouped by any combination of columns—that is, groups can be nested within each other. You cannot group by a column heading; you must use a column name, an expression, or a number representing the position of the item in the select list.

group by all – includes all groups in the results, even those that do not have any rows that meet the search conditions (see “group by and having Clauses” for an example).

aggregate_free_expression – is an expression that includes no aggregates.

having – sets conditions for the **group by** clause, similar to the way that **where** sets conditions for the **select** clause. There is no limit on the number of conditions that can be included.

You can use a **having** clause without a **group by** clause.

If any columns in the select list do not have aggregate functions applied to them and are not included in the query’s **group by** clause (illegal in standard SQL), the meanings of **having** and **where** are somewhat different.

In this situation, a **where** clause restricts the rows that are included in the calculation of the aggregate, but does not restrict the rows returned by the query. Conversely, a **having** clause restricts the rows returned by the query, but does not affect the calculation of the aggregate. See “group by and having Clauses” for examples.

order by – sorts the results by columns. In Transact-SQL, you can use **order by** for items that do not appear in the select list. You can sort by a column name, a column heading (or alias), an expression, or a number representing the position of the item in the **select list** (the *select_list_number*). If you sort by select list number, the columns to which the **order by** clause refers must be included in the select list, and the select list cannot be * (asterisk).

asc – sorts results in ascending order (the default).

desc – sorts results in descending order.

compute – used with row aggregates (**sum**, **avg**, **min**, **max**, and **count**) to generate control break summary values. The summary values appear as additional rows in the query results, allowing you to see detail and summary rows with one statement.

You cannot use a **select into** clause with **compute**.

If you use **compute by,** you must also use an **order by** clause. The columns listed after **compute 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, if the **order by** clause is:

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

the **compute by** clause can be any (or all) of these:

```
compute by a, b, c
compute by a, b
compute by a
```

The keyword **compute** can be used without **by** to generate grand totals, grand counts, and so on. **order by** is optional if you use **compute** without **by.** See “compute Clause” for details and examples.

If Component Integration Services is enabled, you cannot use **compute** for remote servers.

for {**read only** | **update**} – specifies that a cursor result set is read-only or updatable. You can use this option only within a stored procedure and only when the procedure defines a query for a cursor. In this case, the **select** is the only statement allowed in the procedure. It defines the **for read only** or **for update** option (instead of the **declare cursor** statement). This method of declaring cursors provides the advantage of page-level locking while fetching rows.

If the **select** statement in the stored procedure is not used to define a cursor, Adaptive Server ignores the **for read only | update** option. See the Embedded SQL™ documentation for more information about using stored procedures to declare cursors. For information about read-only or updatable cursors, see Chapter 17, "Cursors: Accessing Data Row by Row," in the *Transact-SQL User's Guide*.

of *column_name_list* – is the list of columns from a cursor result set defined as updatable with the **for update** option.

at isolation – specifies the isolation level (0, 1, or 3) of the query. If you omit this clause, the query uses the isolation level of the session in which it executes (isolation level 1 by default). The **at isolation** clause is valid only for single queries or within 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**. If you specify **holdlock**, **noholdlock**, or **shared** in a query that also specifies **at isolation read uncommitted**, Adaptive Server issues a warning and ignores the **at isolation** clause. For the other isolation levels, **holdlock** takes precedence over the **at isolation** clause. For more information about isolation levels, see Chapter 18, "Transactions: Maintaining Data Consistency and Recovery," in the *Transact-SQL User's Guide*.

If Component Integration Services is enabled, you cannot use **at isolation** for remote servers.

read uncommitted – specifies isolation level 0 for the query. You can specify **0** instead of **read uncommitted** with the **at isolation** clause.

read committed – specifies isolation level 1 for the query. You can specify **1** instead of **read committed** with the **at isolation** clause.

serializable – specifies isolation level 3 for the query. You can specify **3** instead of **serializable** with the **at isolation** clause.

for browse – must be attached to the end of a SQL statement sent to Adaptive Server in a DB-Library browse application. See the *Open Client DB-Library Reference Manual* for details.

Examples

1. **`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
```

   Selects all rows and columns from the *publishers* table.

2. **`select pub_id, pub_name, city, state from publishers`**

   Selects all rows from specific columns of the *publishers* table.

3.
   ```
   select "The publisher's name is",
   Publisher = pub_name, pub_id
   from publishers
   ```

```
                       Publisher                      pub_id
----------------------- ---------------------------- ------
The publisher's name is New Age Books                 0736
The publisher's name is Binnet & Hardley              0877
The publisher's name is Algodata Infosystems          1389
```

   Selects all rows from specific columns of the *publishers* table, substituting one column name and adding a string to the output.

4.
   ```
   select type as Type, price as Price
   from titles
   ```

   Selects all rows from specific columns of the *titles* table, substituting column names.

5.
   ```
   select pub_id, total = sum (total_sales)
       into #advance_rpt
   from titles
   where advance < $10000
       and total_sales is not null
   group by pub_id
   having count(*) > 1
   ```

Selects specific columns and rows, placing the results into the temporary table #*advance_rpt*.

6.
```
select "Author_name" = au_fname + " " + au_lname
    into #tempnames
    from authors
```

Concatenates two columns and places the results into the temporary table #*tempnames*.

7.
```
select type, price, advance from titles
order by type desc
compute avg(price), sum(advance) by type
compute sum(price), sum(advance)
```

Selects specific columns and rows, returns the results ordered by type from highest to lowest, and calculates summary information.

8.
```
select type, price, advance from titles
compute sum(price), sum(advance)
```

Selects specific columns and rows, and calculates totals for the *price* and *advance* columns.

9.
```
select * into coffeetabletitles from titles
where price > $20
```

Creates the *coffeetabletitles* table, a copy of the *titles* table which includes only books priced over $20.

10.
```
select * into newtitles from titles
where 1 = 0
```

Creates the *newtitles* table, an empty copy of the *titles* table.

11.
```
select title_id, title
    from titles (index title_id_ind prefetch 16)
    where title_id like "BU%"
```

Gives an optimizer hint.

12.
```
select sales_east.syb_identity,
sales_west.syb_identity
from sales_east, sales_west
```

Selects the IDENTITY column from the *sales_east* and *sales_west* tables by using the *syb_identity* keyword.

13.
```
select *, row_id = identity(10)
into newtitles from titles
```

Creates the *newtitles* table, a copy of the *titles* table with an IDENTITY column.

14.
```
select pub_id, pub_name
from publishers
at isolation read uncommitted
```

Specifies a transaction isolation level for the query.

15.
```
select ord_num from salesdetail
    (index salesdetail parallel 3)
```

Gives an optimizer hint for the parallel degree for the query.

Comments

- The keywords in the **select** statement, as in all other statements, must be used in the order shown in the syntax statement.
- The keyword **all** can be used after **select** for compatibility with other implementations of SQL. **all** is the default. Used in this context, **all** is the opposite of **distinct**. All retrieved rows are included in the results, whether or not some are duplicates.
- Except in **create table**, **create view**, and **select into** statements, column headings may include any characters, including blanks and Adaptive Server keywords, if the column heading is enclosed in quotes. If the heading is not enclosed in quotes, it must conform to the rules for identifiers.
- Column headings in **create table**, **create view**, and **select into** statements, as well as table aliases, must conform to the rules for identifiers.
- 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 would insert all the rows with non-null values into the *advances* table, and produce error messages for all rows where the *advance* column in the *titles* table contained NULL.

  If you cannot make this kind of substitution for your data, you cannot insert data containing null values into the columns with the NOT NULL specification.

Two tables can be identically structured, and yet be different as to whether null values are permitted in some fields. Use **sp_help** to see the null types of the columns in your table.

- The default length of the *text* or *image* data returned with a **select** statement is 32K. Use **set textsize** to change the value. The size for the current session is stored in the global variable *@@textsize*. Certain client software may issue a **set textsize** command on logging into Adaptive Server.
- Data from remote Adaptive Servers can be retrieved through the use of remote procedure calls. See **create procedure** and **execute** for more information.
- A **select** statement used in a cursor definition (through **declare cursor**) must contain a **from** clause, but it cannot contain a **compute**, **for browse**, or **into** clause. If the **select** statement contains any of the following constructs, the cursor is considered read-only and not updatable:
  - **distinct** option
  - **group by** clause
  - Aggregate functions
  - **union** operator

  If you declare a cursor inside a stored procedure with a **select** statement that contains an **order by** clause, that cursor is also considered read-only. Even if it is considered updatable, you cannot delete a row using a cursor that is defined by a **select** statement containing a join of two or more tables. See **declare cursor** for more information.
- If a **select** statement that assigns a value to a variable returns more than one row, the last returned value is assigned to the variable. For example:

```
declare @x varchar(40)
select @x = pub_name from publishers
print @x
```

```
(3 rows affected)
Algodata Infosystems
```

Using *select into*

- **select into** is a two-step operation. The first step creates the new table, and the second step inserts the specified rows into the new 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. Use the **drop table** statement to remove the new table, and then reissue the **select into** statement.

- The name of the new table must be unique in the database and must conform to the rules for identifiers. You can also **select into** temporary tables (see examples 5, 6, and 9).
- You cannot **select into** a table that already exists. Instead, use the **insert...select** command. See **insert** for more information.
- Any rules, constraints, or defaults associated with the base table are not carried over to the new table. Bind rules or defaults to the new table using **sp_bindrule** and **sp_bindefault**.
- **select into** does not carry over the base table's **max_rows_per_page** value, and it creates the new table with a **max_rows_per_page** value of 0. Use **sp_chgattribute** to set the **max_rows_per_page** value.
- The **select into/bulkcopy/pllsort** option must be set to **true** (by executing **sp_dboption**) in order to **select into** a permanent table. You do not have to set the **select into/bulkcopy/pllsort** option to **true** in order to **select into** a temporary table, since the temporary database is never recovered.

  Once you have used **select into** in a database, you must perform a full database dump before you can use the **dump transaction** command. **select into** operations are not logged; therefore, changes are not recoverable from transaction logs. In this situation, issuing the **dump transaction** statement produces an error message instructing you to use **dump database** instead.

  By default, the **select into/bulkcopy/pllsort** option is set to **false** in newly created databases. To change the default situation, set this option to **true** in the *model* database.
- **select into** runs more slowly while a **dump database** is taking place.
- You can use **select into** to create a duplicate table with no data by having a false condition in the **where** clause (see example 10).
- You must provide a column heading for any column in the select list that contains an aggregate function or any expression. The use

of any constant, arithmetic or character expression, built-in functions, or concatenation in the select list requires a column heading for the affected item. The column heading must be a valid identifier or must be enclosed in quotation marks (see examples 5 and 6).

- Because functions allow null values, any column in the select list that contains a function other than **convert** or **isnull** allows null values.
- You cannot use **select into** inside a user-defined transaction or in the same statement as a **compute** clause.
- To select an IDENTITY column into a result table, include the column name (or the **syb_identit**y keyword) in the **select** statement's *column_list*. The new column observes the following rules:
  - 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.
  - If an IDENTITY column is selected as part of an expression, the resulting column does not inherit the IDENTITY property. It is created as NULL if any column in the expression allows nulls; otherwise, it is created as NOT NULL.
  - If the **select** statement contains a **group by** clause or aggregate function, the resulting column does not inherit the IDENTITY property. Columns that include an aggregate of the IDENTITY column are created NULL; others are NOT NULL.
  - 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 defined as NOT NULL.
- You cannot use **select into** to create a new table with multiple IDENTITY columns. If the **select** statement includes both an existing IDENTITY column and a new IDENTITY specification of the form *column_name* = **identity***(precision)*, the statement fails.
- If Component Integration Services is enabled, and if the **into** table resides on Adaptive Server, Adaptive Server uses bulk copy routines to copy the data into the new table. Before doing a **select into** with remote tables, set the **select into/bulkcopy** database option to **true**.
- For information about the Embedded SQL command **select into** *host_var_list*, see the *Open Client Embedded SQL Reference Manual.*

Using *index*, *prefetch*, and *lru | mru*

- The **index**, **prefetch** and **lru | mru** options specify the index, cache and I/O strategies for query execution. These options override the choices made by the Adaptive Server optimizer. Use them with caution, and always check the performance impact with **set statistics io on**. See the *Performance and Tuning Guide* for more information before using these options.

Using *parallel*

- The **parallel** option reduces the number of worker threads that the Adaptive Server optimizer can use for parallel processing. The *degree_of_parallelism* cannot be greater than the configured **max parallel degree**. If you specify a value that is greater than the configured **max parallel degree**, the optimizer ignores the **parallel** option.
- When multiple worker processes merge their results, the order of rows that Adaptive Server returns may vary from one execution to the next. To get rows from partitioned tables in a consistent order, use an **order** by clause, or override parallel query execution by using **parallel 1** in the **from** clause of the query.
- A **from** clause specifying **parallel** is ignored if any of the following conditions is true:
  - The select statement is used for an update or insert.
  - The **from** clause is used in the definition of a cursor.
  - **parallel** is used in the **from** clause within any inner query blocks of a subquery.
  - The select statement creates a view.
  - The table is the inner table of an outer join.
  - The query specifies **min** or **max** on the table and specifies an index.
  - An unpartitioned clustered index is specified or is the only **parallel** option.
  - The query specifies **exists** on the table.
  - The value for the configuration parameter **max scan parallel degree** is 1 and the query specifies an index.
  - A nonclustered index is covered. (For information on index covering, see “Query Processing and Page Reads” in Chapter 3, “Data Storage,” of the Performance and Tuning Guide).

- The table is a system table or a virtual table.
- The query is processed using the OR strategy. (For an explanation of the OR strategy, see "How or Clauses Are Processed" in Chapter 8, "Understanding the Query Optimizer," of the *Performance and Tuning Guide*.)
- The query will return a large number of rows to the user.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The following are Transact-SQL extensions: • **select into** to create a new table • **compute** clauses • Global and local variables • **index** clause, **prefetch, parallel** and **lru | mru** • **holdlock**, **noholdlock**, and **shared** keywords • "*column_heading = column_name*" • Qualified table and column names • **select** in a **for browse** clause • The use, within the select list, of columns that are not in the **group by** list and have no aggregate functions

Permissions

select permission defaults to the owner of the table or view, who can transfer it to other users.

See Also

Commands	**compute Clause, create trigger, delete, group by and having Clauses, insert, order by Clause, set, union Operator, update, where Clause**
Functions	**avg**, **count**, **isnull**, **max**, **min**, **sum**
System procedures	**sp_cachestrategy, sp_dboption**

set

Function

Sets Adaptive Server query-processing options for the duration of the user's work session. Can be used to set some options inside a trigger or stored procedure. Can also be used to activate or deactivate a role in the current session.

Syntax

```
set ansinull {on | off}

set ansi_permissions {on | off}

set arithabort [arith_overflow | numeric_truncation]
   {on | off}

set arithignore [arith_overflow] {on | off}

set {chained, close on endtran, nocount, noexec,
   parseonly, procid, self_recursion, showplan,
   sort_resources} {on | off}

set char_convert {off | on [with {error | no_error}] |
   charset [with {error | no_error}]}

set cis_rpc_handling {on | off}

set cursor rows number for cursor_name

set {datefirst number, dateformat format,
   language language}

set fipsflagger {on | off}

set flushmessage {on | off}

set identity_insert [database.[owner.]]table_name
   {on | off}

set offsets {select, from, order, compute, table,
   procedure, statement, param, execute} {on | off}

set parallel_degree number

set prefetch [on|off]

set process_limit_action {abort | quiet | warning}

set proxy login_name

set quoted_identifier {on | off}

set role {"sa_role" | "sso_role" | "oper_role" |
   role_name [with passwd "password"]} {on | off}

set {rowcount number, textsize number}
set scan_parallel_degree number
set session authorization login_name
set statistics {io, subquerycache, time} {on | off}
set string_rtruncation {on | off}
set table count number
set textsize {number}
set transaction isolation level {0 | 1 | 3}
set transactional_rpc {on | off}
```

Keywords and Options

ansinull – determines whether or not evaluation of NULL-valued operands in SQL equality (=) or inequality (!=) comparisons or aggregate functions, also called **set functions**, is compliant with the SQL92 standard. When you use **set ansinull on**, Adaptive Server generates a warning each time an aggregate function eliminates a null-valued operand from calculation. This option does not affect how Adaptive Server evaluates NULL values in other kinds of SQL statements such as **create table**.

The SQL standards requires that if either one of the two operands of an equality comparison is NULL, the result is UNKNOWN. Transact-SQL treats NULL values differently. If one of the operands is a column, parameter, or variable, and the other operand is the NULL constant or a parameter or variable whose value is NULL, the result is either TRUE or FALSE.

ansi_permissions – determines whether SQL92 permission requirements for **delete** and **update** statements are checked. The default is **off**. Table 1-22 summarizes permission requirements:

Table 1-22: Permissions required for update and delete

Command	Permissions Required with *set ansi_permissions off*	Permissions Required with *set ansi_permissions on*
update	• **update** permission on columns where values are being set	• **update** permission on columns where values are being set • **select** permission on all columns appearing in **where** clause • **select** permission on all columns on right side of **set** clause

Table 1-22: Permissions required for update and delete (continued)

Command	Permissions Required with *set ansi_permissions off*	Permissions Required with *set ansi_permissions on*
delete	• **delete** permission on table	• **delete** permission on table • **select** permission on all columns appearing in **where** clause

arithabort – determines how Adaptive Server behaves when an arithmetic error occurs. The two **arithabort** options, **arithabort arith_overflow** and **arithabort numeric_truncation**, handle different types of arithmetic errors. You can set each option independently or set both options with a single **set arithabort on** or **set arithabort off** statement.

- **arithabort arith_overflow** specifies Adaptive Server's behavior following a divide-by-zero error or a loss of precision during an explicit or implicit datatype conversion. This type of error is serious. 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; however, Adaptive Server does not execute any 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 Adaptive Server's behavior following a loss of scale by an exact numeric type during an implicit datatype conversion. (When an explicit conversion results in a loss of scale, the results are truncated without warning.) The default setting, **arithabort numeric_truncation on**, aborts the statement that causes the error, but Adaptive Server 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.

arithignore arith_overflow – determines whether Adaptive Server displays a message after a divide-by-zero error or a loss of precision. By default, the **arithignore** option is set to **off**. This causes Adaptive Server to display a warning message after any query that results in numeric overflow. To have Adaptive Server ignore overflow errors, use **set arithignore on**. You can omit the optional **arith_overflow** keyword without any effect.

chained – begins a transaction just before the first data retrieval or data modification statement at the beginning of a session and after a transaction ends. In chained mode, Adaptive Server implicitly executes a **begin transaction** command before the following statements: **delete**, **fetch**, **insert**, **open**, **select**, and **update**. You cannot execute **set chained** within a transaction.

char_convert – enables or disables character set conversion between Adaptive Server and a client. If the client is using Open Client DB-Library release 4.6 or later, and the client and server use different character sets, conversion is turned on during the login process and is set to a default based on the character set the client is using. You can also use **set char_convert** *charset* to start conversion between the server character set and a different client character set.

charset can be either the character set's ID or a name from *syscharsets* with a *type* value of less than 2000.

set char_convert off turns conversion off so that characters are sent and received unchanged. **set char_convert on** turns conversion on if it is turned off. If character set conversion was not turned on during the login process or by the **set char_convert** command, **set char_convert on** generates an error message.

When the **with no_error** option is included, Adaptive Server does not notify an application when characters from Adaptive Server cannot be converted to the client's character set. Error reporting is initially turned on when a client connects with Adaptive Server: if you do not want error reporting, you must turn it off for each session with **set char_convert {on | charset} with no_error.** To turn error reporting back on within a session, use **set char_convert {on | charset} with error.**

Whether or not error reporting is turned on, the bytes that cannot be converted are replaced with ASCII question marks (?).

See Chapter 14, "Configuring Client/Server Character Set Conversions," in the *System Administration Guide* for a more complete discussion of error handling in character set conversion.

cis_rpc_handling – determines whether Component Integration Services handles outbound remote procedure call (RPC) requests by default.

close on endtran – causes Adaptive Server to close all cursors opened within a transaction at the end of that transaction. A transaction ends by the use of either the **commit** or **rollback** statement. However, only cursors declared within the scope that sets this option (stored procedure, trigger, and so on) are affected. See Chapter 17, “Cursors: Accessing Data Row by Row,” in the *Transact-SQL User’s Guide* for more information about cursor scopes.

For more information about the evaluated configuration, see Chapter 1, “Overview of Security Features,” in the *Security Administration Guide*.

cursor rows – causes Adaptive Server to return the *number* of rows for each cursor **fetch** request from a client application. The *number* can be a numeric literal with no decimal point or a local variable of type *integer*. If the *number* is less than or equal to zero, the value is set to 1. You can set the **cursor rows** option for a cursor, whether it is open or closed. However, this option does not affect a **fetch** request containing an **into** clause. *cursor_name* specifies the cursor for which to set the number of rows returned.

datefirst – sets the first week day to a number from 1 to 7. The us_english language default is 1 (Sunday).

dateformat – sets the order of the date parts *month/day/year* for entering *datetime* or *smalldatetime* data. Valid arguments are *mdy*, *dmy*, *ymd*, *ydm*, *myd*, and *dym*. The us_english language default is *mdy*.

fipsflagger – determines whether Adaptive Server displays a warning message when Transact-SQL extensions to entry level SQL92 are used. By default, Adaptive Server does not tell you when you use nonstandard SQL.

flushmessage – determines when Adaptive Server returns messages to the user. By default, messages are stored in a buffer until the query that generated them is completed or the buffer is filled to capacity. Use **set flushmessage on** to return messages to the user immediately, as they are generated.

identity_insert – determines whether explicit inserts into a table’s IDENTITY column are allowed. (Note that updates to an IDENTITY column are never allowed.) This option can be used only with base tables. It cannot be used with views or set within a trigger.

Setting **identity_insert** *table_name* **on** allows the table owner, Database Owner, or System Administrator to explicitly insert a value into an IDENTITY column. Inserting a value into the IDENTITY column allows you to specify a seed value for the column or to restore a row that was deleted in error. Unless you have created a unique index on the IDENTITY column, Adaptive Server does not verify the uniqueness of the inserted value; you can insert any positive integer.

The table owner, Database Owner, or System Administrator can use **set identity_insert** *table_name* **on** on a table with an IDENTITY column in order to enable the manual insertion of a value into an IDENTITY column. However, only the following users can actually insert a value into an IDENTITY column, when **identity_insert** is **on**:

- Table owner
- Database Owner, if granted explicit **insert** permission on the column by the table owner
- Database Owner, impersonating the table owner by using the **setuser** command

Setting **identity_insert***table_name* **off** restores the default behavior by prohibiting explicit inserts to IDENTITY columns. At any time, you can use **set identity_insert** *table_name* **on** for a single database table within a session.

language – is the official name of the language that displays system messages. The language must be installed on Adaptive Server. The default is us_english.

nocount – controls the display of rows affected by a statement. **set nocount off** disables the display of rows; **set nocount on** reenables the count of rows.

noexec – compiles each query but does not execute it. **noexec** is often used with **showplan**. After you set **noexec on**, no subsequent commands are executed (including other **set** commands) until you set **noexec off**.

offsets – returns the position of specified keywords (with relation to the beginning of the query) in Transact-SQL statements. The keyword list is a comma-separated list that can include any of the following Transact-SQL constructs: **select**, **from**, **order**, **compute**, **table**, **procedure**, **statement**, **param**, and **execute**. Adaptive Server returns offsets if there are no errors. This option is used in Open Client DB-Library only.

parallel_degree – specifies an upper limit for the number of worker processes used in the parallel execution of a query. This number must be less than or equal to the number of worker processes per query, as set by the **max parallel degree** configuration parameter. The *@@parallel_degree* global variable stores the current setting.

parseonly – checks the syntax of each query and returns any error messages without compiling or executing the query. Do not use **parseonly** inside a stored procedure or trigger.

prefetch – enables or disables large I/Os to the data cache.

process_limit_action – specifies whether Adaptive Server executes parallel queries when an insufficient number of worker processes are available. Under these circumstances, when **process_limit_action** is set to **quiet**, Adaptive Server silently adjusts the plan to use a degree of parallelism that does not exceed the number of available processes. If **process_limit_action** is set to **warning** when an insufficient number of worker processes are available, Adaptive Server issues a warning message when adjusting the plan; and if **process_limit_action** is set to **abort**, Adaptive Server aborts the query and issues an explanatory message an insufficient number of worker processes are available.

procid – returns the ID number of the stored procedure to Open Client DB-Library/C (not to the user) before sending rows generated by the stored procedure.

proxy – allows you to assume the permissions, login name, and *suid* (server user ID) of *login_name*. For *login_name*, specify a valid login from *master..syslogins*, enclosed in quotation marks. To revert to your original login name and *suid*, use **set proxy** with your original *login_name*.

See "Using Proxies" for more information.

quoted_identifier – determines whether Adaptive Server recognizes delimited identifiers. By default, **quoted_identifier** is **off** and all identifiers must conform to the rules for valid identifiers. If you use **set quoted_identifier on**, you can use table, view, and column names that begin with a non-alphabetic character, include characters that would not otherwise be allowed, or are reserved words, by enclosing the identifiers within double quotation marks. Delimited identifiers cannot exceed 28 bytes, may not be recognized by all front-end products, and may produce unexpected results when used as parameters to system procedures.

When **quoted_identifier** is **on**, all character strings enclosed within double quotes are treated as identifiers. Use single quotes around character or binary strings.

role – turns the specified role on or off during the current session. When you log in, all system roles that have been granted to you are turned on. Use **set role *role_name* off** to turn a role off, and **set role *role_name* on** to turn it back on again, as needed. System roles are **sa_role**, **sso_role**, and **oper_role**. If you are not a user in the current database, and if there is no “guest” user, you cannot set **sa_role off**, because there is no server user ID for you to assume.

role_name – is the name of any user-defined role created by the System Security Officer. User-defined roles are not turned on by default. To set user-defined roles to activate at login, the user or the System Security Officer must use **set role on**.

with *passwd* – specifies the password to activate the role. If a user-defined role has an attached password, you must specify the password to activate the role.

rowcount – causes Adaptive Server to stop processing the query (**select**, **insert**, **update**, or **delete**) after the specified number of rows are affected. The *number* can be a numeric literal with no decimal point or a local variable of type *integer*. To turn this option off, use:

```
set rowcount 0
```

scan_parallel_degree – specifies the maximum session-specific degree of parallelism for hash-based scans (parallel index scans and parallel table scans on nonpartitioned tables). This number must be less than or equal to the current value of the **max scan parallel degree** configuration parameter. The *@@scan_parallel_degree* global variable stores the current setting.

self_recursion – determines whether Adaptive Server allows triggers to cause themselves to fire again (this is called **self-recursion**). By default, Adaptive Server does not allow self recursion in triggers. You can turn this option on only for the duration of a current client session; its effect is limited by the scope of the trigger that sets it. For example, if the trigger that sets **self_recursion on** returns or causes another trigger to fire, this option reverts to **off**. This option works only within a trigger and has no effect on user sessions.

session authorization – is identical to **set proxy,** with this exception: **set session authorization** follows the SQL standard, while **set proxy** is a Transact-SQL extension.

See "Using Proxies" for more information.

showplan – generates a description of the processing plan for the query. The results of **showplan** are of use in performance diagnostics. **showplan** does not print results when it is used inside a stored procedure or trigger. See Chapter 9, "Understanding Query Plans," in the *Performance and Tuning Guide* for more information. For parallel queries, **showplan** output also includes the adjusted query plan at run-time, if applicable. See Chapter 14, "Parallel Query Optimization," in the *Performance and Tuning Guide* for more information.

sort_resources – generates a description of the sorting plan for a **create index** statement. The results of **sort_resources** are of use in determining whether a sort operation will be done serially or in parallel. When **sort_resouces** is **on**, Adaptive Server prints the sorting plan but does not execute the **create index** statement. See Chapter 15, "Parallel Sorting," in the *Performance and Tuning Guide* for more information.

statistics io – displays the following statistics information for each table referenced in the statement:

- the number of times the table is accessed (scan count)
- the number of logical reads (pages accessed in memory)
- and the number of physical reads (database device accesses)

For each command, **statistics io** displays the number of buffers written.

If Adaptive Server has been configured to enforce resource limits, **statistics io** also displays the total I/O cost. See "Indexes and I/O Statistics" in Chapter 7, "Indexing for Performance," of the *Performance and Tuning Guide* for more information.

statistics subquerycache – displays the number of cache hits, misses, and the number of rows in the subquery cache for each subquery.

statistics time – displays the amount of time Adaptive Server used to parse and compile for each command. For each step of the command, **statistics time** displays the amount of time Adaptive Server used to execute the command. Times are given in milliseconds and timeticks, the exact value of which is machine-dependent.

string_rtruncation – determines whether Adaptive Server raises a SQLSTATE exception when an **insert** or **update** command truncates a *char* or *varchar* string. If the truncated characters consist only of spaces, no exception is raised. The default setting, **off**, does not raise the SQLSTATE exception, and the character string is silently truncated.

table count – sets the number of tables that Adaptive Server will consider at one time while optimizing a join. The default is 4. Valid values are 1–8. A value greater than 8 defaults to 8. **table count** may improve the optimization of certain join queries, but it increases the compilation cost.

textsize – specifies the maximum size in bytes of *text* or *image* type data that is returned with a **select** statement. The *@@textsize* global variable stores the current setting. To reset **textsize** to the default size (32K), use the command:

```
set textsize 0
```

The default setting is 32K in **isql**. Some client software sets other default values.

transaction isolation level – sets the transaction isolation level for your session to 0, 1, or 3. After you set this option, any current or future transactions operate at that isolation level. By default, Adaptive Server's transaction isolation level is 1, which allows shared read locks on data.

Scans at isolation level 0 do not acquire any locks. Therefore, 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 the 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 cause the scan to be aborted before completion.

If you specify isolation level 3, Adaptive Server applies a **holdlock** to all **select** and **readtext** operations in a transaction, which holds the queries' read locks until the end of that transaction. If you also set chained mode, that isolation level remains in effect for any data retrieval or modification statement that implicitly begins a transaction.

transactional_rpc – controls the handling of remote procedure calls. If this option is set to **on**, when a transaction is pending, the RPC is handled by the Component Integration Services access methods. If this option is set to **off**, the remote procedure call is handled by the Adaptive Server site handler.

Examples

1.
```
set showplan, noexec on
go
select * from publishers
go
```

For each query, returns a description of the processing plan, but does not execute it.

2.
```
set textsize 100
```

Sets the limit on *text* or *image* data returned with a **select** statement to 100 bytes.

3.
```
set rowcount 4
```

For each **insert**, **update**, **delete**, and **selec**t statement, Adaptive Server stops processing the query after it affects the first four rows. For example:

```
select title_id, price from titles

title_id  price
--------  ----------
BU1032         19.99
BU1111         11.95
BU2075          2.99
BU7832         19.99

(4 rows affected)
```

4.
```
set char_convert on with error
```

Activates character set conversion, setting it to a default based on the character set the client is using. Adaptive Server also notifies the client or application when characters cannot be converted to the client's character set.

5.
```
set proxy "mary"
```

The user executing this command now operates within the server as the login “mary” and Mary’s server user ID.

6. **set session authorization "mary"**

   An alternative way of stating example 5.

7. **set cursor rows 5 for test_cursor**

   Returns five rows for each succeeding **fetch** statement requested by a client using *test_cursor*.

8.
   ```
   set identity_insert stores_south on
   go
   insert stores_south (syb_identity)
   values (100)
   go
   set identity_insert stores_south off
   go
   ```

   Inserts a value of 100 into the IDENTITY column of the *stores_south* table, then prohibits further explicit inserts into this column. Note the use of the **syb_identity** keyword; Adaptive Server replaces the keyword with the name of the IDENTITY column.

9. **set transaction isolation level 3**

   Implements read-locks with each **select** statement in a transaction for the duration of that transaction.

10. **set role "sa_role" off**

    Deactivates the user’s System Administrator role for the current session.

11. **set fipsflagger on**

    Tells Adaptive Server to display a warning message if you use a Transact-SQL extension. Then, if you use nonstandard SQL, like this:

    ```
    use pubs2
    go
    ```

    Adaptive Server displays:

    ```
    SQL statement on line number 1 contains Non-ANSI
    text. The error is caused due to the use of use
    database.
    ```

12. **set ansinull on**

Tells Adaptive Server to evaluate NULL-valued operands of equality (=) and inequality (!=) comparisons and aggregate functions in compliance with the entry level SQL92 standard.

When you use **set ansinull on**, aggregate functions and row aggregates raise the following SQLSTATE warning when Adaptive Server finds null values in one or more columns or rows:

```
Warning - null value eliminated in set function
```

If the value of either the equality or the inequality operands is NULL, the comparison's result is UNKNOWN. For example, the following query returns no rows in **ansinull** mode:

```
select * from titles where price = null
```

If you use **set ansinull off**, the same query returns rows in which *price* is NULL.

13. **`set string_rtruncation on`**

    Causes Adaptive Server to generate an exception when truncating a *char* or *nchar* string. If an **insert** or **update** statement would truncate a string, Adaptive Server displays:

    ```
    string data, right truncation
    ```

14.
    ```
    set quoted_identifier on
    go
    create table "!*&strange_table"
         ("emp's_name" char(10),
         age int)
    go
    set quoted_identifier off
    go
    ```

    Tells Adaptive Server to treat any character string enclosed in double quotes as an identifier. The table name "!*&strange_table" and the column name "emp's_name" are legal identifier names while **quoted_identifier** is **on**.

15. **`set cis_rpc_handling on`**

    Specifies that Component Integration Services handles outbound RPC requests by default.

16. **`set transactional_rpc on`**

    Specifies that when a transaction is pending, the RPC is handled by the Component Integration Services access methods rather than by the Adaptive Server site handler.

17. **`set role doctor_role on`**

Activates the "doctor" role. This command is used by users to specify the roles they want activated.

18. **`set role doctor_role with passwd "physician" on`**

    Activates the "doctor" role when the user enters the password.

19. **`set role doctor_role off`**

    Deactivates the "doctor" role.

20. **`set scan_parallel_degree 4`**

    Specifies a maximum degree of parallelism of 4 for parallel index scans and parallel table scans on nonpartitioned tables.

Comments

- Some **set** options can be grouped together, as follows:
  - **parseonly, noexec**, **prefetch**, **showplan**, **rowcount**, and **nocount** control the way a query is executed. It does not make sense to set both **parseonly** and **noexec** on. The default setting for **rowcount** is 0 (return all rows); the default for the others is **off**.
  - The **statistics** options display performance statistics after each query. The default setting for the **statistics** options is **off**.

    For more information about **parseonly, noexec**, **prefetch**, **showplan** and **statistics**, see the *Performance and Tuning Guide*.
  - **offsets** and **procid** are used in DB-Library to interpret results from Adaptive Server. The default setting for these options is **on**.
  - **datefirst**, **dateformat**, and **language** affect date functions, date order, and message display. If used within a trigger or stored procedure, these options do not revert to their previous settings.

    In the default language, us_english, **datefirst** is 1 (Sunday), **dateformat** is *mdy*, and messages are displayed in us_english. Some language defaults (including us_english) produce Sunday=1, Monday=2, and so on; others produce Monday=1, Tuesday=2, and so on.

    set language implies that Adaptive Server should use the first weekday and date format of the language it specifies, but does not override an explicit **set datefirst** or **set dateformat** command issued earlier in the current session.
  - **cursor rows** and **close on endtran** affect the way Adaptive Server handles cursors. The default setting for **cursor rows** with all cursors is 1. The default setting for **close on endtran** is **off**.

- **chained** and **transaction isolation level** allow Adaptive Server to handle transactions in a way that is compliant with the SQL standards.
- **fipsflagger**, **string_rtruncation**, **ansinull**, **ansi_permissions**, **arithabort**, and **arithignore** affect aspects of Adaptive Server error handling and compliance to SQL standards.

➤ ***Note***

The **arithabort** and **arithignore** options were redefined for release 10.0 and later. If you use these options in your applications, examine them to be sure they are still producing the desired effect.

- You can only use the **cis_rpc_handling** and **transactional_rpc** options when Component Integration Services is enabled.
- **parallel_degree** and **scan_parallel_degree** limit the degree of parallelism for queries, if Adaptive Server is configured for parallelism. When you use these options, you give the optimizer a hint to limit parallel queries to use fewer worker processes than allowed by the configuration parameters. Setting these parameters to 0 restores the server-wide configuration values.

  If you specify a number that is greater than the numbers allowed by the configuration parameters, Adaptive Server issues a warning message and uses the value set by the configuration parameter.

• If you use the **set** command inside a trigger or stored procedure, most **set** options revert to their former settings after the trigger or procedure executes.

  The **datefirst**, **dateformat**, and **language** settings do not revert to their former settings after the procedure or trigger executes, but remain for the entire Adaptive Server session or until you explicitly reset them.

• If you specify more than one **set** option, the first syntax error causes all following options to be ignored. However, the options specified before the error are executed, and the new option values are set.

• All **set** options except **showplan** and **char_convert** take effect immediately. **showplan** takes effect in the following batch. Here are two examples that use **set showplan on**:

```
set showplan on
select * from publishers
go
```

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

(3 rows affected)
```

But:

```
set showplan on
go
select * from publishers
go
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
    STEP 1
        The type of query is SELECT

        FROM TABLE
            publishers
        Nested iteration
        Table Scan
        Ascending Scan.
        Positioning at start of table.

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

(3 rows affected)
```

Roles and *set* Options

- When you log into Adaptive Server, all system-defined roles granted to you are automatically activated. User-defined roles granted to you are not automatically activated. To automatically activate user-defined roles granted to you, use **sp_modifylogin**. For information on how to use **sp_modifylogin**, see **sp_modifylogin** in your *Adaptive Server Reference Manual*. Use **set role** *role_name* **on** or **set role** *role_name* **off** to turn roles on and off.

  For example, if you have been granted the System Administrator role, you assume the identity (and user ID) of Database Owner in the current database. To assume your real user ID, execute this command:

```
set role "sa_role" off
```

If you are not a user in the current database, and if there is no "guest" user, you cannot set **sa_role off**.

- If the user-defined role you intend to activate has an attached password, you must specify the password to turn the role on. Thus, you would enter:

  ```
  set role "role_name" with passwd "password" on
  ```

Component Integration Services and *set* Options

- You can enable or disable Component Integration Services RPC handling for an application with **set cis_rpc_handling on**. If you enabled global configuration with **sp_configure "cis rpc handling" 1**, you cannot disable it with **set cis_rpc_handling off**. If you disable the global property, you can use **set cis_rpc_handling** to enable or disable the capability as required.
- When Component Integration Services is enabled, you can place RPCs within transactions. These RPCs are called **transactional RPCs**. A transactional RPC is an RPC whose work can be included in the context of a current transaction. This remote unit of work can be committed or rolled back along with the work performed by the local transaction.

  To use transactional RPCs, enable Component Integration Services with **sp_configure**, and then issue the **set transactional_rpc** command. When **set transactional_rpc** is **on** and a transaction is pending, the RPC is handled by the Component Integration Services access methods rather than by the Adaptive Server site handler.

  The **set transactional_rpc** command default is **off**. The **set cis_rpc_handing** command overrides the **set transactional_rpc** command. If you set **cis_rpc_handling on**, all outbound RPCs are handled by Component Integration Services.
- See the *Component Integration Services User's Guide* for a discussion of using **set transactional_rpc**, **set cis_rpc_handling**, and **sp_configure**.

Using Proxies

- Before you can use the **set proxy** or **set session authorization** command, a System Security Officer must grant permission to execute **set proxy** or **set session authorization** from the *master* database.
- Executing **set proxy** or **set session authorization** with the original *login_name* reestablishes your previous identity.
- You cannot execute **set proxy** or **set session authorization** from within a transaction.
- Adaptive Server permits only one level of login identity change. Therefore, after you use **set proxy** or **set session authorization** to change identity, you must return to your original identity before changing it again. For example, assume that your login name is "ralph". You want to create a table as "mary", create a view as "joe", and then return to your own login identity. Execute:

```
set proxy "mary"
    create table mary_sales
    (stor_id  char(4),
    ord_num   varchar(20),
    date      datetime)
grant select on mary_sales to all

set proxy "ralph"

set proxy "joe"
    create view joes_view (publisher, city, state)
    as select stor_id, ord_num, date
    from mary_sales

set proxy "ralph"
```

SQL92 Compliance

- The SQL92 standard specifies behavior that differs from Transact-SQL behavior in earlier Adaptive Server releases. Compliant behavior is enabled by default for all Embedded-SQL precompiler applications. Other applications needing to match this standard of behavior can use the **set** options listed in Table 1-23.

Table 1-23: Options to set for entry level SQL92 compliance

Option	Setting
ansi_permissions	on
ansinull	on

Table 1-23: Options to set for entry level SQL92 compliance (continued)

Option	Setting
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**

Global Variables Affected by *set* Options

- Table 1-24 lists the global variables that contain information about the session options controlled by the **set** command.

Table 1-24: 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. *@@rowcount* is updated even when **nocount** is on.
@scan_parallel_degree	Contains the current maximum parallel degree setting for nonclustered index scans.

Table 1-24: Global variables containing session options (continued)

Global Variable	Description
@@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.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

In general, **set** permission defaults to all users and no special permissions are required to use it. Exceptions include **set role**, **set proxy**, and **set session authorization.**

To use **set role**, a System Administrator or System Security Officer must have granted you the role. If you gain entry to a database only because you have a certain role, you cannot turn that role off while you are using the database. For example, if you are not normally authorized to use a database *info_plan*, but you use it as a System Administrator, Adaptive Server returns an error message if you try to set **sa_role off** while you are still in *info_plan*.

To use **set proxy** or **set session authorization**, you must have been granted permission by a System Security Officer.

See Also

Commands	**create trigger**, **fetch**, **insert**, **grant**, **revoke**
Functions	**convert**

setuser

Function

Allows a Database Owner to impersonate another user.

Syntax

```
setuser ["user_name"]
```

Examples

1.
```
setuser "mary"
go
grant select on authors to joe
setuser
go
```

   The Database Owner temporarily adopts Mary's identity in the database in order to grant Joe permissions on *authors*, a table owned by Mary.

Comments

- The Database Owner uses **setuser** to adopt the identity of another user in order to use another user's database object, to grant permissions, to create an object, or for some other reason.
- When the Database Owner uses the **setuser** command, Adaptive Server checks the permissions of the user being impersonated instead of the permissions of the Database Owner. The user being impersonated must be listed in the *sysusers* table of the database.
- **setuser** affects permissions only in the local database. It does not affect remote procedure calls or accessing objects in other databases.
- The **setuser** command remains in effect until another **setuser** command is given or until the current database is changed with the **use** command.
- Executing the **setuser** command with no user name reestablishes the Database Owner's original identity.
- System Administrators can use **setuser** to create objects that will be owned by another user. However, since a System Administrator operates outside the permissions system, she or he cannot use **setuser** to acquire another user's permissions.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

setuser permission defaults to the Database Owner and is not transferable.

See Also

Commands	**grant, revoke, use**

shutdown

Function

Shuts down the Adaptive Server from which the command is issued, its local Backup Server, or a remote Backup Server. This command can be issued only by a System Administrator.

Syntax

shutdown [*srvname*] [with {wait | nowait}]

Keywords and Options

srvname – is the logical name by which the Backup Server is known in the Adaptive Server's *sysservers* system table. This parameter is not required when shutting down the local Adaptive Server.

with wait – is the default. This shuts down the Adaptive Server or Backup Server gracefully.

with nowait – shuts down the Adaptive Server or Backup Server immediately, without waiting for currently executing statements to finish.

➤ ***Note***

Use of **shutdown with nowai**t can lead to gaps in IDENTITY column values.

Examples

1. **shutdown**

   Shuts down the Adaptive Server from which the **shutdown** command is issued.

2. **shutdown with nowait**

   Shuts down the Adaptive Server immediately.

3. **shutdown SYB_BACKUP**

   Shuts down the local Backup Server.

4. **shutdown REM_BACKUP**

   Shuts down the remote Backup Server REM_BACKUP.

Comments

- Unless you use the **nowait** option, **shutdown** attempts to bring Adaptive Server down gracefully by:
  - Disabling logins (except for the System Administrator)
  - Performing a checkpoint in every database
  - Waiting for currently executing SQL statements or stored procedures to finish

  Shutting down the server without the **nowait** option minimizes the amount of work that must be done by the automatic recovery process.
- Unless you use the **nowait** option, **shutdown** *backup_server* waits for active dumps and / or loads to complete. Once you issue a **shutdown** command to a Backup Server, no new dumps or loads that use this Backup Server can start.
- Use **shutdown with nowait** only in extreme circumstances. In Adaptive Server, issue a **checkpoint** command before executing a **shutdown with nowait.**
- You can halt only the local Adaptive Server with **shutdown**; you cannot halt a remote Adaptive Server.
- You can halt a Backup Server only if:
  - It is listed in your *sysservers* table. The system procedure **sp_addserver** adds entries to *sysservers.*
  - It is listed in the interfaces file for the Adaptive Server where you execute the command.
- Use the **sp_helpserver** system procedure to determine the name by which a Backup Server is known to the Adaptive Server. Specify the Backup Server's *name*—not its *network_name*—as the *srvname* parameter. For example:

  `sp_helpserver`

```
name        network_name   status                              id
----------  ------------   ----------------------------------- --
REM_BACKUP  WHALE_BACKUP   timeouts, no net password encryption 3
SYB_BACKUP  SLUG_BACKUP    timeouts, net password encryption    1
eel         eel                                                 0
whale       whale          timeouts, no net password encryption 2
```

To shut down the remote Backup Server named WHALE_BACKUP, use the following command:

```
shutdown REM_BACKUP
```

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

shutdown permission defaults to System Administrators and is not transferable.

See Also

Commands	**alter database**
System procedures	**sp_addserver, sp_helpserver**

truncate table

Function

Removes all rows from a table.

Syntax

```
truncate table [[database.]owner.]table_name
```

Keywords and Options

table_name – is the name of the table to truncate. Specify the database name if the table is in another database, and specify the owner's name if more than one table of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

Examples

1. `truncate table authors`

   Removes all data from the *authors* table.

Comments

- **truncate table** deletes all rows from a table. The table structure and all the indexes continue to exist until you issue a **drop table** command. The rules, defaults and constraints that are bound to the columns remain bound, and triggers remain in effect.
- **truncate table** deallocates the distribution pages for all indexes; remember to run **update statistics** after adding new rows to the table.
- **truncate table** is equivalent to—but faster than—a **delete** command without a **where** clause. **delete** removes rows one at a time and logs each deleted row as a transaction; **truncate table** deallocates whole data pages and makes fewer log entries. Both **delete** and **truncate table** reclaim the space occupied by the data and its associated indexes.
- Because the deleted rows are not logged individually, **truncate table** cannot fire a trigger.
- You cannot use **truncate table** if a another table has rows that reference it. Delete the rows from the foreign table, or truncate the foreign table, and then truncate the primary table.

- You cannot use the **truncate table** command on a partitioned table. Unpartition the table with the **unpartition** clause of the **alter table** command before issuing the **truncate table** command.

  You can use the **delete** command without a **where** clause to remove all rows from a partitioned table without first unpartitioning it. This method is generally slower than **truncate table**, since it deletes one row at a time and logs each **delete** operation.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

truncate table permission defaults to the table owner and is not transferable. To truncate a system audit table (*sysaudits_01*, *sysaudits_02*, *sysaudits_03*, and so on, through *sysaudits_08*), you must be a System Security Officer.

See Also

Commands	**create trigger, delete, drop table**

union Operator

Function

Returns a single result set that combines the results of two or more queries. Duplicate rows are eliminated from the result set unless the **all** keyword is specified.

Syntax

```
select select_list [into clause]
          [from clause] [where clause]
          [group by clause] [having clause]
   [union [all]
       select select_list
          [from clause] [where clause]
          [group by clause] [having clause] ]...
   [order by clause]
   [compute clause]
```

Keywords and Options

union – creates the union of data specified by two **select** statements.

all – includes all rows in the results; duplicates are not removed.

into – creates a new table based on the columns specified in the select list and the rows chosen in the **where** clause. The first query in the union operation is the only one that can contain an **into** clause.

Examples

1.
```
select stor_id, stor_name from sales
union
select stor_id, stor_name from sales_east
```

The result set includes the contents of the *stor_id* and *stor_name* columns of both the *sales* and *sales_east* tables.

2.
```
select pub_id, pub_name, city into results
from publishers
union
select stor_id, stor_name, city from stores
union
select stor_id, stor_name, city from stores_east
```

The **into** clause in the first query specifies that the table *results* holds the final result set of the union of the specified columns of the *publishers*, *stores*, and *stores_east* tables.

3.
```
select au_lname, city, state from authors
union
((select stor_name, city, state from sales
union
select stor_name, city, state from sales_east)
union
select pub_name, city, state from publishers)
```

   First, the **union** of the specified columns in the *sales* and *sales_east* tables is generated. Then, the **union** of that result with *publishers* is generated. Finally, the **union** of the second result and *authors* is generated.

Comments

- The total number of tables that can appear on all sides of a **union** query is 256.
- The **order by** and **compute** clauses are allowed only at the end of the **union** statement to define the order of the final results or to compute summary values.
- The **group by** and **having** clauses can be used only within individual queries and cannot be used to affect the final result set.
- The default evaluation order of a SQL statement containing **union** operators is left-to-right.
- Since **union** is a binary operation, parentheses must be added to an expression involving more than two queries to specify evaluation order.
- The first query in a **union** statement may contain an **into** clause that creates a table to hold the final result set. The **into** statement must be in the first query, or you will receive an error message (see example 2).
- The **union** operator can appear within an **insert...select** statement. For example:

```
insert into sales.all
  select * from sales
  union
  select * from sales_east
```

- All select lists in a SQL statement must have the same number of expressions (column names, arithmetic expressions, aggregate functions, and so on). For example, this statement is invalid because the first select list contains more expressions than the second:

```
select au_id, title_id, au_ord from titleauthor
union
select stor_id, date from sales
```

- Corresponding columns in the select lists of **union** statements must occur in the same order, because **union** compares the columns one-to-one in the order given in the individual queries.
- The column names in the table resulting from a **union** are taken from the **first** individual query in the **union** statement. If you want to define a new column heading for the result set, you must do it in the first query. Also, if you want to refer to a column in the result set by a new name (for example, in an **order by** statement), you must refer to it by that name in the first **select** statement. For example, the following query is correct:

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

- The descriptions of the columns that are part of a **union** operation do not have to be identical. Table 1-25 lists the rules for the datatypes and the corresponding column in the result table.

Table 1-25: Resulting datatypes in union operations

Datatype of Columns in *union* Operation	Datatype of Corresponding Column in Result Table
Not datatype-compatible (data conversion is not handled implicitly by Adaptive Server)	Error returned by Adaptive Server.
Both are fixed-length character with lengths L1 and L2	Fixed-length character with length equal to the greater of L1 and L2.
Both are fixed-length binary with lengths L1 and L2	Fixed-length binary with length equal to the greater of L1 and L2.
Either or both are variable-length character	Variable-length character with length equal to the maximum of the lengths specified for the column in the union.
Either or both are variable-length binary	Variable-length binary with length equal to the maximum of the lengths specified for the columns in the union.

Table 1-25: Resulting datatypes in union operations (continued)

Datatype of Columns in *union* Operation	**Datatype of Corresponding Column in Result Table**
Both are numeric datatypes (for example, *smallint*, *int*, *float*, *money*)	A datatype equal to the maximum precision of the two columns. For example, if a column in table A is of type *int* and the corresponding column in table B is of type *float*, then the datatype of the corresponding column of the result table is *float*, because *float* is more precise than *int*.
Both column descriptions specify NOT NULL	Specifies NOT NULL.

Restrictions

- You cannot use **union** in the **select** statement of an updatable cursor.
- You cannot use the **union** operator in a **create view** statement.
- You cannot use the **union** operator in a subquery.
- You cannot use the **union** operator with the **for browse** clause.
- You cannot use the **union** operator on queries that select *text* or *image* data.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The following are Transact-SQL extensions: • The use of **union** in the select clause of an **insert** statement • Specifying new column headings in the **order by** clause of a **select** statement when the **union** operator is present in the **select** statement

See Also

Commands	**compute Clause, declare, group by and having Clauses, order by Clause, select, where Clause**
Functions	**convert**

update

Function

Changes data in existing rows, either by adding data or by modifying existing data.

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|(select_statement)}
       [, column_name2 =
           {expression2|NULL|(select_statement)}]... |
       [, variable_name2 =
           {expression2|NULL|(select_statement)}]...
   [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]
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 =
           {expression2|NULL|(select_statement)}]...
   where current of cursor_name
```

Keywords and Options

table_name | *view_name* – is the name of the table or view to update. Specify the database name if the table or view is in another database, and specify the owner’s name if more than one table or view of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

set – specifies the column name or variable name and assigns the new value. The value can be an expression or a NULL. When more than one column name or variable name and value are listed, they must be separated by commas.

from – uses data from other tables or views to modify rows in the table or view you are updating.

where – is a standard **where** clause (see "where Clause").

index *index_name* – specifies the index to be used to access *table_name*. You cannot use this option when you update a view.

prefetch *size* – specifies the I/O size, in kilobytes, for tables bound to caches with large I/Os configured. Values for *size* are **2**, **4**, **8**, and **16**. You cannot use this option when you update a view. The procedure **sp_helpcache** shows the valid sizes for the cache to which an object is bound or for the default cache.

If Component Integration Services is enabled, you cannot use **prefetch** for remote servers.

lru | mru – specifies the buffer replacement strategy to use for the table. Use **lru** to force the optimizer to read the table into the cache on the MRU/LRU (most recently used/least recently used) chain. Use **mru** to discard the buffer from cache and replace it with the next buffer for the table. You cannot use this option when you update a view.

where current of – causes Adaptive Server to update the row of the table or view indicated by the current cursor position for *cursor_name*.

Examples

1.
```
update authors
set au_lname = "MacBadden"
where au_lname = "McBadden"
```

   All the McBaddens in the *authors* table are now MacBaddens.

2.
```
update titles
set total_sales = total_sales + qty
from titles, salesdetail, sales
where titles.title_id = salesdetail.title_id
    and salesdetail.stor_id = sales.stor_id
    and salesdetail.ord_num = sales.ord_num
    and sales.date in
       (select max(sales.date) from sales)
```

   Modifies the *total_sales* column to reflect the most recent sales recorded in the *sales* and *salesdetail* tables. This assumes that only one set of sales is recorded for a given title on a given date, and that updates are current.

3.
```
update titles
set price = 24.95
where current of title_crsr
```

   Changes the price of the book in the *titles* table that is currently pointed to by *title_crsr* to $24.95.

4.
```
update titles
set price = 18.95
where syb_identity = 4
```

   Finds the row for which the IDENTITY column equals 4 and changes the price of the book to $18.95. Adaptive Server replaces the **syb_identity** keyword with the name of the IDENTITY column.

5.
```
declare @x money
select @x = 0
update titles
    set total_sales = totals_sales + 1,
    @x = price
    where title_id = "BU1032"
```

   Updates the *titles* table using a declared variable.

Comments

- Use **update** to change values in rows that have already been inserted. Use **insert** to add new rows.
- You can refer to up to 15 tables in an **update** statement.
- **update** interacts with the **ignore_dup_key**, **ignore_dup_row**, and **allow_dup_row** options set with the **create index** command. (See **create index** for more information.)
- You can define a trigger that takes a specified action when an **update** command is issued on a specified table or on a specified column in a table.

Using Variables in *update* statements

- You can assign variables in the **set** clause of an **update** statement, similarly to setting them in a **select** statement.
- Before you use a variable in an **update** statement, you must declare the variable using **declare**, and initialize it with **select**, as shown in example 5.
- Variable assignment occurs for every qualified row in the update.
- When a variable is referenced on the right side of an assignment in an **update** statement, the current value of the variable changes as each row is updated. The **current value** is the value of the variable just before the update of the current row. The following example shows how the current value changes as each row is updated.

  Suppose you have the following statement:

  ```
  declare @x int
  select @x=0
  update table1
      set C1=C1+@x, @x=@x+1
      where column2=xyz
  ```

  The value of C1 before the update begins is 1. The following table shows how the current value of the @x variable changes after each update:

Row	Initial C1 Value	Initial @x Value	Calculations: C1+@x= updated C1	Updated C1 Value	Calculations: @x+1= updated @x	Updated @x Value
A	1	0	1+0	1	0+1	1
B	1	1	1+1	2	1+1	2
C	2	2	2+2	4	2+1	3
D	4	3	4+3	7	3+1	4

- When multiple variable assignments are given in the same **update** statement, the values assigned to the variables can depend on their order in the assignment list, but they might not always do so. For best results, do not rely on placement to determine the assigned values.
- If multiple rows are returned and a non-aggregating assignment of a column to a variable occurs, then the final value of the

variable will be the last row process; therefore, it might not be useful.

- An **update** statement that assigns values to variables need not set the value of any qualified row.
- If no rows qualify for the update, the variable is not assigned.
- A variable that is assigned a value in the **update** statement cannot be referenced in subquery in that same **update** statement, regardless of where the subquery appears in that **update** statement.
- A variable that is assigned a value in the **update** statement cannot be referenced in a **where** or **having** clause in that same **update** statement.
- In an update driven by a join, a variable that is assigned a value in the right hand side of the **update** statement uses columns from the table that is not being updated. The result value depends on the join order chosen for the update and the number of rows that qualify from the joined table.
- Updating a variable is not affected by a rollback of the **update** statement because the value of the updated variable is not stored on disk.

Using *update* with Transactions

- When you set **chained transaction mode on**, and no transaction is currently active, Adaptive Server implicitly begins a transaction with the **update** statement. To complete the update, you must either **commit** the transaction or **rollback** the changes. For example:

```
update stores set city = 'Concord'
    where stor_id = '7066'
if exists (select t1.city, t2.city
    from stores t1, stores t2
    where t1.city = t2.city
    and t1.state = t2.state
    and t1.stor_id < t2.stor_id)
        rollback transaction
else
        commit transaction
```

This batch begins a transaction (using chained transaction mode) and updates a row in the *stores* table. If it updates a row containing the same city and state information as another store in the table, it rolls back the changes to the *stores* table and ends the transaction. Otherwise, it commits the updates and ends the transaction.

- Adaptive Server does not prevent you from issuing an **update** statement that updates a single row more than once in a given transaction. However, because of the way the **update** statement 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.

Using *update* with Character Data

- Updating variable-length character data or *text* columns with the empty string (“”) inserts a single space. Fixed-length character columns are padded to the defined length.
- All trailing spaces are removed from variable-length column data, except in the case of a string containing only spaces. Strings that contain only spaces are truncated to a single space. Strings longer than the specified length of a *char*, *nchar*, *varchar*, or *nvarchar* column are silently truncated unless you set **string_rtruncation on**.
- An **update** to a *text* column initializes the *text* column, assigns it a valid text pointer, and allocates at least one 2K data page.

Using *update* with Cursors

- To update a row using a cursor, first define the cursor with **declare cursor**, and then open it. The cursor name cannot be a Transact-SQL parameter or a local variable. The cursor must be updatable, or Adaptive Server returns an error. Any update to the cursor result set also affects the base table row from which the cursor row is derived.
- The *table_name* or *view_name* specified with an **update...where current of** 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 being updated.

  After the update, the cursor position remains unchanged. You can continue to update the row at that cursor position, provided 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 that are part of the tables specified in the *select_statement*. However, when you specify a *column_name_list* with **for update,** and you are declaring the cursor, you can update only those specific columns.

Updating IDENTITY Columns

- A column with the IDENTITY property cannot be updated, either through its base table or through a view. To determine whether a column was defined with the IDENTITY property, use the **sp_help** system procedure on the column's base table.
- An IDENTITY column selected into a result table observes the following rules with regard to inheritance of the IDENTITY property:
  - 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.
  - If an IDENTITY column is selected as part of an expression, the resulting column does not inherit the IDENTITY property. It is created as NULL if any column in the expression allows nulls; otherwise, it is NOT NULL.
  - If the select statement contains a group by clause or aggregate function, the resulting column does not inherit the IDENTITY property. Columns that include an aggregate of the IDENTITY column are created NULL; others are created NOT NULL.
  - 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 defined as NOT NULL.

Updating Data Through Views

- You cannot **update** views defined with the **distinct** clause.
- If a view is created **with check option**, each row that is updated through the view must remain visible through the view. For example, the *stores_cal* view includes all rows of the *stores* table where *state* has a value of "CA". The **with check option** clause checks each **update** statement against the view's selection criteria:

```
create view stores_cal
as select * from stores
where state = "CA"
with check option
```

An **update** statement such as this one fails if it changes *state* to a value other than “CA”:

```
update stores_cal
set state = "WA"
where store_id = "7066"
```

- If a view is created **with check option**, all views derived from the base view must satisfy the view’s selection criteria. Each row updated through a **derived** view must remain visible through the base view.

  Consider the view *stores_cal30*, which is derived from *stores_cal*. The new view includes information about stores in California with payment terms of “Net 30”:

  ```
  create view stores_cal30
  as select * from stores_cal
  where payterms = "Net 30"
  ```

  Because *stores_cal* was created **with check option**, all rows updated through *stores_cal30* must remain visible through *stores_cal*. Any row that changes *state* to a value other than “CA” is rejected.

  Notice that *stores_cal30* does not have a **with check option** clause of its own. Therefore, you can update a row with a *payterms* value other than “Net 30” through *stores_cal30*. For example, 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"
  ```

- You cannot update a row through a view that joins columns from two or more tables, unless both of the following conditions are true:
  - The view has no **with check option** clause, and
  - All columns being updated belong to the same base table.
- **update** statements are allowed on join views that contain a **with check option** clause. The update fails if any of the affected columns appear in the **where** clause in an expression that includes columns from more than one table.

- If you update a row through a join view, all affected columns must belong to the same base table.

Using *index*, *prefetch*, or *lru | mru*

- **index**, **prefetch**, and **lru | mru** override the choices made by the Adaptive Server optimizer. Use them with caution, and always check the performance impact with **set statistics io on**. See the *Performance and Tuning Guide* for more information about using these options.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	The use of a **from** clause or a qualified table or column name are Transact-SQL extensions detected by the FIPS flagger. Updates through a join view or a view of which the target list contains an expression are Transact-SQL extensions that cannot be detected until run time and are not flagged by the FIPS flagger. The use of variables is a Transact-SQL extension.

Permissions

update permission defaults to the table or view owner, who can transfer it to other users.

If you have **set ansi_permissions on**, you will need, in addition to the regular permissions required for **update** statements, **select** permission on all columns appearing in the **where** clause, and on all columns following the **set** clause.

See Also

Commands	**create default, create index, create rule, create trigger, insert, where Clause**
System procedures	**sp_bindefault**, **sp_bindrule**, **sp_help**, **sp_unbindefault**, **sp_unbindrule**

update all statistics

Function

Updates all statistics information for a given table.

Syntax

```
update all statistics table_name
```

Keywords and Options

table_name – is the name of the table for which statistics are being updated.

Examples

1. **`update all statistics salesdetail`**

   Updates index and partition statistics for the *salesdetail* table.

Comments

- The **update all statistics** command updates all statistics information for a given table. Adaptive Server keeps statistics about the distribution of pages within a table, and uses these statistics when considering whether or not to use a parallel scan in query processing on partitioned tables, and which index(es) to use in query processing. The optimization of your queries depends on the accuracy of the stored statistics.
- If the table is not partitioned, **update all statistics** runs **update statistics** on the table.
- If the table is partitioned and has no indexes, **update all statistics** runs **update partition statistics** on the table. If the table is partitioned and has indexes, **update all statistics** runs **update statistics** and **update partition statistics** on the table.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

update all statistics permission defaults to the table owner and is not transferrable.

See Also

Commands	**update statistics**, **update partition statistics**

update partition statistics

Function

Updates information about the number of pages in each partition for a partitioned table.

Syntax

```
update partition statistics table_name
   [partition_number]
```

Keywords and Options

table_name – is the name of a partitioned table.

partition_number – is the number of the partition for which you are updating information. If you do not specify a partition number, **update partition statistics** updates the number of data pages in all partitions in the specified table.

Comments

- Adaptive Server keeps statistics about the distribution of pages within a partitioned table and uses these statistics when considering whether to use a parallel scan in query processing. The optimization of your queries depends on the accuracy of the stored statistics. If Adaptive Server crashes, the distribution information could be inaccurate.

  To see if the distribution information is accurate, use the **data_pgs** function to determine the number of pages in the table, as follows:

  ```
  select data_pgs(sysindexes.id, doampg)
      from sysindexes
      where sysindexes.id = object_id("table_name")
  ```

  Then, use the system procedure **sp_helpartition** on the table and add up the numbers in the "ptn_data_pgs" column of the output. The sum of the total of the number of pages that **sp_helpartition** reports should be slightly greater than the number returned by **data_pgs** because **sp_helpartition**'s page count includes OAM pages.

  If the distribution information is inaccurate, run **update partition statistics** on the table. While updating the distribution information, **update partition statistics** locks the OAM page and the control page of the partition.

- When you run **update partition statistics** on a table that contains data, or you create an index on a table that contains data, the *controlpage* column in *syspartitions* is updated to point to the control page for the partition.
- **update partition statistics** updates control page values used to estimate the number of pages in a table. These statistics are used by **sp_helpartition.**

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

update partition statistics permission defaults to the table owner and is not transferable.

See Also

Commands	**alter table**, **update all statistics**
Functions	**ptn_data_pgs**
System procedures	**sp_helpartition**

update statistics

Function

Updates information about the distribution of key values in specified indexes.

Syntax

```
update statistics table_name [index_name]
```

Keywords and Options

table_name – is the name of the table with which the index is associated. *table_name* is required, since Transact-SQL does not require index names to be unique in a database.

index_name – is the name of the index to be updated. If an index name is not specified, the distribution statistics for all the indexes in the specified table are updated.

Examples

1. **`update statistics titles`**

   Updates the distribution statistics for any indexes on the *titles* table.

2. **`update statistics salesdetail titleidind`**

   Updates the distribution statistics for the index *titleidind* on the *salesdetail* table.

Comments

- Adaptive Server keeps statistics about the distribution of the key values in each index, and uses these statistics in its decisions about which index(es) to use in query processing.
- When you create a nonclustered index on a table that contains data, **update statistics** is automatically run for the new index. When you create a clustered index on a table that contains data, **update statistics** is automatically run for all indexes.
- The optimization of your queries depends on the accuracy of the distribution steps. If there is significant change in the key values in your index, you should rerun **update statistics** on that index. Use the **update statistics** command if a great deal of data in an indexed column has been added, changed, or removed (that is, if you suspect that the distribution of key values has changed).

- Run **update statistics** after adding new rows to a table whose rows had previously been deleted with **truncate table**.
- When you run **update statistics** on an index that contains data, or create an index on a table that contains data, the *distribution* column in *sysindexes* is updated to point to the distribution page for the index.
- **update statistics** updates allocation page values used to estimate the number of rows in a table. These statistics are used by the **rowcnt** function and **sp_spaceused**.

Standards and Compliance

Standard	**Compliance Level**
SQL92	Transact-SQL extension

Permissions

update statistics permission defaults to the table owner and is not transferable. The command can also be executed by the Database Owner, who can impersonate the table owner by running the **setuser** command.

See Also

Commands	**create index**
System procedures	**sp_helpindex**

use

Function

Specifies the database with which you want to work.

Syntax

```
use database_name
```

Keywords and Options

database_name – is the name of the database to open.

Examples

1.
   ```
   use pubs2
   go
   ```

   The current database is now *pubs2*.

Comments

- The **use** command must be executed before you can reference objects in a database.
- **use** cannot be included in a stored procedure or a trigger.
- An alias permits a user to use a database under another name in order to gain access to that database. Use the system procedure **sp_addalias**.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

If the database has a "guest" account, all users can use the database. If the database does not have a "guest" account, you must be a valid user in the database, have an alias in the database, or be a System Administrator or System Security Officer.

See Also

Commands	create database, drop database
System procedures	**sp_addalias**, **sp_adduser**, **sp_modifylogin**

waitfor

Function

Specifies a specific time, a time interval, or an event for the execution of a statement block, stored procedure, or transaction.

Syntax

```
waitfor { delay time | time time | errorexit
   | processexit | mirrorexit }
```

Keywords and Options

delay – instructs Adaptive Server to wait until the specified amount of time has passed, up to a maximum of 24 hours.

time – instructs Adaptive Server to wait until the specified time.

time – a time in one of the acceptable formats for *datetime* data, or a variable of character type. You cannot specify dates—the date portion of the *datetime* value is not allowed.

errorexit – instructs Adaptive Server to wait until a kernel or user process terminates abnormally.

processexit – instructs Adaptive Server to wait until a kernel or user process terminates for any reason.

mirrorexit – instructs Adaptive Server to wait for a mirror failure.

Examples

1.
```
begin
     waitfor time "14:20"
     insert chess(next_move)
        values('Q-KR5')
     execute sendmail 'judy'
end
```

At 2:20 p.m., the *chess* table will be updated with my next move, and a procedure called *sendmail* will insert a row in a table owned by Judy, notifying her that a new move now exists in the *chess* table.

2.
```
declare @var char(8)
select @var = "00:00:10"
begin
    waitfor delay @var
     print "Ten seconds have passed.  Your time
       is up."
end
```

   After 10 seconds, Adaptive Server prints the message specified.

3.
```
begin
   waitfor errorexit
   print "Process exited abnormally!"
end
```

   After any process exits abnormally, Adaptive Server prints the message specified.

Comments

- After issuing the **waitfor** command, you cannot use your connection to Adaptive Server until the time or event that you specified occurs.
- 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 terminated, check the *sysprocesses* table with the system procedure **sp_who**.
- The time you specify with **waitfor time** or **waitfor delay** can include hours, minutes, and seconds. Use the format "hh:mi:ss", as described in "Date / time Datatypes" in Chapter 7, "System and User-Defined Datatypes."

  For example:

  ```
  waitfor time "16:23"
  ```

  instructs Adaptive Server to wait until 4:23 p.m. The statement:

  ```
  waitfor delay "01:30"
  ```

  instructs Adaptive Server to wait for 1 hour and 30 minutes.
- Changes in system time (such as setting the clock back for Daylight Savings Time) can delay the **waitfor** command.
- You can use **waitfor mirrorexit** within a DB-Library program to notify users when there is a mirror failure.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

waitfor permission defaults to all users. No permission is required to use it.

See Also

Commands	**begin...end**
Datatypes	"Date / time Datatypes"
System procedures	**sp_who**

where Clause

Function

Sets the search conditions in a **select**, **insert**, **update**, or **delete** statement.

Syntax

Search conditions immediately follow the keyword **where** in a **select, insert, update,** or **delete** statement. If you use more than one search condition in a single statement, connect the conditions with **and** or **or**.

```
where [not] expression comparison_operator expression

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

where [not] expression is [not] null

where [not]
   expression [not] between expression and expression

where [not]
   expression [not] in ({value_list | subquery})

where [not] exists (subquery)

where [not]
   expression comparison_operator
   {any | all} (subquery)

where [not] column_name join_operator column_name

where [not] logical_expression

where [not] expression {and | or} [not] expression
```

Keywords and Options

not – negates any logical expression or keywords such as **like, null, between**, **in**, and **exists**.

expression – is a column name, a constant, a function, a subquery, or any combination of column names, constants, and functions connected by arithmetic or bitwise operators. For more information about expressions, see “Expressions” in Appendix A, “Expressions, Identifiers, and Wildcard Characters.”

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

In comparing *char, nchar, varchar,* and *nvarchar* data, **<** means closer to the beginning of the alphabet and **>** means closer to the end of the alphabet.

Case and special character evaluations depend on the collating sequence of the operating system on the machine on which Adaptive Server is located. For example, lowercase letters may be greater than uppercase letters, and uppercase letters may be greater than numbers.

Trailing blanks are ignored for the purposes of comparison. For example, "Dirk" is the same as "Dirk ".

In comparing dates, **<** means earlier and **>** means later. Put quotes around all character and date data used with a comparison operator. For example:

```
= "Bennet"
> "94609"
```

See "System and User-Defined Datatypes" in Chapter 7, "System and User-Defined Datatypes," for more information about data entry rules.

like – is a keyword indicating that the following character string (enclosed by single or double quotes) is a matching pattern. **like** is available for *char*, *varchar*, *nchar*, *nvarchar*, and *datetime* columns, but not to search for seconds or milliseconds.

You can use the keyword **like** and wildcard characters with *datetime* data as well as with *char* and *varchar*. When you use **like** with *datetime* values, Adaptive Server converts the dates to 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" into a column named *arrival_time*, the clause:

```
where arrival_time = '9:20'
```

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

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

would find it.

match_string – is a string of characters and wildcard characters enclosed in quotes. Table 1-26 lists the wildcard characters.

Table 1-26: Wildcard characters

Wildcard Character	Meaning
%	Any string of 0 or more characters
_	Any single character
[ ]	Any single character within the specified range ([a-f]) or set ([abcdef])
[^]	Any single character that is not within the specified range ([^a-f]) or set ([^abcdef])

escape – specifies an escape character with which you can search for literal occurrences of wildcard characters.

escape_character – is any single character. See "Using the escape Clause" in Appendix A, "Expressions, Identifiers, and Wildcard Characters," for more information.

is null – searches for null values.

between – is the range-start keyword. Use **and** for the range-end value. The range:

```
where @val between x and y
```

is inclusive; the range:

```
x and @val < y
```

is not. Queries using **between** return no rows if the first value specified is greater than the second value.

and – joins two conditions and returns results when both of the conditions are true.

When more than one logical operator is used in a statement, **and** operators are usually evaluated first. However, you can change the order of execution with parentheses.

in – allows you to select values that match any one of a list of values. The comparator can be a constant or a column name, and the list can be a set of constants or, more commonly, a subquery. (See Chapter 5, "Subqueries: Using Queries Within Other Queries," in the *Transact-SQL User's Guide* for information on using **in** with a subquery.) Enclose the list of values in parentheses.

value_list – is a list of values. Put single or double quotes around character values, and separate each value from the following one with a comma (see example 7). The list can be a list of variables, for example:

```
in (@a, @b, @c)
```

However, you cannot use a variable containing a list, such as:

```
@a = "'1', '2', '3'"
```

for a values list.

exists – is used with a subquery to test for the existence of some result from the subquery. (See Chapter 5, "Subqueries: Using Queries Within Other Queries," in the *Transact-SQL User's Guide* for more information.)

subquery – is a restricted **select** statement (**order by** and **compute** clauses and the keyword **into** are not allowed) inside the **where** or **having** clause of a **select**, **insert**, **delete**, or **update** statement, or a subquery. (See Chapter 5, "Subqueries: Using Queries Within Other Queries," in the *Transact-SQL User's Guide* for more information.)

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. (See Chapter 5, "Subqueries: Using Queries Within Other Queries," in the *Transact-SQL User's Guide* for more information.)

all – is used with **>** or **<** and 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. (See Chapter 5, "Subqueries: Using Queries Within Other Queries," in the *Transact-SQL User's Guide* for more information.)

column_name – is the name of the column used in the comparison. Qualify the column name with its table or view name if there is any ambiguity. For columns with the IDENTITY property, you can specify the **syb_identity** keyword, qualified by a table name where necessary, rather than the actual column name.

join_operator – is a comparison operator or one of the join operators **=*** or ***=**. (See Chapter 4, "Joins: Retrieving Data from Several Tables," in the *Transact-SQL User's Guide* for more information.)

logical_expression – is an expression that returns TRUE or FALSE.

or – joins two conditions and returns results when either of the conditions is true.

When more than one logical operator is used in a statement, **or** operators are normally evaluated after **and** operators. However, you can change the order of execution with parentheses.

Examples

1.
```
where advance * $2 > total_sales * price
```
2.
```
where phone not like '415%'
```
   Finds all the rows in which the phone number does not begin with 415.
3.
```
where au_lname like "[CK]ars[eo]n"
```
   Finds the rows for authors named Carson, Carsen, Karsen, and Karson.
4.
```
where sales_east.syb_identity = 4
```
   Finds the row of the *sales_east* table in which the IDENTITY column has a value of 4.
5.
```
where advance < $5000 or advance is null
```
6.
```
where (type = "business" or type = "psychology")
 and advance > $5500
```
7.
```
where total_sales between 4095 and 12000
```
8.
```
where state in ('CA', 'IN', 'MD')
```
   Finds the rows in which the state is one of the three in the list.

Comments

- **where** and **having** search conditions are identical, except that aggregate functions are not permitted in **where** clauses. For example, this clause is legal:

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

  This clause is not legal:

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

  See Chapter 2, "Aggregate Functions," for information on the use of aggregate functions, and "group by and having Clauses" for examples.
- Joins and subqueries are specified in the search conditions: see the Chapter 4, "Joins: Retrieving Data from Several Tables," and Chapter 5, "Subqueries: Using Queries Within Other Queries," in the *Transact-SQL User's Guide* for full details.
- You include up to 252 **and** and **or** conditions in a **where** clause.
- There are two ways to specify literal quotes within a *char* or *varchar* entry. The first method is to use two quotes. For example, if you began 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.'
  ```

  Or use 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 double quotes 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?"’
  ```
- To enter a character string that is longer than the width of your screen, enter a backslash (\) before going to the next line.
- If a column is compared to a constant or variable in a **where** clause, Adaptive Server converts the constant or variable into the datatype of the column so that the optimizer can use the index for

data retrieval. For example, *float* expressions are converted to *int* when compared to an *int* column. For example:

```
where int_column = 2
```

selects rows where *int_column* = 2.

- When Adaptive Server optimizes queries, it evaluates the search conditions in **where** and **having** clauses, and determines which conditions are search arguments (SARGs) that can be used to choose the best indexes and query plan. For each table in a query, a maximum of 128 search arguments can be used to optimize the query. All of the search conditions, however, are used to qualify the rows. For more information on search arguments, see Chapter 8, "Search Arguments and Using Indexes," in the *Performance and Tuning Guide*.

Standards and Compliance

Standard	Compliance Level
SQL92	Entry level compliant

See Also

Commands	**delete**, **execute**, **group by and having Clauses**, **insert**, **select**, **update**
Datatypes	"System and User-Defined Datatypes"
System procedures	**sp_helpjoins**

while

Function

Sets a condition for the repeated execution of a statement or statement block. The statement(s) are executed repeatedly, as long as the specified condition is true.

Syntax

```
while logical_expression
        statement
```

Keywords and Options

logical_expression – is any expression that returns TRUE, FALSE, or NULL.

statement – can be a single SQL statement, but is usually a block of SQL statements delimited by **begin** and **end.**

Examples

1.
```
while (select avg(price) from titles) < $30
begin
    select title_id, price
        from titles
        where price > $20
    update titles
        set price = price * 2
end
```

If the average price is less than $30, double the prices of all books in the *titles* table. As long as it is still less than $30, the **while** loop keeps doubling the prices. In addition to determining the titles whose price exceeds $20, the **select** inside the **while** loop indicates how many loops were completed (each average result returned by Adaptive Server indicates one loop).

Comments

- The execution of statements in the **while** loop can be controlled from inside the loop with the **break** and **continue** commands.
- The **continue** command causes the **while** loop to restart, skipping any statements after the **continue**. The **break** command causes an exit from the **while** loop. Any statements that appear after the

keyword **end**, which marks the end of the loop, are executed. The **break** and **continue** commands are often activated by **if** tests.

For example:

```
while (select avg(price) from titles) < $30
begin
    update titles
        set price = price * 2
    if (select max(price) from titles) > $50
        break
    else
        if (select avg(price) from titles) > $30
            continue
    print "Average price still under $30"
end

select title_id, price from titles
    where price > $30
```

This batch continues to double the prices of all books in the *titles* table as long as the average book price is less than $30. However, if any book price exceeds $50, the **break** command stops the **while** loop. The **continue** command prevents the **print** statement from executing if the average exceeds $30. Regardless of how the **while** loop terminates (either normally or because of the **break** command), the last query indicates which books are priced over $30.

- If two or more **while** loops are nested, the **break** command exits to the next outermost loop. First, all the statements after the end of the inner loop run, and then the next outermost loop restarts.

◆ ***WARNING!***

If a create table or create view command occurs within a while loop, Adaptive Server creates the schema for the table or view before determining whether the condition is true. This may lead to errors if the table or view already exists.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

while permission defaults to all users. No permission is required to use it.

See Also

Commands	**begin...end, break, continue, goto Label**

writetext

Function

Permits minimally logged, interactive updating of an existing *text* or *image* column.

Syntax

```
writetext [[database.]owner.]table_name.column_name
   text_pointer [with log] data
```

Keywords and Options

table_name.column_name – is the name of the table and *text* or *image* column to update. Specify the database name if the table is in another database, and specify the owner's name if more than one table of that name exists in the database. The default value for *owner* is the current user, and the default value for *database* is the current database.

text_pointer – a *varbinary(16)* value that stores the pointer to the *text* or *image* data. Use the **textptr** function to determine this value, as shown in example 1. *text* and *image* data is not stored in the same set of linked pages as other table columns. It is stored in a separate set of linked pages. A pointer to the actual location is stored with the data; **textptr** returns this pointer.

with log – logs the inserted *text* or *image* data. The use of this option aids media recovery, but logging large blocks of data quickly increases the size of the transaction log, so make sure that the transaction log resides on a separate database device (see **create database**, **sp_logdevice**, and the *System Administration Guide* for details).

data – is the data to write into the *text* or *image* column. *text* data must be enclosed in quotes. *image* data must be preceded by "0x". Check the information about the client software you are using to determine the maximum length of *text* or *image* data that can be accomodated by the client.

Examples

1.
   ```
   declare @val varbinary(16)
   select @val = textptr(copy) from blurbs
       where au_id = "409-56-7008"
   writetext blurbs.copy @val with log "hello world"
   ```

   This example puts the text pointer into the local variable *@val*. Then, **writetext** places the text string "hello world" into the text field pointed to by *@val*.

Comments

- The maximum length of text that can be inserted interactively with **writetext** is approximately 120K bytes for *text* and *image* data.
- By default, **writetext** is a minimally logged operation; only page allocations and deallocations are logged, but the *text* or *image* data is not logged when it is written into the database. In order to use **writetext** in its default, minimally logged state, a System Administrator must use **sp_dboption** to set **select into/bulkcopy/pllsort** to **true**.
- **writetext** updates *text* data in an existing row. The update completely replaces all of the existing text.
- **writetext** operations are not caught by an **insert** or **update** trigger.
- **writetext** requires a valid text pointer to the *text* or *image* column. In order for a valid text pointer to exist, a *text* column must contain either actual data or a null value that has been explicitly entered with **update**.

  Given the table *textnull* with columns *key* and *x*, where *x* is a *text* column that permits nulls, this **update** sets all the *text* values to NULL and assigns a valid text pointer in the *text* column:

  ```
  update textnull
  set x = null
  ```

  No text pointer results from an **insert** of an explicit null:

  ```
  insert textnull values (2,null)
  ```

  or this implicit null insert:

  ```
  insert textnull (key)
  values (2)
  ```

- **insert** and **update** on *text* columns are logged operations.
- You cannot use **writetext** on *text* and *image* columns in views.

- If you attempt to use **writetext** on *text* values after changing to a multibyte character set, and you have not run **dbcc fix_text**, the command fails, and an error message is generated, instructing you to run **dbcc fix_text** on the table.
- **writetext** in its default, non-logged mode runs more slowly while a **dump database** is taking place.
- The Client-Library functions **dbwritetext** and **dbmoretext** are faster and use less dynamic memory than **writetext**. These functions can insert up to 2GB of *text* data.

Standards and Compliance

Standard	**Compliance Level**
SQL92	Transact-SQL extension

Permissions

writetext permission defaults to the table owner, who can transfer it to other users.

See Also

Commands	**readtext**
Datatypes	"text and image Datatypes"

Functions

2 Transact-SQL Functions

This chapter describesthe Transact-SQL functions. Functions are used to return information from the database. They are allowed in the **select** list, in the **where** clause, and anywhere else an expression is allowed. They are often used as part of a stored procedure or program.

Types of Functions

Table 2-1 lists the different types of Transact-SQL functions and describes the type of information each returns:

Table 2-1: Types of Transact-SQL functions

Type of Function	Description
Aggregate Functions	Generate summary values that appear as new columns or as additional rows in the query results.
Datatype Conversion Functions	Change expressions from one datatype to another and specify new display formats for date/time information.
Date Functions	Do computations on *datetime* and *smalldatetime* values and their components, date parts.
Mathematical Functions	Return values commonly needed for operations on mathematical data.
Security Functions	Return security-related information.
String Functions	Operate on binary data, character strings, and expressions.
System Functions	Return special information from the database.
Text and Image Functions	Supply values commonly needed for operations on *text* and *image* data.

Table 2-2 lists the functions in alphabetical order.

Table 2-2: List of Transact-SQL functions

Function	Type	Return Value
abs	Mathematical	The absolute value of an expression.
acos	Mathematical	The angle (in radians) whose cosine is specified.
ascii	String	The ASCII code for the first character in an expression.
asin	Mathematical	The angle (in radians) whose sine is specified.
atan	Mathematical	The angle (in radians) whose tangent is specified.
atn2	Mathematical	The angle (in radians) whose sine and cosine are specified.
avg	Aggregate	The numeric average of all (distinct) values.
ceiling	Mathematical	The smallest integer greater than or equal to the specified value.
char	String	The character equivalent of an integer.
charindex	String	Returns an integer representing the starting position of an expression.
char_length	String	The number of characters in an expression.
col_length	System	The defined length of a column.
col_name	System	The name of the column whose table and column IDs are specified.
convert	Datatype Conversion	The specified value, converted to another datatype or a different *datetime* display format.
cos	Mathematical	The cosine of the specified angle (in radians).
cot	Mathematical	The cotangent of the specified angle (in radians).
count	Aggregate	The number of (distinct) non-null values.
curunreservedpgs	System	The number of free pages in the specified disk piece.
data_pgs	System	The number of pages used by the specified table or index.
datalength	System	The actual length, in bytes, of the specified column or string.
dateadd	Date	The date produced by adding a given number of years, quarters, hours, or other date parts to the specified date.
datediff	Date	The difference between two dates.
datename	Date	The name of the specified part of a *datetime* value.
datepart	Date	The integer value of the specified part of a *datetime* value.

Table 2-2: List of Transact-SQL functions (continued)

Function	Type	Return Value
db_id	System	The ID number of the specified database.
db_name	System	The name of the database whose ID number is specified.
degrees	Mathematical	The size, in degrees, of an angle with a specified number of radians.
difference	String	The difference between two **soundex** values.
exp	Mathematical	The value that results fom raising the constant e to the specified power.
floor	Mathematical	The largest integer that is less than or equal to the specified value.
getdate	Date	The current system date and time.
hextoint	Datatype Conversion	The platform-independent integer equivalent of the specified hexadecimal string.
host_id	System	The host process ID of the client process.
host_name	System	The current host computer name of the client process.
index_col	System	The name of the indexed column in the specified table or view.
inttohex	Datatype Conversion	The platform-independent, hexadecimal equivalent of the specified integer.
isnull	System	Substitutes the value specified in *expression2* when *expression1* evaluates to NULL.
is_sec_service_on	Security	"1" if the security service is active ; "0" if it is not.
lct_admin	System	Manages the last-chance threshold.
log	Mathematical	The natural logarithm of the specified number.
log10	Mathematical	The base 10 logarithm of the specified number.
lower	String	The uppercase equivalent of the specified expression.
isnull	String	The specified expression, trimmed of leading blanks.
max	Aggregate	The highest value in a column.
min	Aggregate	The lowest value in a column.
mut_excl_roles	System	The mutual exclusivity between two roles.
object_id	System	The object ID of the specified object.
object_name	System	The name of the object whose object ID is specified.
patindex	String, Text and Image	The starting position of the first occurrence of a specified pattern.

Table 2-2: List of Transact-SQL functions (continued)

Function	Type	Return Value
pi	Mathematical	The constant value 3.1415926535897936.
power	Mathematical	The value that results from raising the specified number to a given power.
proc_role	System	1 if the user has the correct role to execute the procedure; 0 if the user does not have this role.
ptn_data_pgs	System	The number of data pages used by a partition.
radians	Mathematical	The size, in radians, of an angle with a specified number of degrees.
rand	Mathematical	A random value between 0 and 1, generated using the specified seed value.
replicate	String	A string consisting of the specified expression repeated a given number of times.
reserved_pgs	System	The number of pages allocated to the specified table or index.
reverse	String	The specified string, with characters listed in reverse order.
right	String	The part of the character expression, starting the specified number of characters from the right.
role_contain	System	1 if *role2* contains *role1*.
role_id	System	The system role ID of the role whose name you specify.
role_name	System	The name of a role whose system role ID you specify.
round	Mathematical	The value of the specified number, rounded to a given number of decimal places.
rowcnt	System	An estimate of the number of rows in the specified table.
rtrim	String	The specified expression, trimmed of trailing blanks.
show_role	System	The login's currently active roles.
show_sec_services	Security	A list of the user's currently active security services.
sign	Mathematical	The sign (+1 for positive, 0, or -1 for negative) of the specified value.
sin	Mathematical	The sine of the specified angle (in radians).
soundex	String	A 4-character code representing the way an expression sounds.
space	String	A string consisting of the specified number of single-byte spaces.
sqrt	Mathematical	The square root of the specified number.

Table 2-2: List of Transact-SQL functions (continued)

Function	Type	Return Value
str	String	The character equivalent of the specified number.
stuff	String	The string formed by deleting a specified number of characters from one string and replacing them with another string.
substring	String	The string formed by extracting a specified number of characters from another string.
sum	Aggregate	The total of the values.
suser_id	System	The server user's ID number from the *syslogins* system table.
suser_name	System	The name of the current server user, or the user whose server user ID is specified.
tan	Mathematical	The tangent of the specified angle (in radians).
textptr	Text and Image	The pointer to the first page of the specified *text* column.
textvalid	Text and Image	1 if the pointer to the specified *text* column is valid; 0 if it is not.
tsequal	System	Compares *timestamp* values to prevent update on a row that has been modified since it was selected for browsing.
upper	String	The uppercase equivalent of the specified string.
used_pgs	System	The number of pages used by the specified table and its clustered index.
user	System	The name of the current server user.
user_id	System	The ID number of the specified user or the current user.
user_name	System	The name within the database of the specified user or the current user.
valid_name	System	0 if the specified string is not a valid identifier; a number other than 0 if the string is valid.
valid_user	System	1 if the specified ID is a valid user or alias in at least one database on this Adaptive Server.

The following sections describe the types of functions in detail. The remainder of the chapter contains descriptions of the individual functions in alphabetical order.

Aggregate Functions

The aggregate functions generate summary values that appear as new columns in the query results. The aggregate functions are:

- **avg**
- **count**
- **max**
- **min**
- **sum**

Aggregate functions can be used in the select list or the **having** clause of a **select** statement or subquery. They cannot be used in a **where** clause.

Because each aggregate in a query requires its own worktable, a query using aggregates cannot exceed the maximum number of worktables allowed in a query (12).

When an aggregate function is applied to a *char* datatype value, it implicitly converts the value to *varchar*, stripping all trailing blanks.

Aggregates Used with *group* by

Aggregates are often used with **group by**. With **group by**, the table is divided into groups. Aggregates produce a single value for each group. Without **group by**, an aggregate function in the select list produces a single value as a result, whether it is operating on all the rows in a table or on a subset of rows defined by a **where** clause.

Aggregate Functions and NULL Values

Aggregate functions calculate the summary values of the non-null values in a particular column. If the **ansinull** option is set **off** (the default), there is no warning when an aggregate function encounters a null. If **ansinull** is set **on**, a query returns the following SQLSTATE warning when an aggregate function encounters a null:

```
Warning- null value eliminated in set function
```

Vector and Scalar Aggregates

Aggregate functions can be applied to all the rows in a table, in which case they produce a single value, a scalar aggregate. They can also be applied to all the rows that have the same value in a specified column or expression (using the **group by** and, optionally, the **having** clause), in which case, they produce a value for each group, a vector aggregate. The results of the aggregate functions are shown as new columns.

You can nest a vector aggregate inside a scalar aggregate. For example:

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

type
------------  ------------  ------------
UNDECIDED             NULL        15.23
business             13.73        15.23
mod_cook             11.49        15.23
popular_comp         21.48        15.23
psychology           13.50        15.23
trad_cook            15.96        15.23

(6 rows affected)
```

The **group by** clause applies to the vector aggregate—in this case, **avg**(*price*). The scalar aggregate, **avg**(**avg**(*price*)), is the average of the average prices by type in the *titles* table.

In standard SQL, when a *select_list* includes an aggregate, all the *select_list* columns must either have aggregate functions applied to them or be in the **group by** list. Transact-SQL has no such restrictions.

Example 1 shows a **select** statement with the standard restrictions. Example 2 shows the same statement with another item (*title_id*) added to the select list. **order by** is also added to illustrate the difference in displays. These “extra” columns can also be referenced in a **having** clause.

1. **select type, avg(price), avg(advance)**
   from titles
   group by type

```
type
------------  ------------  ------------
UNDECIDED             NULL          NULL
business             13.73      6,281.25
mod_cook             11.49      7,500.00
popular_comp         21.48      7,500.00
psychology           13.50      4,255.00
trad_cook            15.96      6,333.33

(6 rows affected)
```

2. **select type, title_id, avg(price), avg(advance)**
   from titles
   group by type
   order by type

```
type         title_id
-----------  --------    ---------- ---------
UNDECIDED       MC3026      NULL        NULL
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
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
```

You can use either a column name or any other expression (except a column heading or alias) after **group by.**

Null values in the **group by** column are put into a single group.

The **compute** clause in a **select** statement uses row aggregates to produce summary values. The row aggregates make it possible to retrieve detail and summary rows with one command. Example 3 illustrates this feature:

3. **select type, title_id, price, advance**
   from titles
   where type = "psychology"
   order by type
   compute sum(price), sum(advance) by type

```
type        title_id    price       advance
----------- -------     ----------  ---------
psychology  PS1372          21.59   7,000.00
psychology  PS2091          10.95   2,275.00
psychology  PS2106          7.00    6,000.00
psychology  PS3333          19.99   2,000.00
psychology  PS7777          7.99    4,000.00
                            sum     sum
                            ------- ----------
                            67.52   21,275.00
```

Note the difference in display between example 3 and the examples without **compute** (examples 1 and 2).

Aggregate functions cannot be used on virtual tables such as *sysprocesses* and *syslocks*.

If you include an aggregate function in the **select** clause of a cursor, that cursor is not updatable.

Aggregate Functions As Row Aggregates

Row aggregate functions generate summary values that appear as additional rows in the query results.

To use the aggregate functions as row aggregates, use the following syntax:

```
Start of select statement

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

where:

- *column_name* is the name of a column. It must be enclosed in parentheses. Only exact numeric, approximate numeric, and money columns can be used with **sum** and **avg**.

  One **compute** clause can apply the same function to several columns. When using more than one function, use more than one **compute** clause.

- **by** indicates that row aggregate values are to be calculated for subgroups. Whenever the value of the **by** item changes, row aggregate values are generated. If you use **by**, you must use **order by**.

  Listing more than one item after **by** breaks a group into subgroups and applies a function at each level of grouping.

The row aggregates make it possible to retrieve detail and summary rows with one command. The aggregate functions, on the other hand, ordinarily produce a single value for all the selected rows in the table or for each group, and these summary values are shown as new columns.

The following examples illustrate the differences:

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

type
----------  ----------  ----------------
mod_cook         22.98         15,000.00
trad_cook        47.89         19,000.00

(2 rows affected)

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

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

```
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)
```

The columns in the **compute** clause must appear in the select list.

If the **ansinull** option is set **off** (the default), there is no warning when a row aggregate encounters a null. If **ansinull** is set **on**, a query returns the following SQLSTATE warning when a row aggregate encounters a null:

```
Warning- null value eliminated in set function
```

You cannot use **select into** in the same statement as a **compute** clause because statements that include **compute** generate tables that include the summary results, which are not stored in the database.

Datatype Conversion Functions

Datatype conversion functions change expressions from one datatype to another and specify new display formats for date / time information. The datatype conversion functions are:

- **convert()**
- **inttohex()**
- **hextoint()**

The datatype conversion functions can be used in the select list, in the **where** clause, and anywhere else an expression is allowed.

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.

Table 2-3 indicates whether individual datatype conversions are performed implicitly or explicitly or are unsupported.

Table 2-3: Explicit, implicit, and unsupported datatype conversions

From / To	tinyint	smallint	int	decimal	numeric	real	float	char, nchar	varchar, nvarchar	text	smallmoney	money	bit	smalldatetime	datetime	binary	varbinary	image
tinyint	–	I	I	I	I	I	I	E	E	U	I	I	I	U	U	I	I	U
smallint	I	–	I	I	I	I	I	E	E	U	I	I	I	U	U	I	I	U
int	I	I	–	I	I	I	I	E	E	U	I	I	I	U	U	I	I	U
decimal	I	I	I	I/E	I/E	I	I	E	E	U	I	I	I	U	U	I	I	U
numeric	I	I	I	I/E	I/E	I	I	E	E	U	I	I	I	U	U	I	I	U
real	I	I	I	I	I	–	I	E	E	U	I	I	I	U	U	I	I	U
float	I	I	I	I	I	I	–	E	E	U	I	I	I	U	U	I	I	U
char, nchar	E	E	E	E	E	E	E	I	I	E	E	E	E	I	I	I	I	E
varchar, nvarchar	E	E	E	E	E	E	E	I	I	E	E	E	E	I	I	I	I	E
text	U	U	U	U	U	U	U	E	E	U	U	U	U	U	U	U	U	U
smallmoney	I	I	I	I	I	I	I	I	I	U	–	I	I	U	U	I	I	U
money	I	I	I	I	I	I	I	I	I	U	I	–	I	U	U	I	I	U
bit	I	I	I	I	I	I	I	I	I	U	I	I	–	U	U	I	I	U
smalldatetime	U	U	U	U	U	U	U	E	E	U	U	U	U	–	I	I	I	U
datetime	U	U	U	U	U	U	U	E	E	U	U	U	U	I	–	I	I	U
binary	I	I	I	I	I	I	I	I	I	U	I	I	I	I	I	–	I	E
varbinary	I	I	I	I	I	I	I	I	I	U	I	I	I	I	I	I	–	E
image	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	E	E	U

Key:
E Explicit datatype conversion is required.
I Conversion can be done either implicitly or with an explicit datatype conversion function.
I/E Explicit datatype conversion function required when there is loss of precision or scale and arithabort numeric_truncation is on; otherwise, implicit conversion is allowed.
U Unsupported conversion.
– Conversion of a datatype to itself. These conversions are allowed but are meaningless.

Converting Character Data to a Non-Character Type

Character data can be converted to a non-character type—such as a money, date/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, SQL Server converts the blanks into the default *datetime* value of "Jan 1, 1900".

Syntax errors are generated when the data includes unacceptable characters. Following are some examples of characters that cause syntax errors:

- Commas or decimal points in integer data
- Commas in monetary data
- Letters in exact or approximate numeric data or bit stream data
- Misspelled month names in date / time data

Converting from One Character Type to Another

When converting from a multibyte character set to a single-byte character set, characters with no single-byte equivalent are converted to blanks.

text columns can be explicitly converted to *char*, *nchar*, *varchar*, or *nvarchar*. You are limited to the maximum length of the *character* datatypes, 255 bytes. If you do not specify the length, the converted value has a default length of 30 bytes.

Converting Numbers to a Character Type

Exact and approximate numeric data can be converted to a character type. If the new type is too short to accommodate the entire string, an insufficient space error is generated. For example, the following conversion tries to store a 5-character string in a 1-character type:

```
select convert(char(1), 12.34)
Insufficient result space for explicit conversion
of NUMERIC value '12.34' to a CHAR field.
```

➤ ***Note***

When converting *float* data to a character type, the new type should be at least 25 characters long.

Rounding During Conversion to and 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 four places.

Data converted from a money type follows the same rounding behavior if possible. If the new type is an exact numeric with less than three decimal places, the data is rounded to the scale of the new type. For example, when $4.50 is converted to an integer, it yields 5:

```
select convert(int, $4.50)

 -----------
           5
```

Data converted to *money* or *smallmoney* is assumed to be in full currency units such as dollars rather than in fractional units such as cents. For example, the integer value of 5 is converted to the money equivalent of 5 dollars, not 5 cents, in the us_english language.

Converting Date/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 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.

For example, if you provide a float or numeric value as an argument to a built-in function that expects an integer, the value of the float or numeric is truncated. However, Adaptive Server does not implicitly convert numerics that have a fractional part but returns a scale error message. For example, Adaptive Server returns error 241 for numerics that have a fractional part and error 257 if other datatypes are passed.

Use the **arithabort** and **arithignore** options to determine how Adaptive Server handles errors resulting from numeric conversions.

➤ ***Note***

The **arithabort** and **arithignore** options have been redefined for release 10.0 or later. If you use these options in your applications, examine them to be sure they are still producing the desired behavior.

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 date / time type
- Conversions of hexadecimal strings requiring more than 4 bytes of storage using **hextoint**

Both arithmetic overflow and divide-by-zero errors are considered serious, whether they occur during an implicit or explicit conversion. 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 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, and Adaptive Server does not execute statements that follow the error-generating statement in the batch. 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 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.

➤ ***Note***

For entry level SQL92 compliance, set:

- **arithabort arith_overflow off**
- **arithabort numeric_truncation on**
- **arithignore off**

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 happens rarely.

Conversions Between Binary and Integer Types

The *binary* and *varbinary* types store hexadecimal-like data consisting of a "0x" prefix followed by a string of digits and letters.

These strings are interpreted differently by different platforms. For example, the string "0x0000100" represents 65536 on machines that consider byte 0 most significant and 256 on machines that consider byte 0 least significant.

The binary types can be converted to integer types either explicitly, using 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. Because of this, results may vary from one platform to another. Use the **hextoint** function for platform-independent conversion of hexadecimal strings to integers, and the **inttohex** function for platform-independent conversion of integers to hexadecimal values.

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),0x00000000000000006b14bd1e6eea00000000000000000000000000000000)

----------
123.456000
```

This example converts the same *numeric* data back to *binary*:

```
select convert(binary,convert(numeric(38, 18), 123.456))

--------------------------------------------------------------
0x00000000000000006b14bd1e6eea00000000000000000000000000000000
```

Converting Image Columns to Binary Types

You can use the **convert** function to convert an *image* column to *binary* or *varbinary*. You are limited to the maximum length of the *binary* datatypes, 255 bytes. If you do not specify the length, the converted value has a default length of 30 characters.

Converting Other Types to *bit*

Exact and approximate numeric types can be converted to the *bit* type implicitly. Character types require an explicit **convert** function.

The expression being converted must consist only of digits, a decimal point, a currency symbol, and a plus or minus sign. The presence of other characters generates syntax errors.

The *bit* equivalent of 0 is 0. The *bit* equivalent of any other number is 1.

Date Functions

The date functions manipulate values of the type *datetime* or *smalldatetime*.

The date functions are:

- **dateadd**
- **datediff**
- **datename**
- **datepart**
- **getdate**

Date functions can be used in the select list or **where** clause of a query.

Use the *datetime* datatype only for dates after January 1, 1753. *datetime* values must be enclosed in single or double quotes. Use *char, nchar, varchar* or *nvarchar* for earlier dates. Adaptive Server recognizes a wide variety of date formats. See “Datatype Conversion Functions” and “Date / time Datatypes” in Chapter 7, “System and User-Defined Datatypes” for more information.

Adaptive Server automatically converts between character and *datetime* values when necessary (for example, when you compare a character value to a *datetime* value).

Date Parts

The date parts, the abbreviations recognized by Adaptive Server, and the acceptable values are:

Date Part	Abbreviation	Values
year	**yy**	1753 – 9999 (2079 for *smalldatetime*)
quarter	**qq**	1 – 4
month	**mm**	1 – 12
week	**wk**	1 – 54
day	**dd**	1 – 31
dayofyear	**dy**	1 – 366
weekday	**dw**	1 – 7 (Sun.-Sat.)
hour	**hh**	0 – 23
minute	**mi**	0 – 59
second	**ss**	0 – 59
millisecond	**ms**	0 – 999

If you enter a year with only 2 digits, <50 is the next century (“25” is “2025”) and >=50 is this century (“50” is “1950”).

Milliseconds can be preceded either with a colon or a period. If preceded by a colon, the number means thousandths of a second. If preceded by a period, a single digit means tenths of a second, two digits mean hundredths of a second, and three digits mean thousandths of a second. For example, “12:30:20:1” means twenty and one-thousandth of a second past 12:30; “12:30:20.1” means twenty and one-tenth of a second past 12:30. Adaptive Server may round or truncate millisecond values when adding *datetime* data.

Mathematical Functions

Mathematical functions return values commonly needed for operations on mathematical data. Mathematical function names are not keywords.

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 automatically converts the argument to the desired type.

The mathematical functions are:

- **abs**
- **acos**
- **asin**

- **atan**
- **atn2**
- **ceiling**
- **cos**
- **cot**
- **degrees**
- **exp**
- **floor**
- **log**
- **log10**
- **pi**
- **power**
- **radians**
- **rand**
- **round**
- **sign**
- **sin**
- **sqrt**
- **tan**

Error traps are provided to handle domain or range errors of these functions. Users can **set** the **arithabort** and **arithignore** options to determine how domain errors are handled:

- **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 or aborts the batch 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.
- **arithabort numeric_truncation** specifies behavior following a loss of scale by an exact numeric type during an implicit datatype conversion. (When an explicit conversion results in a loss of scale, the results are truncated without warning.) 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.

- By default, the **arithignore arith_overflow** option is turned **off**, causing Adaptive Server to display a warning message after any query that results in numeric overflow. Set the **arithignore** option **on** to ignore overflow errors.

➤ ***Note***

The **arithabort** and **arithignore** options have been redefined for release 10.0 or later. If you use these options in your applications, examine them to be sure they still produce the desired effects.

Security Functions

Security functions return security-related information.

The security functions are:

- **is_sec_service_on**
- **show_sec_services**

String Functions

String function operate on binary data, character strings, and expressions. The string functions are:

- **ascii**
- **char**
- **charindex**
- **char_length**
- **difference**
- **lower**
- **ltrim**
- **patindex**
- **replicate**
- **reverse**
- **right**

- **rtrim**
- **soundex**
- **space**
- **str**
- **stuff**
- **substring**
- **upper**

String functions can be nested, and they can be used in a select list, in a **where** clause, or anywhere an expression is allowed. When you use constants with a string function, enclose them in single or double quotes. String function names are not keywords.

Each string 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.

Limits on String Functions

Results of string functions are limited to 255 characters.

If **set string_rtruncation** is **on**, a user receives an error if an **insert** or **update** truncates a character string. However, SQL Server does not report an error if a displayed string is truncated. For example:

```
select replicate("a", 250) + replicate("B", 250)
```

```
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
------------------------------------------------------------
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaBBBBB
```

System Functions

System functions return special information from the database. The system functions are:

- **col_length**

- **col_name**
- **curunreservedpgs**
- **data_pgs**
- **datalength**
- **db_id**
- **db_name**
- **host_id**
- **host_name**
- **index_col**
- **isnull**
- **lct_admin**
- **mut_excl_roles**
- **object_id**
- **object_name**
- **proc_role**
- **ptn_data_pgs**
- **reserved_pgs**
- **role_contain**
- **role_id**
- **role_name**
- **rowcnt**
- **show_role**
- **suser_id**
- **suser_name**
- **tsequal**
- **used_pgs**
- **user**
- **user_id**
- **user_name**
- **valid_name**
- **valid_user**

The system functions can be used in a **select** list, in a **where** clause, and anywhere an expression is allowed.

When the argument to a system function is optional, the current database, host computer, server user, or database user is assumed.

Text and Image Functions

Text and image functions operate on *text* and *image* data. The text and image functions are:

- **textptr**
- **textvalid**

Text and image built-in function names are not keywords. Use the **set textsize** option to limit the amount of *text* or *image* data that is retrieved by a **select** statement.

The **patindex** text function can be used on *text* and *image* columns and can also be considered a text and image function.

Use the **datalength** function to get the length of data in *text* and *image* columns.

text and *image* columns cannot be used:

- As parameters to stored procedures
- As values passed to stored procedures
- As local variables
- In **order by, compute**, and **group by** clauses
- In an index
- In a **where** clause, except with the keyword **like**
- In joins
- In triggers

abs

Function

Returns the absolute value of an expression.

Syntax

abs(*numeric_expression*)

Arguments

numeric_expression – is a column, variable, or expression whose datatype is an exact numeric, approximate numeric, money, or any type that can be implicitly converted to one of these types.

Examples

1. **select abs(-1)**

```
-----------
          1
```

Returns the absolute value of -1.

Comments

- **abs**, a mathematical function, 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.
- For general information about mathematical functions, see "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **abs**.

See Also

Function	**ceiling**, **floor**, **round**, **sign**

acos

Function

Returns the angle (in radians) whose cosine is specified.

Syntax

acos(*cosine*)

Arguments

cosine – is the cosine of the angle, expressed as a column name, variable, or constant of type *float*, *real*, *double precision*, or any datatype that can be implicitly converted to one of these types.

Examples

1. **select acos(0.52)**

```
--------------------
           1.023945
```

Returns the angle whose cosine is 0.52.

Comments

- **acos**, a mathematical function, returns the angle (in radians) whose cosine is the specified value.
- For general information about mathematical functions, refer to “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **acos**.

See Also

Functions	**cos**, **degrees**, **radians**

ascii

Function

Returns the ASCII code for the first character in an expression.

Syntax

```
ascii(char_expr)
```

Arguments

char_expr – is a character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

Examples

1. **`select au_lname, ascii(au_lname) from authors where ascii(au_lname) < 70`**

```
au_lname
------------------------------ -----------
Bennet                                  66
Blotchet-Halls                          66
Carson                                  67
DeFrance                                68
Dull                                    68
```

Returns the authors last names and the ACSII codes for the first letters in their last names, if the ASCII code is less than 70.

Comments

- **ascii**, a string function, returns the ASCII code for the first character in the expression.
- If *char_expr* is NULL, returns NULL.
- For general information about string functions, refer to “String Functions” on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **ascii**.

See Also

Functions	**char**

asin

Function

Returns the angle (in radians) whose sine is specified.

Syntax

```
asin(sine)
```

Arguments

sine – is the sine of the angle, expressed as a column name, variable, or constant of type *float*, *real*, *double precision*, or any datatype that can be implicitly converted to one of these types.

Examples

1. **select asin(0.52)**

```
--------------------
            0.546851
```

Comments

- **asin**, a mathematical function, returns the angle (in radians) whose sine is the specified value.
- For general information about mathematical functions, refer to “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **asin**.

See Also

Functions	**degrees**, **radians**, **sin**

atan

Function

Returns the angle (in radians) whose tangent is specified.

Syntax

```
atan(tangent)
```

Arguments

tangent – is the tangent of the angle, expressed as a column name, variable, or constant of type *float*, *real*, *double precision*, or any datatype that can be implicitly converted to one of these types.

Examples

1. **select atan(0.50)**

```
--------------------
            0.463648
```

Comments

- **atan**, a mathematical function, returns the angle (in radians) whose tangent is the specified value.
- For general information about mathematical functions, refer to "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **atan**.

See Also

Functions	**atn2**, **degrees**, **radians**, **tan**

atn2

Function

Returns the angle (in radians) whose sine and cosine are specified.

Syntax

```
atn2(sine, cosine)
```

Arguments

sine – is the sine of the angle, expressed as a column name, variable, or constant of type *float*, *real*, *double precision*, or any datatype that can be implicitly converted to one of these types.

cosine – is the cosine of the angle, expressed as a column name, variable, or constant of type *float*, *real*, *double precision*, or any datatype that can be implicitly converted to one of these types.

Examples

1. **select atn2(.50, .48)**

```
--------------------
            0.805803
```

Comments

- **atn2**, a mathematical function, returns the angle (in radians) whose sine and cosine are specified.
- For general information about mathematical functions, refer to “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **atn2**.

See Also

Functions	**atan**, **degrees**, **radians**, **tan**

avg

Function

Returns the numeric average of all (distinct) values.

Syntax

```
avg([all | distinct] expression)
```

Arguments

all – applies **avg** to all values. **all** is the default.

distinct – eliminates duplicate values before **avg** is applied. **distinct** is optional.

expression – is a column name, constant, function, any combination of column names, constants, and functions connected by arithmetic or bitwise operators, or a subquery. With aggregates, an expression is usually a column name. (See "Expressions" in Appendix A, "Expressions, Identifiers, and Wildcard Characters," for more information.)

Examples

1.
```
select avg(advance), sum(total_sales)
from titles
where type = "business"
```

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

   Calculates the average advance and the sum of total sales for all business books. Each of these aggregate functions produces a single summary value for all of the retrieved rows.

2.
```
select type, avg(advance), sum(total_sales)
from titles
group by type
```

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

Used with a **group by** clause, the aggregate functions produce single values for each group, rather than for the whole table. This statement produces summary values for each type of book.

3. **select pub_id, sum(advance), avg(price)**
   from titles
   group by pub_id
   having sum(advance) > $25000 and avg(price) > $15

   Groups the *titles* table by publishers and includes only those groups of publishers who have paid more than $25,000 in total advances and whose books average more than $15 in price.

```
pub_id
------ ------------------- -------------------
0877             41,000.00                15.41
1389             30,000.00                18.98
```

Comments

- **avg**, an aggregate function, finds the average of the values in a column. **avg** can only be used on numeric (integer, floating point, or money) datatypes. Null values are ignored in calculating averages.
- For general information about aggregate functions, refer to "Aggregate Functions" on page 2-6.
- When you average integer data, Adaptive Server treats the result as an *int* value, even if the datatype of the column is *smallint* or *tinyint*. To avoid overflow errors in DB-Library programs, declare all variables for results of averages or sums as type *int*.
- You cannot use **avg**() with the binary datatypes.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **avg**.

See Also

Functions	**max**, **min**

ceiling

Function

Returns the smallest integer greater than or equal to the specified value.

Syntax

`ceiling(`*`value`*`)`

Arguments

value – is a column, variable, or expression whose datatype is exact numeric, approximate numeric, money, or any type that can be implicitly converted to one of these types.

Examples

1. **`select ceiling(123.45)`**

```
124
```

2. **`select ceiling(-123.45)`**

```
-123
```

3. **`select ceiling(1.2345E2)`**

```
24.000000
```

4. **`select ceiling(-1.2345E2)`**

```
-123.000000
```

5. **`select ceiling($123.45)`**

```
124.00
```

6. **`select discount, ceiling(discount) from salesdetail where title_id = "PS3333"`**

```
discount
 -------------------- --------------------
           45.000000            45.000000
           46.700000            47.000000
           46.700000            47.000000
           50.000000            50.000000
```

Comments

- **ceiling**, a mathematical function, returns the smallest integer that is greater than or equal to the specified value. The return value has the same datatype as the value supplied.

For *numeric* and *decimal* values, results have the same precision as the value supplied and a scale of zero.

- For general information about mathematical functions, refer to “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **ceiling**.

See Also

Commands	**set**
Functions	**abs**, **floor**, **round**, **sign**

char

Function

Returns the character equivalent of an integer.

Syntax

char(*integer_expr*)

Arguments

integer_expr – is any integer (*tinyint*, *smallint*, or *int*) column name, variable, or constant expression between 0 and 255.

Examples

1. **select char(42)**

```
-
*
```

2. **select xxx = char(65)**

```
xxx
---
A
```

Comments

- **char**, a string function, converts a single-byte integer value to a character value. (**char** is usually used as the inverse of **ascii**.)
- **char** returns a *char* datatype. If the resulting value is the first byte of a multibyte character, the character may be undefined.
- If *char_expr* is NULL, returns NULL.
- For general information about string functions, refer to "String Functions" on page 2-22.

Reformatting Output with char

- You can use concatenation and **char**() values to add tabs or carriage returns to reformat output. **char**(10) converts to a return; **char**(9) converts to a tab.

For example:

```
/* just a space */
select title_id +  " " + title from titles where title_id = "T67061"
/* a return */
select title_id + char(10) + title from titles where title_id = "T67061"
/* a tab */
select title_id + char(9) + title from titles where title_id = "T67061"
```

```
------------------------------------------------------------------------
T67061 Programming with Curses
------------------------------------------------------------------------
T67061

Programming with Curses
------------------------------------------------------------------------
T67061      Programming with Curses
```

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **char**.

See Also

Functions	**ascii**, **str**

charindex

Function

Returns an integer representing the starting position of an expression.

Syntax

charindex(*expression1*, *expression2*)

Arguments

expression – is a binary or character column name, variable or constant expression. Can be *char*, *varchar*, *nchar* or *nvarchar* data, *binary* or *varbinary*.

Examples

1.
   ```
   select charindex("wonderful", notes)
   from titles
   where title_id = "TC3218"
   ```

   ```
   -----------
            46
   ```

   Returns the position at which the character expression “wonderful” begins in the *notes* column of the *titles* table.

Comments

- **charindex**, a string function, searches *expression2* for the first occurrence of *expression1* and returns an integer representing its starting position. If *expression1* is not found, **charindex** returns 0.
- If *expression1* contains wildcard characters, **charindex** treats them as literals.
- If *char_expr* is NULL, returns NULL.
- For general information about string functions, refer to “String Functions” on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **charindex**.

See Also

Functions	**patindex**

char_length

Function

Returns the number of characters in an expression.

Syntax

```
char_length(char_expr)
```

Arguments

char_expr – is a character-type column name, variable, or constant expression of *char, varchar, nchar* or *nvarchar* type.

Examples

1.
```
select char_length("notes" from titles
where title_id = "PC9999"

 -----------
           5
```

2.
```
declare @var varchar(20)
select @var = "abcd    "
select char_length(@var)

-----------
          7
```

Comments

- **char_length**, a string function, returns an integer representing the number of characters in a character expression or text value.
- For variable-length columns, **char_length** strips the expression of trailing blanks before counting the number of characters. For literals or variables, **char_length** does not strip the expression of trailing blanks (see example 2).
- For multi-byte character sets, the number of characters in the expression is usually less than the number of bytes; use **datalength** to determine the number of bytes.
- If *char_expr* is NULL, returns 0.
- For general information about string functions, refer to “String Functions” on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **char_length**.

See Also

Functions	**datalength**

col_length

Function

Returns the defined length of a column.

Syntax

```
col_length(object_name, column_name)
```

Arguments

object_name – is name of a database object, such as a table, view, procedure, trigger, default, or rule. The name can be fully qualified (that is, it can include the database and owner name). It must be enclosed in quotes.

column_name – is the name of the column.

Examples

1. **select x = col_length("titles", "title")**

```
 x
----
  80
```

Finds the length of the *title* column in the *titles* table. The “x” gives a column heading to the result.

Comments

- **col_length**, a system function, returns the defined length of column.
- For general information about system functions, refer to “System Functions” on page 2-23.
- To find the actual length of the data stored in each row, use **datalength**.
- For *text* and *image* columns, **col_length** returns 16, the length of the *binary(16)* pointer to the actual text page.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **col_length**.

See Also

Functions	**datalength**

col_name

Function

Returns the name of the column whose table and column IDs are specified.

Syntax

```
col_name(object_id, column_id[, database_id])
```

Arguments

object_id – is a numeric expression that is an object ID for a table, view, or other database object. These are stored in the *id* column of *sysobjects*.

column_id – is a numeric expression that is a column ID of a column. These are stored in the *colid* column of *syscolumns*.

database_id – is a numeric expression that is the ID for a database. These are stored in the *db_id* column of *sysdatabases*.

Examples

1. **select col_name(208003772, 2)**

```
------------------------------
title
```

Comments

- **col_name**, a system function, returns the column's name.
- For general information about system functions, refer to "System Functions" on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **col_name.**

See Also

Functions	**db_id**, **object_id**

convert

Function

Returns the specified value, converted to another datatype or a different *datetime* display format.

Syntax

```
convert (datatype [(length) | (precision[, scale])],
   expression[, style])
```

Arguments

datatype – is the system-supplied datatype (for example, *char*(10), *varbinary*(50), or *int*) into which to convert the expression. You cannot use user-defined datatypes.

length – is an optional parameter used with *char*, *nchar*, *varchar*, *nvarchar*, *binary* and *varbinary* datatypes. If you do not supply a length, Adaptive Server truncates the data to 30 characters for the character types and 30 bytes for the binary types. The maximum allowable length for character and binary data is 255 bytes.

precision – is the number of significant digits in a *numeric* or *decimal* datatype. For *float* datatypes, precision is the number of significant binary digits in the mantissa. If you do not supply a precision, Adaptive Server uses the default precision of 18 for *numeric* and *decimal* datatypes.

scale – is the number of digits to the right of the decimal point in a *numeric*, or *decimal* datatype. If you do not supply a scale, Adaptive Server uses the default scale of 0.

expression – is the value to be converted from one datatype or date format to another.

style – is the display format to use for the converted data. When converting *money* or *smallmoney* data to a character type, use a *style* of 1 to display a comma after every 3 digits.

When converting *datetime* or *smalldatetime* data to a character type, use the style numbers in Table 2-4 to specify the display format. Values in the left-most column display 2-digit years (yy).

For 4-digit years (yyyy), add 100, or use the value in the middle column.

Table 2-4: Display formats for date/time information

Without Century (yy)	With Century (yyyy)	Output
N / A	0 or 100	*mon dd yyyy hh:mi*AM (or PM)
1	101	*mm / dd / yy*
2	102	*yy.mm.dd*
3	103	*dd / mm / yy*
4	104	*dd.mm.yy*
5	105	*dd-mm-yy*
6	106	*dd mon yy*
7	107	*mon dd, yy*
8	108	*hh:mm:ss*
N / A	9 or 109	*mon dd yyyy hh:mi:ss:mmm*AM (or PM)
10	110	*mm-dd-yy*
11	111	*yy / mm / dd*
12	112	*yymmdd*

The default values (*style* 0 or 100), and *style* 9 or 109 always return the century (yyyy). When converting to *char* or *varchar* from *smalldatetime*, styles that include seconds or milliseconds show zeros in those positions.

Examples

1.
```
select title, convert(char(12), total_sales)
from titles
```
2.
```
select title, total_sales
from titles
where convert(char(20), total_sales) like "1%"
```
3.
```
select convert(char(12), getdate(), 3)
```
   Converts the current date to style "3", *dd/mm/yy*.
4.
```
select convert(varchar(12), pubdate, 3) from titles
```
   If the value *pubdate* can be null, you must use *varchar* rather than *char*, or errors may result.

5. **`select convert(integer, 0x00000100)`**

   Returns the integer equivalent of the string "0x00000100". Results can vary from one platform to another.

6. **`select convert (binary, 10)`**

   Returns the platform-specific bit pattern as a Sybase binary type.

7. **`select convert(bit, $1.11)`**

   Returns 1, the bit string equivalent of $1.11.

Comments

- **convert**, a datatype conversion function, converts between a wide variety of datatypes and reformats date / time and money data for display purposes.
- For more information about datatype conversion, refer to "Datatype Conversion Functions" on page 2-11.
- **convert()** generates a domain error when the argument falls outside the range over which the function is defined. This should happen rarely.
- You can use **convert** 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.

Standards and Compliance

Standard	**Compliance Level**
SQL92	Transact-SQL extension

Permissions

Any user can execute **convert.**

See Also

Datatypes	System and User-Defined Datatypes
Functions	**hextoint**, **inttohex**

cos

Function

Returns the cosine of the specified angle.

Syntax

```
cos(angle)
```

Arguments

angle – is any approximate numeric (*float*, *real*, or *double precision*) column name, variable, or constant expression.

Examples

1. **select cos(44)**

```
    0.999843
```

Comments

- **cos**, a mathematical function, returns the cosine of the specified angle (in radians).
- For general information about mathematical functions, refer to "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **cos**.

See Also

Functions	**acos**, **degrees**, **radians**, **sin**

cot

Function

Returns the cotangent of the specified angle.

Syntax

```
cot(angle)
```

Arguments

angle – is any approximate numeric (*float*, *real*, or *double precision*) column name, variable, or constant expression.

Examples

1. **select cot(90)**

```
--------------------
          -0.501203
```

Comments

- **cot**, a mathematical function, returns the cotangent of the specified angle (in radians).
- For general information about mathematical functions, refer to “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **cot**.

See Also

Functions	**degrees**, **radians**, **tan**

count

Function

Returns the number of (distinct) non-null values or the number of selected rows.

Syntax

```
count([all | distinct] expression)
```

Arguments

all – applies **count** to all values. **all** is the default.

distinct – eliminates duplicate values before **count** is applied. **distinct** is optional.

expression – is a column name, constant, function, any combination of column names, constants, and functions connected by arithmetic or bitwise operators, or a subquery. With aggregates, an expression is usually a column name. (See “Expressions” in Appendix A, “Expressions, Identifiers, and Wildcard Characters,” for more information.)

Examples

1.
```
select count(distinct city)
from authors
```

   Finds the number of different cities in which authors live.

2.
```
select type
from titles
group by type
having count(*) > 1
```

   Lists the types in the *titles* table, but eliminates the types that include only one book or none.

Comments

- **count**, an aggregate function, finds the number of non-null values in a column. For general information about aggregate functions, refer to “Aggregate Functions” on page 2-6.
- When **distinct** is specified, **count** finds the number of unique non-null values. **count** can be used with all datatypes except *text* and *image*. Null values are ignored when counting.

- **count**(*column_name*) returns a value of 0 on empty tables, on columns that contain only null values, and on groups that contain only null values.
- **count(*)** finds the number of rows. **count(*)** does not take any arguments, and cannot be used with **distinct**. All rows are counted, regardless of the presence of null values.
- When tables are being joined, include **count(*)** in the **select list** to produce the count of the number of rows in the joined results. If the objective is to count the number of rows from one table that match criteria, use **count**(*column_name*).
- **count()** can be used as an existence check in a subquery. For example:

  ```
  select * from tab where 0 <
      (select count(*) from tab2 where ...)
  ```

  However, because **count()** counts all matching values, **exists** or **in** may return results faster. For example:

  ```
  select * from tab where exists
      (select * from tab2 where ...)
  ```

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **count**.

See Also

Commands	**compute Clause**, **group by and having Clauses**, **select**, **where Clause**

curunreservedpgs

Function

Returns the number of free pages in the specified disk piece.

Syntax

curunreservedpgs*(dbid, lstart, unreservedpgs*)

Arguments

dbid – is the ID for a database. These are stored in the *db_id* column of *sysdatabases*.

lstart – is a page within the disk piece for which pages are to be returned.

unreservedpgs – is the default value to return if the *dbtable* is presently unavailable for the requested database.

Examples

1.
```
select db_name(dbid), d.name,
    curunreservedpgs(dbid, lstart, unreservedpgs)
    from sysusages u, sysdevices d
    where d.low <= u.size + vstart
        and d.high >= u.size + vstart -1
        and d.status &2 = 2
```

```
master          master       184
master          master       832
tempdb          master       464
tempdb          master      1016
tempdb          master       768
model           master       632
sybsystemprocs    master     1024
pubs2             master     248
```

Returns the database name, device name, and the number of unreserved pages for each device fragment.

2.
```
select curunreservedpgs (dbid, sysusages.lstart, 0)
```

Displays the number of free pages on the segment for *dbid* starting on *sysusages.lstart*.

Comments

- **curunreservedpgs**, a system function, returns the number of free pages in a disk piece. For general information about system functions, refer to “System Functions” on page 2-23.
- 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*.

Standards and Compliance

Standard	**Compliance Level**
SQL92	Transact-SQL extension

Permissions

Any user can execute **curunreservedpgs**.

See Also

Functions	**db_id**, **lct_admin**

data_pgs

Function

Returns the number of pages used by the specified table or index.

Syntax

```
data_pgs(object_id,
   {data_oam_pg_id | index_oam_pg_id})
```

Arguments

object_id – is an object ID for a table, view, or other database object. These are stored in the *id* column of *sysobjects*.

data_oam_pg_id – is the page ID for a data OAM page, stored in the *doampg* column of *sysindexes*.

index_oam_pg_id – is the page ID for an index OAM page, stored in the *ioampg* column of *sysindexes*.

Examples

1.
```
select sysobjects.name,
Pages = data_pgs(sysindexes.id, doampg)
from sysindexes, sysobjects
where sysindexes.id = sysobjects.id
    and sysindexes.id > 100
    and (indid = 1 or indid = 0)
```

   Estimates the number of data pages used by user tables (which have object IDs that are greater than 100). An *indid* of 0 indicates a table without a clustered index; an *indid* of 1 indicates a table with a clustered index. This example does not include nonclustered indexes or text chains.

2.
```
select sysobjects.name,
Pages = data_pgs(sysindexes.id, ioampg)
from sysindexes, sysobjects
where sysindexes.id = sysobjects.id
    and sysindexes.id > 100
    and (indid > 1)
```

   Estimates the number of data pages used by user tables (which have object IDs that are greater than 100), nonclustered indexes, and page chains.

Comments

- **data_pgs** works only on objects in the current database.

Accuracy of Results

- If used on the transaction log (*syslogs*), the result may not be accurate and can be off by up to 16 pages.

Errors

- Instead of returning an error, **data_pgs** returns 0 if any of the following are true:
  - The *object_id* does not exist in *sysobjects*
  - The *control_page_id* does not belong to the table specified by *object_id*
  - The *object_id* is -1
  - The *page_id* is -1

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **data_pgs**.

Tables Used

sysindexes, *syspartitions*

See Also

Functions	**object_id**, **rowcnt**, **used_pgs**
System procedures	**sp_spaceused**

datalength

Function

Returns the actual length, in bytes, of the specified column or string.

Syntax

```
datalength(expression)
```

Arguments

expression – is a column name, variable, constant expression, or a combination of any of these that evaluates to a single value. It can be of any datatype. *expression* is usually a column name. If *expression* is a character constant, it must be enclosed in quotes.

Examples

1.
```
select Length = datalength(pub_name)
from publishers
```

```
Length
-----------
         13
         16
          20
```

Finds the length of the *pub_name* column in the *publishers* table.

Comments

- **datalength**, a system function, returns the length of *expression* in bytes.
- **datalength** finds the actual length of the data stored in each row. **datalength** is useful on *varchar*, *varbinary*, *text* and *image* datatypes, since these datatypes can store variable lengths (and do not not store trailing blanks). When a *char* value is declared to allow nulls, Adaptive Server stores it internally as a *varchar*. For all other datatypes, **datalength** reports their defined length.
- **datalength** of any NULL data returns NULL.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **datalength**.

See Also

Functions	**char_length**, **col_length**

dateadd

Function

Returns the date produced by adding a given number of years, quarters, hours, or other date parts to the specified date.

Syntax

dateadd(*date_part*, *integer*, *date*)

Arguments

date_part – is a date part or abbreviation. See "Date Parts" on page 2-20 for a list of the date parts and abbreviations recognized by Adaptive Server.

numeric – is an integer expression.

date – is either the function **getdate**, a character string in one of the acceptable date formats, an expression that evaluates to a valid date format, or the name of a *datetime* column.

Examples

1.
   ```
   select newpubdate = dateadd(day, 21, pubdate)
   from titles
   ```

   Displays the new publication dates when the publication dates of all the books in the *titles* table slip by 21 days.

Comments

- **dateadd**, a date function, adds an interval to a specified date. For more information about date functions, refer to "Date Functions" on page 2-19.
- **dateadd** takes three arguments—the date part, a number, and a date. The result is a *datetime* value equal to the date plus the number of date parts.

  If the date argument is a *smalldatetime* value, 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.
- Use the *datetime* datatype only for dates after January 1, 1753. *datetime* values must be enclosed in single or double quotes. Use *char*, *nchar*, *varchar* or *nvarchar* for earlier dates. Adaptive Server

recognizes a wide variety of date formats. See “Datatype Conversion Functions” and “System and User-Defined Datatypes” for more information.

Adaptive Server automatically converts between character and *datetime* values when necessary (for example, when you compare a character value to a *datetime* value).

- Using the date part **weekday** or **dw** with **dateadd** is not logical, and produces spurious results. Use **day** or **dd** instead.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **dateadd**.

See Also

Commands	**select**, **where Clause**
Datatypes	“Date / time Datatypes”
Functions	**datediff**, **datename**, **datepart**, **getdate**

datediff

Function

Returns the difference between two dates.

Syntax

datediff(*datepart*, *date1*, *date2*)

Arguments

datepart – is a date part or abbreviation. See “Date Parts” on page 2-20 for a list of the date parts, the abbreviations recognized by Adaptive Server.

date1 – can be either the function **getdate**, a character string in an acceptable date format, an expression that evaluates to a valid date format, or the name of a *datetime* column.

date2 – can be either the function **getdate**, a character string in an acceptable date format, an expression that evaluates to a valid date format, or the name of a *datetime* or *smalldatetime* column.

Examples

1.
```
select newdate = datediff(day, pubdate, getdate())
from titles
```

   This query finds the number of days that have elapsed between *pubdate* and the current date (obtained with the **getdate** function).

Comments

- **datediff**, a date function, calculates the number of date parts between two specified dates. For more information about date functions, refer to “Date Functions” on page 2-19.
- **datediff** takes three arguments. The first is a date part. The second and third are dates. The result is a signed integer value equal to *date2* - *date1*, in date parts.
- **datediff** produces results of datatype *int*, and causes errors if the result is greater than 2,147,483,647. For milliseconds, this is approximately 24 days, 20:31.846 hours. For seconds, this is 68 years, 19 days, 3:14:07 hours.
- **datediff** results are always truncated, not rounded, when the result is not an even multiple of the date part. For example, using **hour** as

the date part, the difference between "4:00AM" and "5:50AM" is 1.

When you use **day** as the date part, **datediff** counts the number of midnights between the two times specified. For example, the difference between January 1, 1992, 23:00 and January 2, 1992, 01:00 is 1; the difference between January 1, 1992 00:00 and January 1, 1992, 23:59 is 0.

- The **month** datepart counts the number of first-of-the-months between two dates. For example, the difference between January 25 and February 2 is 1; the difference between January 1 and January 31 is 0.
- When you use the date part **week** with **datediff**, you get the number of Sundays between the two dates, including the second date but not the first. For example, the number of weeks between Sunday, January 4 and Sunday, January 11 is 1.
- If *smalldatetime* values are used, they are converted to *datetime* values internally for the calculation. Seconds and milliseconds in *smalldatetime* values are automatically set to 0 for the purpose of the difference calculation.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **datediff**.

See Also

Datatypes	"Date/time Datatypes"
Commands	**select**, **where Clause**
Functions	**dateadd**, **datename**, **datepart**, **getdate**

datename

Function

Returns the name of the specified part of a *datetime* value.

Syntax

```
datename (datepart, date)
```

Arguments

datepart – is a date part or abbreviation. See "Date Parts" on page 2-20 for a list of the date parts, the abbreviations recognized by Adaptive Server.

date – can be either the function **getdate**, a character string in an acceptable date format, an expression that evaluates to a valid date format, or the name of a *datetime* or *smalldatetime* column.

Examples

1.
```
select datename(month, getdate())
```

```
November
```

This example assumes a current date of November 25, 1995.

Comments

- **datename**, a date function, returns the name of the specified part (such as the month "June") of a *datetime* or *smalldatetime* value, as a character string. If the result is numeric, such as "23" for the day, it is still returned as a character string.
- For more information about date functions, refer to "Date Functions" on page 2-19.
- The date part **weekday** or **dw** returns the day of the week (Sunday, Monday, and so on) when used with **datename**.
- Since *smalldatetime* is accurate only to the minute, when a *smalldatetime* value is used with **datename**, seconds and milliseconds are always 0.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **datename**.

See Also

Datatypes	"Date / time Datatypes"
Commands	**select**, **where Clause**
Functions	**dateadd**, **datediff**, **datepart**, **getdate**

datepart

Function

Returns the integer value of the specified part of a *datetime* value.

Syntax

datepart(*date_part*, *date*)

Arguments

date_part – is a date part. Table 2-5 lists the date parts, the abbreviations recognized by **datepart**, and the acceptable values.

Table 2-5: Date parts and their values

Date Part	Abbreviation	Values
year	**yy**	1753 – 9999 (2079 for *smalldatetime*)
quarter	**qq**	1 – 4
month	**mm**	1 – 12
week	**wk**	1 – 54
day	**dd**	1 – 31
dayofyear	**dy**	1 – 366
weekday	**dw**	1 – 7 (Sun.-Sat.)
hour	**hh**	0 – 23
minute	**mi**	0 – 59
second	**ss**	0 – 59
millisecond	**ms**	0 – 999
calweekofyear	**cwk**	1-53
calyearofweek	**cyr**	1753 – 9999
caldayofweek	**cdw**	1 – 7

If you enter the year as two digits, <50 is the next century ("25" is "2025") and >=50 is this century ("50" is "1950").

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 twenty and one-thousandth of a second past 12:30; "12:30:20.1" means twenty and one-tenth of a second past 12:30.

date – can be either the function **getdate**, a character string in an acceptable date format, an expression that evaluates to a valid date format, or the name of a *datetime* or *smalldatetime* column.

Examples

1. **`select datepart(month, getdate())`**

   ```
   -----------
            11
   ```

   This example assumes a current date of November 25, 1995.

2. **`select datepart(year, pubdate) from titles where type = "trad_cook"`**

   ```
    -----------
          1990
          1985
          1987
   ```

3. **`select datepart(cwk,'1993/01/01')`**

   ```
   -----------
            53
   ```

4. **`select datepart(cyr,'1993/01/01')`**

   ```
   -----------
          1992
   ```

5. **`select datepart(cdw,'1993/01/01')`**

   ```
   -----------
             5
   ```

Comments

- **datepart**, a date function, returns an integer value for the specified part of a *datetime* value. For more information about date functions, refer to “Date Functions” on page 2-19.
- The date part **weekday** or **dw** returns the corresponding number when used with **datepart**. The numbers that correspond to the names of weekdays depend on the **datefirst** setting. Some language defaults (including us_english) produce 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**.
- **calweekofyear**, which can be abbreviated as **cwk**, returns the ordinal position of the week within the year. **calyearofweek**, which can be abbreviated as **cyr**, returns the year in which the week begins. **caldayofweek**, which can abbreviated as **cdw**, returns the ordinal position of the day within the week. You cannot use **calweekofyear**, **calyearofweek**, and **caldayofweek** as date parts for **dateadd**, **datediff** and **datename**.

- Since *smalldatetime* is accurate only to the minute, when a *smalldatetime* value is used with **datepart**, seconds and milliseconds are always 0.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **datepart**.

See Also

Datatypes	"Date / time Datatypes"
Commands	**select**, **where Clause**
Functions	**dateadd**, **datediff**, **datename**, **getdate**

db_id

Function

Returns the ID number of the specified database.

Syntax

db_id(*database_name*)

Arguments

database_name – is the name of a database. *database_name* must be a character expression; if it is a constant expression, it must be enclosed in quotes.

Examples

1.
```
select db_id("sybsystemprocs")
------
4
```

Comments

- **db_id**, a system function, returns the database ID number.
- If you do not specify a *database_name*, **db_id** returns the ID number of the current database.
- For general information about system functions, refer to “System Functions” on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **db_id**.

See Also

Functions	**db_name**, **object_id**

db_name

Function

Returns the name of the database whose ID number is specified.

Syntax

db_name([*database_id*])

Arguments

database_id – is a numeric expression for the database ID (stored in *sysdatabases.dbid*).

Examples

1. **select db_name()**

   Returns the name of the current database.

2. **select db_name(4)**

   ```
   ------------------------------
   sybsystemprocs
   ```

Comments

- **db_name**, a system function, returns the database name.
- If no *database_id* is supplied, **db_name** returns the name of the current database.
- For general information about system functions, refer to "System Functions" on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **db_name**.

See Also

Functions	**db_id**, **col_name**, **object_name**

degrees

Function

Returns the size, in degrees, of an angle with the specified number of radians.

Syntax

degrees(*numeric*)

Arguments

numeric – is a number, in radians, to convert to degrees.

Examples

1. **select degrees(45)**

```
-----------
       2578
```

Comments

- **degrees**, a mathematical function, 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 datatypes are used, internal conversion to *float* may cause loss of precision.
- For general information about mathematical functions, refer to “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **degrees**.

See Also

Functions	**radians**

difference

Function

Returns the difference between two **soundex** values.

Syntax

```
difference(char_expr1, char_expr2)
```

Arguments

char_expr1 – is a character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

char_expr2 – is another character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

Examples

1. **select difference("smithers", "smothers")**

```
---------
4
```

2. **select difference("smothers", "brothers")**

```
---------
2
```

Comments

- **difference**, a string function, returns an integer representing the difference between two **soundex** values.
- The **difference** function compares two strings and evaluates the similarity between them, returning a value from 0 to 4. The best match is 4.

  The string values must be composed of a contiguous sequence of valid single- or double-byte roman letters.
- If *char_expr1* or *char_expr2* is NULL, returns NULL.
- For general information about string functions, refer to “String Functions” on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **difference**.

See Also

Functions	**soundex**

exp

Function

Returns the value that results from raising the constant e to the specified power.

Syntax

```
exp(approx_numeric)
```

Arguments

approx_numeric – is any approximate numeric (*float*, *real*, or *double precision*) column name, variable, or constant expression.

Examples

1. **select exp(3)**

```
--------------------
           20.085537
```

Comments

- **exp**, a mathematical function, returns the exponential value of the specified value.
- For general information about mathematical functions, see "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **exp**.

See Also

Functions	**log**, **log10**, **power**

floor

Function

Returns the largest integer that is less than or equal to the specified value.

Syntax

`floor(`*`numeric`*`)`

Arguments

numeric – is 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.

Examples

1. **`select floor(123)`**

```
-----------
        123
```

2. **`select floor(123.45)`**

```
-------
    123
```

3. **`select floor(1.2345E2)`**

```
--------------------
         123.000000
```

4. **`select floor(-123.45)`**

```
-------
   -124
```

5. **`select floor(-1.2345E2)`**

```
--------------------
        -124.000000
```

6. **`select floor($123.45)`**

```
------------------------
                 123.00
```

Comments

- **floor**, a mathematical function, 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 numeric and decimal expressions, the results have a precision equal to that of the expression and a scale of 0.
- For general information about mathematical functions, see "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **floor**.

See Also

Functions	**abs**, **ceiling**, **round**, **sign**

getdate

Function

Returns the current system date and time.

Syntax

```
getdate()
```

Arguments

None.

Examples

1. **`select getdate()`**

   ```
   Nov 25 1995 10:32AM
   ```

2. **`select datepart(month, getdate())`**

   ```
   1
   ```

3. **`select datename(month, getdate())`**

   ```
   November
   ```

   These examples assume a current date of November 25, 1995, 10:32 a.m.

Comments

- **getdate**, a date function, returns the current system date and time.
- For more information about date functions, see “Date Functions” on page 2-19.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **getdate**.

See Also

Datatypes	“Date/time Datatypes”
Functions	**dateadd**, **datediff**, **datename**, **datepart**

hextoint

Function

Returns the platform-independent integer equivalent of a hexadecimal string.

Syntax

hextoint (*hexadecimal_string*)

Arguments

hexadecimal_string – is the hexadecimal value to be converted to an integer. This must be either a character type column or variable name or a valid hexadecimal string, with or without a “0x” prefix, enclosed in quotes.

Examples

1. **select hextoint ("0x00000100")**

   Returns the integer equivalent of the hexadecimal string “0x00000100”. The result is always 256, regardless of the platform on which it is executed.

Comments

- **hextoint**, a datatype conversion function, returns the platform-independent integer equivalent of a hexadecimal string.
- For more information about datatype conversion, see “Datatype Conversion Functions” on page 2-11.
- 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.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **hextoint**.

See Also

Functions	**convert**, **inttohex**

host_id

Function

Returns the host process ID or the client process.

Syntax

```
host_id()
```

Arguments

None.

Examples

1. **select host_id()**

```
------------------------------
24711
```

Comments

- **host_id**, a system function, returns the host process ID of the client process (not the Server process).
- For general information about system functions, see “System Functions” on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **host_id**.

See Also

Functions	**host_name**

host_name

Function

Returns the current host computer name of the client process.

Syntax

```
host_name()
```

Arguments

None.

Examples

1. **select host_name()**

```
------------------------------
violet
```

Comments

- **host_name**, a system function, returns the current host computer name of the client process (not the Server process).
- For general information about system functions, see “System Functions” on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **host_name**.

See Also

Functions	**host_id**

index_col

Function

Returns the name of the indexed column in the specified table or view.

Syntax

```
index_col (object_name, index_id, key_# [, user_id])
```

Arguments

object_name – is the name of a table or view. The name can be fully qualified (that is, it can include the database and owner name). It must be enclosed in quotes.

index_id – is the number of *object_name*'s index. This number is the same as the value of *sysindexes.indid*.

key_# – is a key in the index. This value is between 1 and *sysindexes.keycnt* for a clustered index and between 1 and *sysindexes.keycnt*+1 for a nonclustered index.

user_id – is the owner of *object_name*. If you do not specify *user_id*, it defaults to the caller's user ID.

Examples

1.
```
declare @keycnt integer
select @keycnt = keycnt from sysindexes
     where id = object_id("t4")
     and indid = 1
while @keycnt > 0
begin
     select index_col("t4", 1, @keycnt)
     select @keycnt = @keycnt - 1
end
```

Finds the names of the keys in the clustered index on table *t4*.

Comments

- **index_col**, a system function, returns the name of the indexed column. For general information about system functions, see "System Functions" on page 2-23.
- **index_col** returns NULL if *object_name* is not a table or view name.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **index_col**.

See Also

Functions	**object_id**
System Procedures	**sp_helpindex**

inttohex

Function

Returns the platform-independent hexadecimal equivalent of the specified integer.

Syntax

```
inttohex (integer_expression)
```

Arguments

integer_expression – is the integer value to be converted to a hexadecimal string.

Examples

1. **select inttohex (10)**

```
--------
0000000A
```

Comments

- **inttohex**, a datatype conversion function, returns the platform-independent hexadecimal equivalent of an integer, without a "0x" prefix. For more information about datatype conversion, see "Datatype Conversion Functions" on page 2-11.
- 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.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **inttohex**.

See Also

Functions	**convert**, **hextoint**

isnull

Function

Substitutes the value specified in *expression2* when *expression1* evaluates to NULL.

Syntax

```
isnull(expression1, expression2)
```

Arguments

expression – is a column name, variable, constant expression, or a combination of any of these that evaluates to a single value. It can be of any datatype. *expression* is usually a column name. If *expression* is a character constant, it must be enclosed in quotes.

Examples

1.
   ```
   select isnull(price,0)
   from titles
   ```

   Returns all rows from the *titles* table, replacing null values in *price* with 0.

Comments

- **isnull**, a system function, substitutes the value specified in *expression2* when *expression1* evaluates to NULL. For general information about system functions, see "System Functions" on page 2-23.
- The datatypes of the expressions must convert implicitly, or you must use the **convert** function.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **isnull**.

See Also

Functions	**convert**

is_sec_service_on

Function

Returns 1 if the security service is active and 0 if it is not.

Syntax

is_sec_service_on(*security_service_nm*)

Arguments

security_service_nm – is the name of the security service.

Examples

1. **select is_sec_service_on("unifiedlogin")**

Comments

- Use **is_sec_service_on** to determine whether a given security service is active during the session.
- To find valid names of security services, run this query:

  select * from syssecmechs

  The *available_service* column displays the security services that are supported by Adaptive Server.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **is_sec_service_on**.

See Also

Functions	**show_sec_services**

lct_admin

Function

Manages the last-chance threshold.

Syntax

```
lct_admin({{"lastchance" | "logfull" | "unsuspend"},
   database_id | "reserve", log_pages})
```

Arguments

lastchance – creates a last-chance threshold in the specified database.

logfull – returns 1 if the last-chance threshold has been crossed in the specified database and 0 if it has not.

unsuspend – awakens suspended tasks in the database and disables the last-chance threshold if that threshold has been crossed.

database_id – specifies the database.

reserve – returns the number of free log pages required to successfully dump a transaction log of the specified size.

log_pages – is the number of pages for which to determine a last-chance threshold.

Examples

1. `select lct_admin("lastchance", 1)`

   This creates the log segment last-chance threshold for the database with *dbid* 1. It returns the number of pages at which the new threshold resides. If there was a previous last-chance threshold, it is replaced.

2. `select lct_admin("logfull", 6)`

   Returns 1 if the last-chance threshold for the database with *db_id* of 6 has been crossed, and 0 if it has not.

3. `select lct_admin("reserve", 64)`

   ```
   -----------
            16
   ```

   Calculates and returns the number of log pages that would be required to successfully dump the transaction log in a log containing 64 pages.

4. **`select lct_admin("unsuspend", 4)`**

   This form awakens any processes that may be suspended at the log segment last-chance threshold.

Comments

- **lct_admin**, a system function, manages the log segment's last-chance threshold. For general information about system functions, see "System Functions" on page 2-23.
- If **lct_admin("lastchance"**, *dbid***)** returns zero, the log is not on a separate segment in this database, so no last-chance threshold exists.
- Whenever you create a database with a separate log segment, the server creates a default last chance threshold that defaults to calling **sp_thresholdaction**. This happens even if a procedure called **sp_thresholdaction** does not exist on the server at all.

  If your log crosses the last-chance threshold, Adaptive Server suspends activity, tries to call **sp_thresholdaction**, finds it does not exist, generates an error, and then leaves processes suspended until the log can be truncated.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **lct_admin**.

See Also

Commands	**dump transaction**
Functions	**curunreservedpgs**
System Procedures	**sp_addthreshold**, **sp_dropthreshold**, **sp_helpthreshold**, **sp_modifythreshold**, **sp_thresholdaction**

log

Function

Returns the natural logarithm of the specified number.

Syntax

```
log(approx_numeric)
```

Arguments

approx_numeric – is any approximate numeric (*float*, *real*, or *double precision*) column name, variable, or constant expression.

Examples

1. **select log(20)**

```
--------------------
            2.995732
```

Comments

- **log**, a mathematical function, returns the natural logarithm of the specified value.
- For general information about mathematical functions, see “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **log**.

See Also

Functions	**log10**, **power**

log10

Function

Returns the base 10 logarithm of the specified number.

Syntax

log10(*approx_numeric*)

Arguments

approx_numeric – is any approximate numeric (*float*, *real*, or *double precision*) column name, variable, or constant expression.

Examples

1. **select log10(20)**

```
--------------------
           1.301030
```

Comments

- **log10**, a mathematical function, returns the base 10 logarithm of the specified value.
- For general information about mathematical functions, see “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **log10**.

See Also

Functions	**log**, **power**

lower

Function

Returns the lowercase equivalent of the specified expression.

Syntax

```
lower(char_expr)
```

Arguments

char_expr – is a character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

Examples

1. **select lower(city) from publishers**

```
--------------------
boston
washington
berkeley
```

Comments

- **lower**, a string function, converts uppercase to lowercase, returning a character value.
- **lower** is the inverse of **upper**.
- If *char_expr* is NULL, returns NULL.
- For general information about string functions, see "String Functions" on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **lower**.

See Also

Functions	**upper**

ltrim

Function

Returns the specified expression, trimmed of leading blanks.

Syntax

```
ltrim(char_expr)
```

Arguments

char_expr – is a character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

Examples

1. **`select ltrim("    123")`**

```
-------
123
```

Comments

- **ltrim**, a string function, removes leading blanks from the character expression. Only values equivalent to the space character in the current character set are removed.
- If *char_expr* is NULL, returns NULL.
- For general information about string functions, see "String Functions" on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **ltrim**.

See Also

Functions	**rtrim**

max

Function

Returns the highest value in an expression.

Syntax

```
max(expression)
```

Arguments

expression – is a column name, constant, function, any combination of column names, constants, and functions connected by arithmetic or bitwise operators, or a subquery.

Examples

1. **`select max(discount) from salesdetail`**

   ```
   --------------------
               62.200000
   ```

   Returns the maximum value in the *discount* column of the *salesdetail* table as a new column.

2. **`select discount from salesdetail`**
   `compute max(discount)`

   Returns the maximum value in the *discount* column of the *salesdetail* table as a new row.

Comments

- **max**, an aggregate function, finds the maximum value in a column or expression. For general information about aggregate functions, see “Aggregate Functions” on page 2-6.
- **max** can be used with exact and approximate numeric, character, and *datetime* columns. It cannot be used with *bit* columns. With character columns, **max** finds the highest value in the collating sequence. **max** ignores null values. **max** implicitly converts *char* datatypes to *varchar*, stripping all trailing blanks.
- Adaptive Server goes directly to the end of the index to find the last row for **max** when there is an index on the aggregated column, unless:
  - The *expression* not a column
  - The column is not the first column of an index

- There is another aggregate in the query
- There is a **group by** or **where** clause

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	By default, **max** is not compliant; use **set ansinull on** for compliant behavior.

Permissions

Any user can execute **max**.

See Also

Commands	**compute Clause**, **group by and having Clauses**, **select**, **where Clause**
Functions	**avg**, **min**

min

Function

Returns the lowest value in a column.

Syntax

`min(`*`expression`*`)`

Arguments

expression – is a column name, constant, function, any combination of column names, constants, and functions connected by arithmetic or bitwise operators, or a subquery. With aggregates, an expression is usually a column name. (See "Expressions" in Appendix A, "Expressions, Identifiers, and Wildcard Characters," for more information.)

Examples

```
1. select min(price) from titles
   where type = "psychology"

   -----------------------
                      7.00
```

Comments

- **min**, an aggregate function, finds the minimum value in a column.
- For general information about aggregate functions, see "Aggregate Functions" on page 2-6.
- **min** can be used with numeric, character, and *datetime* columns. It cannot be used with *bit* columns. With character columns, **min** finds the lowest value in the sort sequence. **min** implicitly converts *char* datatypes to *varchar,* stripping all trailing blanks. **min** ignores null values. **distinct** is not available, since it is not meaningful with **min**.
- Adaptive Server goes directly to the first qualifying row for **min** when there is an index on the aggregated column, unless:
  - The *expression* is not a column
  - The column is not the first column of an index
  - There is another aggregate in the query
  - There is a **group by** clause

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Entry level compliant	By default, **min** is not compliant; use **set ansinull on** for compliant behavior.

Permissions

Any user can execute **min**.

See Also

Commands	**compute Clause**, **group by and having Clauses**, **select**, **where Clause**
Functions	**avg**, **max**

mut_excl_roles

Function

Returns information about the mutual exclusivity between two roles.

Syntax

```
mut_excl_roles (role1, role2 [membership |
   activation])
```

Arguments

role1 – is one user-defined role in a mutually exclusive relationship.

role2 – is the other user-defined role in a mutually exclusive relationship.

level – is the level (membership or activation) at which the specified roles are exclusive.

Examples

1.
```
alter role admin add exclusive membership
supervisor
select
mut_excl_roles("admin", "supervisor", "membership")

-----------
          1
```

Shows that the *admin* and *supervisor* roles are mutually exclusive.

Comments

- **mut_excl_roles**, a system function, returns information about the mutual exclusivity between two roles. If the System Security Officer defines *role1* as mutually exclusive with *role2* or a role directly contained by *role2*, **mut_excl_roles** returns 1; if the roles are not mutually exclusive, **mut_excl_roles** returns 0.
- For general information about system functions, refer to “System Functions” on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **mut_excl_roles**.

See Also

Commands	**alter role**, **create role**, **drop role**, **grant**, **set**, **revoke**
Functions	**proc_role**, **role_contain**, **role_id**, **role_name**
System Procedures	**sp_activeroles**, **sp_displaylogin**, **sp_displayroles**, **sp_helprotect**, **sp_modifylogin**, **sp_role**

object_id

Function

Returns the object ID of the specified object.

Syntax

`object_id(`*`object_name`*`)`

Arguments

object_name – is the name of a database object, such as a table, view, procedure, trigger, default, or rule. The name can be fully qualified (that is, it can include the database and owner name). Enclose the *object_name* in quotes.

Examples

1. **`select object_id("titles")`**

```
-----------
  208003772
```

2. **`select object_id("master..sysobjects")`**

```
-----------
          1
```

Comments

- **object_id**, a system function, returns the object's ID. Object IDs are stored in the *id* column of *sysobjects*.
- For general information about system functions, refer to "System Functions" on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **object_id**.

See Also

Functions	**col_name**, **db_id**, **object_name**
System Procedures	**sp_help**

object_name

Function

Returns the name of the object whose object ID is specified.

Syntax

`object_name(`*`object_id`*`[, `*`database_id`*`])`

Arguments

object_id – is the object ID of a database object, such as a table, view, procedure, trigger, default, or rule. Object IDs are stored in the *id* column of *sysobjects*.

database_id – is the ID for a database if the object is not in the current database. Database IDs are stored in the *db_id* column of *sysdatabases*.

Examples

1. **select object_name(208003772)**

```
------------------------------
titles
```

2. **select object_name(1, 1)**

```
------------------------------
sysobjects
```

Comments

- **object_name**, a system function, returns the object's name.
- For general information about system functions, refer to "System Functions" on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **object_name**.

See Also

Functions	**col_name**, **db_name**, **object_id**
System Procedures	**sp_help**

patindex

Function

Returns the starting position of the first occurrence of a specified pattern.

Syntax

```
patindex("%pattern%", char_expr [, using {bytes |
   characters | chars} ] )
```

Arguments

pattern – is a character expression of the *char* or *varchar* datatype that may include any of the pattern-match wildcard characters supported by Adaptive Server. The % wildcard character must precede and follow *pattern* (except when searching for first or last characters). See “Pattern Matching with Wildcard Characters” in Appendix A, “Expressions, Identifiers, and Wildcard Characters,” for a description of the wildcard characters that can be used in *pattern*.

char_expr – is a character-type column name, variable, or constant expression of *char, varchar, nchar* or *nvarchar* type.

using – specifies a format for the starting position.

bytes – returns the offset in bytes.

chars or **characters** – returns the offset in characters (the default).

Examples

1. **select au_id, patindex("%circus%", copy)**
   from blurbs

```
au_id
 ----------- -----------
 486-29-1786           0
 648-92-1872           0
 998-72-3567          38
 899-46-2035          31
 672-71-3249           0
 409-56-7008           0
```

Selects the author ID and the starting character position of the word “circus” in the *copy* column.

2.
```
select au_id, patindex("%circus%", copy,
     using chars)
from blurbs
```

3.
```
select au_id, patindex("%circus%", copy,
     using chars)
from blurbs
```

The same as example 1.

4.
```
select name
from sysobjects
where patindex("sys[a-d]%", name) > 0
```

```
name
------------------------------
sysalternates
sysattributes
syscharsets
syscolumns
syscomments
sysconfigures
sysconstraints
syscurconfigs
sysdatabases
sysdepends
sysdevices
```

Finds all the rows in *sysobjects* that start with "sys" and whose fourth character is "a", "b", "c", or "d".

Comments

- **patindex**, a string function, returns an integer representing the starting position of the first occurrence of *pattern* in the specified character expression, or a zero if *pattern* is not found. For general information about string functions, refer to "String Functions" on page 2-22.
- **patindex** can be used on all character data, including *text* and *image* data.
- By default, **patindex** returns the offset in characters; to return the offset in bytes (multibyte character strings), specify **using bytes**.
- Include percent signs before and after *pattern*. To look for *pattern* as the first characters in a column, omit the preceding %; to look for *pattern* as the last characters in a column, omit the trailing %.
- If *char_expr* is NULL, returns 0.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **patindex**.

See Also

Functions	**charindex**, **substring**

pi

Function

Returns the constant value 3.1415926535897936.

Syntax

```
pi()
```

Arguments

None.

Examples

1. **select pi()**

```
-------------------
          3.141593
```

Comments

- **pi**, a mathematical function, returns the constant value of 3.1415926535897931.
- For general information about mathematical functions, refer to "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **pi**.

See Also

Functions	**degrees**, **radians**

power

Function

Returns the value that results from raising the specified number to a given power.

Syntax

```
power(value, power)
```

Arguments

value – is a numeric value.

power – is an exact numeric, approximate numeric, or money value.

Examples

1. **select power(2, 3)**

```
-----------
          8
```

Comments

- **power**, a mathematical function, returns the value of *value* raised to the power *power*. Results are of the same type as *value*.

  For expressions of type *numeric* or *decimal*, the results have an internal precision of 77 and a scale equal to that of the expression.

- For general information about mathematical functions, refer to "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **power**.

See Also

Functions	**exp**, **log**, **log10**

proc_role

Function

Returns information about whether the user has been granted the specified role.

Syntax

```
proc_role ("role_name")
```

Arguments

role_name – checks that the user has been granted, and has activated, a role.

Examples

1.
   ```
   create procedure sa_check as
   if (proc_role("sa_role") > 0)
   begin
        return(1)
   end
   print "You are a System Administrator."
   ```

   Creates a procedure to check if the user is a System Administrator.

2. `select proc_role("sso_role")`

   Checks that the user has been granted the System Security Officer role.

3. `select proc_role("oper_role")`

   Checks that the user has been granted the Operator role.

Comments

- **proc_role**, a system function, checks whether an invoking user has been granted a specified role. **proc_role** returns 0 if the user does not have the correct role. If the invoking user has the correct role, **proc_role** returns 1. If the invoking user has a currently active role which contains the correct role, but the correct role has not been activated, **proc_role** returns 2.
- For general information about system functions, refer to “System Functions” on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **proc_role**.

See Also

Commands	**alter role**, **create role**, **drop role**, **grant**, **set**, **revoke**
Functions	**mut_excl_roles**, **role_contain**, **role_id**, **role_name**, **show_role**
System Procedures	**sp_activeroles**, **sp_displaylogin**, **sp_displayroles**, **sp_helprotect**, **sp_modifylogin**, **sp_role**

ptn_data_pgs

Function

Returns the number of data pages used by a partition.

Syntax

```
ptn_data_pgs(object_id, partition_id)
```

Arguments

object_id – is the object ID for a table, stored in the *id* column of *sysobjects*, *sysindexes*, and *syspartitions*.

partition_id – is the partition number of a table.

Examples

1.
```
select ptn_data_pgs(object_id("salesdetail"), 1)

-----------
          5
```

Comments

- **ptn_data_pgs**, a system function, returns the number of data pages in a partitioned table.
- Use the **object_id** function to get an object's ID, and use **sp_helpartiton** to list the partitions in a table.
- The data pages returned by **ptn_data_pgs** may be inaccurate. Use the **update partition statistics**, **dbcc checktable**, **dbcc checkdb**, or **dbcc checkalloc** commands before using **ptn_data_pgs** to get the most accurate value.
- For general information about system functions, refer to "System Functions" on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Only the table owner can execute **ptn_data_pgs**.

See Also

Commands	**update partition statistics**
Functions	**data_pgs**, **object_id**
System Procedures	**sp_helpartition**

radians

Function

Returns the size, in radians, of an angle with the specified number of degrees.

Syntax

radians(*numeric*)

Arguments

numeric – is 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.

Examples

1. **select radians(2578)**

```
-----------
         44
```

Comments

- **radians**, a mathematical function, 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 money datatypes are used, internal conversion to *float* may cause loss of precision.

- For general information about mathematical functions, refer to “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **radians**.

See Also

Functions	**degrees**

rand

Function

Returns a random value between 0 and 1, which is generated using the specified seed value.

Syntax

rand([*integer*])

Arguments

integer – is any integer (*tinyint*, *smallint* or *int)* column name, variable, constant expression, or a combination of these.

Examples

1. **select rand()**

```
--------------------
          0.395740
```

2. **declare @seed int**
   select @seed=100
   select rand(@seed)

```
--------------------
            0.000783
```

Comments

- **rand**, a mathematical function, returns a random float value between 0 and 1, using the optional integer as a seed value.
- The **rand** function uses the output of a 32-bit pseudo-random integer generator. The integer is divided by the maximum 32-bit integer to give a double value between 0.0 and 1.0. The **rand** function is seeded randomly at server start-up, so getting the same sequence of random numbers is unlikely, unless the user first initializes this function with a constant seed value. The **rand** function is a global resource. Multiple users calling the **rand** function progress along a single stream of pseudo-random values. If a repeatable series of random numbers is needed, the user must assure that the function is seeded with the same value initially and that no other user calls **rand** while the repeatable sequence is desired.
- For general information about mathematical functions, refer to "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **rand**.

See Also

Datatypes	"Approximate Numeric Datatypes"

replicate

Function

Returns a string consisting of the specified expression repeated a given number of times.

Syntax

replicate (*char_expr*, *integer_expr*)

Arguments

char_expr – is a character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

integer_expr – is any integer (*tinyint*, *smallint*, or *int*) column name, variable, or constant expression.

Examples

1. **select replicate("abcd", 3)**

```
------------
abcdabcdabcd
```

Comments

- **replicate**, a string function, 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.
- If *char_expr* is NULL, returns a single NULL.
- For general information about string functions, refer to “String Functions” on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **replicate**.

See Also

Functions	**stuff**

reserved_pgs

Function

Returns the number of pages allocated to the specified table or index.

Syntax

```
reserved_pgs(object_id, {doampg|ioampg})
```

Arguments

object_id – is a numeric expression that is an object ID for a table, view, or other database object. These are stored in the *id* column of *sysobjects*.

doampg | ioampg – specifies table (**doampg**) or index (**ioampg**).

Examples

```
1. select reserved_pgs(id, doampg)
   from sysindexes where id =
       object_id("syslogs")

   -------------
            534
```

Returns the page count for the *syslogs* table.

Comments

- **reserved_pgs**, a system function, Returns the number of pages allocated to a table or an index. For general information about system functions, refer to “System Functions” on page 2-23.
- **reserved_pgs does** report pages used for internal structures.
- **reserved_pgs** works only on objects in the current database.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **reserved_pgs**.

See Also

Commands	**update statistics**
Functions	**data_pgs**, **used_pgs**

reverse

Function

Returns the specified string with characters listed in reverse order.

Syntax

reverse(*expression*)

Arguments

expression – is a character- or binary-type column name, variable, or constant expression of *char, varchar, nchar, nvarchar, binary,* or *varbinary* type.

Examples

1. **select reverse("abcd")**

```
----
dcba
```

2. **select reverse(0x12345000)**

```
----------
0x00503412
```

Comments

- **reverse**, a string function, returns the reverse of *expression*.
- If *expression* is NULL, returns NULL.
- For general information about string functions, refer to "String Functions" on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **reverse**.

See Also

Functions	**lower**, **upper**

right

Function

The rightmost part of the expression with the specified number of characters.

Syntax

right(*expression*, *integer_expr*)

Arguments

expression – is a character- or binary-type column name, variable, or constant expression of *char*, *varchar*, *nchar*, *nvarchar*, *binary*, or *varbinary* type.

integer_expr – is any integer (*tinyint*, *smallint*, or *int*) column name, variable, or constant expression.

Examples

1. **select right("abcde", 3)**

```
---
cde
```

2. **select right("abcde", 2)**

```
--
de
```

3. **select right("abcde", 6)**

```
-----
abcde
```

4. **select right(0x12345000, 3)**

```
-------
0x345000
```

5. **select right(0x12345000, 2)**

```
------
0x5000
```

6. **select right(0x12345000, 6)**

```
---------
0x12345000
```

Comments

- **right**, a string function, returns the specified number of characters from the rightmost part of the character or binary expression.
- The return value has the same datatype as the character or binary expression.
- If *expression* is NULL, returns NULL.
- For general information about string functions, refer to “String Functions” on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **right**.

See Also

Functions	**rtrim**, **substring**

role_contain

Function

Returns 1 if *role2* contains *role1*.

Syntax

```
role_contain("role1", "role2")
```

Arguments

role1 – is the name of a system or user-defined role.

role2 – is the name of another system or user-defined role.

Examples

1. **select role_contain("intern_role", "doctor_role")**

```
-----------
1
```

2. **select role_contain("specialist_role", "intern_role")**

```
-----------
0
```

Comments

- **role_contain**, a system function, returns 1 if *role1* is contained by *role2*.
- For more information about contained roles and role hierarchies, see Chapter 4, "Defining and Changing a Role Hierarchy," in the *Security Administration Guide*.
- For more information about system functions, see Chapter 2, "System Functions."

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **role_contain**.

See Also

Functions	**mut_excl_roles**, **proc_role**, **role_id**, **role_name**
Commands	**alter role**
System Procedures	**sp_activeroles**, **sp_configure**, **sp_displaylogin**, **sp_displayroles**, **sp_helpprotect**, **sp_modifylogin**, **sp_role**

role_id

Function

Returns the system role ID of the role whose name you specify.

Syntax

`role_id("`*`role_name`*`")`

Arguments

role_name – is the name of a system or user-defined role. Role names and role IDs are stored in the *syssrvroles* system table.

Examples

1. **`select role_id("sa_role")`**

   ```
   ------
   0
   ```

   Returns the system role ID of **sa_role**.

2. **`select role_id("intern_role")`**

   ```
   ------
   6
   ```

   Returns the system role ID of the "intern_role".

Comments

- **role_id**, a system function, returns the system role ID (*srid*). System role IDs are stored in the *srid* column of the *syssrvroles* system table.
- If the *role_name* is not a valid role in the system, Adaptive Server returns NULL.
- For more information on roles, see Chapter 4, "Administering Roles," in the *Security Administration Guide*. For more information about system functions, see "System Functions" in Chapter 2, "Transact-SQL Functions."

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **role_id**.

See Also

Functions	**mut_excl_roles**, **proc_role**, **role_contain**, **role_name**

role_name

Function

Returns the name of a role whose system role ID you specify.

Syntax

```
role_name(role_id)
```

Arguments

role_id – is the system role ID (*srid*) of the role. Role names are stored in *syssrvroles*.

Examples

1. **select role_name(01)**

```
    -----------------------------
    sso_role
```

Comments

- **role_name**, a system function, returns the role name.
- For more information on roles, see Chapter 4, "Administering Roles," in the *Security Administration Guide.* For more information about system functions, see "System Functions" in Chapter 2, "Transact-SQL Functions."

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **role_name**.

See Also

Functions	**mut_excl_roles, proc_role, role_contain, role_id**

round

Function

Returns the value of the specified number, rounded to a given number of decimal places.

Syntax

round(*number*, *decimal_places*)

Arguments

number – is 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.

decimal_places – is the number of decimal places to round to.

Examples

1. **select round(123.4545, 2)**

```
----------
  123.4500
```

2. **select round(123.45, -2)**

```
----------
  100.00
```

3. **select round(1.2345E2, 2)**

```
-----------------
       123.450000
```

4. **select round(1.2345E2, -2)**

```
-----------------
      100.000000
```

Comments

- **round**, a mathematical function, rounds the *number* so that it has *decimal_places* significant digits.
- A positive *decimal_places* determines the number of significant digits to the right of the decimal point; a negative *decimal_places*, the number of significant digits to the left of the decimal point.
- Results are of the same type as *number* and, for numeric and decimal expressions, have an internal precision equal to the

precision fo the first argument plus 1 and a scale equal to that of *number*.

- **round** always returns a value. If *decimal_places* is negative and exceeds the number of significant digits in *number*, Adaptive Server returns a result of 0. (This is expressed in the form 0.00, where the number of zeros to the right of the decimal point is equal to the scale of *numeric*.) For example:

  ```
  select round(55.55, -3)
  ```

  returns a value of 0.00.

- For general information about mathematical functions, refer to "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **round**.

See Also

Functions	**abs**, **ceiling**, **floor**, **sign**, **str**

rowcnt

Function

Returns an estimate of the number of rows in the specified table.

Syntax

```
rowcnt(sysindexes.doampg)
```

Arguments

sysindexes.doampg – is the row count maintained in *sysindexes*.

Examples

```
1. select name, rowcnt(sysindexes.doampg)
        from sysindexes
        where name in
           (select name from sysobjects
            where type = "U")

   name
   ------------------------------ -----------
   roysched                                87
   salesdetail                            116
   stores                                   7
   discounts                                4
   au_pix                                   0
   blurbs                                   6
```

Comments

- **rowcnt**, a system function, returns the estimated number of rows in a table. For general information about system functions, refer to “System Functions” on page 2-23.
- The value returned by **rowcnt** can vary unexpectedly when Adaptive Server reboots and recovers transactions. The value is most accurate after running one of the following commands:
  - **dbcc checkalloc**
  - **dbcc checkdb**
  - **dbcc checktable**
  - **update all statistics**
  - **update statistics**

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **rowcnt**.

See Also

Catalog stored procedures	**sp_statistics**
Commands	**dbcc**, **update statistics**
Functions	**data_pgs**
System procedures	**sp_helpartition**, **sp_spaceused**

rtrim

Function

Returns the specified expression, trimmed of trailing blanks.

Syntax

```
rtrim(char_expr)
```

Arguments

char_expr – is a character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

Examples

1. **`select rtrim("abcd    ")`**

```
--------
abcd
```

Comments

- **rtrim**, a string function, removes trailing blanks. For general information about string functions, refer to "String Functions" on page 2-22.
- If *char_expr* is NULL, returns NULL.
- Only values equivalent to the space character in the current character set are removed.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **rtrim**.

See Also

Functions	**ltrim**

show_role

Function

Shows the login's currently active system-defined roles.

Syntax

```
show_role()
```

Arguments

None.

Examples

1. **select show_role()**

   ```
   sa_role sso_role oper_role replication_role
   ```

2.
   ```
   if charindex("sa_role", show_role()) >0
   begin
        print "You have sa_role"
   end
   ```

Comments

- **show_role**, a system function, returns the login's current active system-defined roles, if any (**sa_role**, **sso_role**, **oper_role**, or **replication_role**). If the login has no roles, **show_role** returns NULL.
- When a Database Owner invokes **show_role** after using **setuser**, **show_role** displays the active roles of the Database Owner, not the user impersonated with **setuser**.
- For general information about system functions, refer to "System Functions" on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **show_role**.

See Also

Commands	**alter role**, **create role**, **drop role**, **grant**, **set**, **revoke**
Functions	**proc_role**, **role_contain**
System Procedures	**sp_activeroles**, **sp_displaylogin**, **sp_displayroles**, **sp_helprotect**, **sp_modifylogin**, **sp_role**

show_sec_services

Function

Lists the security services that are active for the session.

Syntax

```
show_sec_services()
```

Arguments

None.

Examples

```
1. select show_sec_services()
```

Comments

- Use **show_sec_services** to list the security services that are active during the session.
- If no security services are active, **show_sec_services** returns NULL.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **show_sec_services**.

See Also

Functions	**is_sec_service_on**

sign

Function

Returns the sign (+1 for positive, 0, or -1 for negative) of the specified value.

Syntax

sign(*numeric*)

Arguments

numeric – is 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.

Examples

1. **select sign(-123)**

```
-----------
         -1
```

2. **select sign(0)**

```
-----------
          0
```

3. **select sign(123)**

```
-----------
          1
```

Comments

- **sign**, a mathematical function, returns the positive (+1), zero (0), or negative (-1).
- Results are of the same type, and have the same precision and scale, as the numeric expression.
- For general information about mathematical functions, refer to "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **sign**.

See Also

Functions	**abs**, **ceiling**, **floor**, **round**

sin

Function

Returns the sine of the specified angle (in radians).

Syntax

```
sin(approx_numeric)
```

Arguments

approx_numeric – is any approximate numeric (*float*, *real*, or *double precision*) column name, variable, or constant expression.

Examples

1. **`select sin(45)`**

```
--------------------
           0.850904
```

Comments

- **sin**, a mathematical function, returns the sine of the specified angle (measured in radians).
- For general information about mathematical functions, refer to "Mathematical Functions" on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **sin**.

See Also

Functions	**cos**, **degrees**, **radians**

soundex

Function

Returns a 4-character code representing the way an expression sounds.

Syntax

```
soundex(char_expr)
```

Arguments

char_expr – is a character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

Examples

1.
```
select soundex ("smith"), soundex ("smythe")

----- -----
S530  S530
```

Comments

- **soundex**, a string function, returns a 4-character soundex code for character strings that are composed of a contiguous sequence of valid single- or double-byte roman letters.
- The **soundex** function converts an alpha string to a four-digit code for use in locating similar-sounding words or names. All vowels are ignored unless they constitute the first letter of the string.
- If *char_expr* is NULL, returns NULL.
- For general information about string functions, refer to “String Functions” on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **soundex**.

See Also

Functions	**difference**

space

Function

Returns a string consisting of the specified number of single-byte spaces.

Syntax

```
space(integer_expr)
```

Arguments

integer_expr – is any integer (*tinyint*, *smallint*, or *int*) column name, variable, or constant expression.

Examples

1. **`select "aaa", space(4), "bbb"`**

```
--- ---- ---
aaa      bbb
```

Comments

- **space**, a string function, returns a string with the indicated number of single-byte spaces.
- For general information about string functions, refer to "String Functions" on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **space**.

See Also

Functions	**isnull**, **isnull**, **rtrim**

sqrt

Function

Returns the square root of the specified number.

Syntax

```
sqrt(approx_numeric)
```

Arguments

approx_numeric – is any approximate numeric (*float*, *real*, or *double precision*) column name, variable, or constant expression that evaluates to a positive number.

Examples

1. **select sqrt(4)**

```
 2.000000
```

Comments

- **sqrt**, a mathematical function, returns the square root of the specified value.
- If you attempt to select the square root of a negative number, Adaptive Server returns the following error message:

  ```
  Domain error occurred.
  ```

- For general information about mathematical functions, refer to “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **sqrt**.

See Also

Functions	**power**

str

Function

Returns the character equivalent of the specified number.

Syntax

str(*approx_numeric* [, *length* [, *decimal*]])

Arguments

approx_numeric – is any approximate numeric (*float*, *real*, or *double precision*) column name, variable, or constant expression.

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). The default is 10.

decimal – sets the number of decimal digits to be returned. The default is 0.

Examples

1. **select str(1234.7, 4)**

```
----
1235
```

2. **select str(-12345, 6)**

```
------
-12345
```

3. **select str(123.45, 5, 2)**

```
-----
123.5
```

Comments

- **str**, a string function, returns a character representation of the floating point number. For general information about string functions, refer to “String Functions” on page 2-22.
- *length* and *decimal* are optional. If given, they must be nonnegative. **str** rounds the decimal portion of the number so that the results fit within the specified length. The length should be long enough to accommodate the decimal point and, if negative, 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)
```

```
--
**
```

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.

- If *approx_numeric* is NULL, returns NULL.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **str**.

See Also

Functions	**abs**, **ceiling**, **floor**, **round**, **sign**

stuff

Function

Returns the string formed by deleting a specified number of characters from one string and replacing them with another string.

Syntax

stuff(*char_expr1, start, length, char_expr2*)

Arguments

char_expr1 – is a character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

start – specifies the character position at which to begin deleting characters.

length – specifies the number of characters to delete.

char_expr2 – is another character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

Examples

1. **select stuff("abc", 2, 3, "xyz")**

```
----
axyz
```

2. **select stuff("abcdef", 2, 3, null)**

```
go
---
aef
```

3. **select stuff("abcdef", 2, 3, "")**

```
----
a ef
```

Comments

- **stuff**, a string function, deletes *length* characters from *char_expr1* at *start*, and then inserts *char_expr2* into *char_expr1* at *start*. For general information about string functions, refer to “String Functions” on page 2-22.
- 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 be deleted is longer than *expr1*, *expr1* is deleted through its last character (see example 1).

- To use **stuff** to delete a character, replace *expr2* with “NULL” rather than with empty quotation marks. Using “ ‘’ to specify a null character replaces it with a space (see examples 2 and 3).
- If *char_expr1* is NULL, returns NULL. If *char_expr1* is a string value and *char_expr2* is NULL, replaces the deleted characters with nothing.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **stuff**.

See Also

Functions	**replicate**, **substring**

substring

Function

Returns the string formed by extracting the specified number of characters from another string.

Syntax

substring(*expression, start, length*)

Arguments

expression – is a binary or character column name, variable or constant expression. Can be *char*, *varchar*, *nchar* or *nvarchar* data, *binary* or *varbinary*.

start – specifies the character position at which the substring begins.

length – specifies the number of characters in the substring.

Examples

1.
```
select au_lname, substring(au_fname, 1, 1)
from authors
```
Displays the last name and first initial of each author, for example, "Bennet A."

2.
```
select substring(upper(au_lname), 1, 3)
from authors
```
Converts the author's last name to uppercase and then displays the first three characters.

3.
```
select substring((pub_id + title_id), 1, 6)
```
Concatenates *pub_id* and *title_id*, and then displays the first six characters of the resulting string.

4.
```
select substring(xactid,5,2) from syslogs
```
Extracts the lower four digits from a binary field, where each position represents two binary digits:

Comments

- **substring**, a string function, returns part of a character or binary string. For general information about string functions, refer to "String Functions" on page 2-22.
- If any of the arguments to **substring** are NULL, returns NULL.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **substring**.

See Also

Functions	**charindex**, **patindex**, **stuff**

sum

Function

Returns the total of the values.

Syntax

```
sum([all | distinct] expression)
```

Arguments

all – applies **sum** to all values. **all** is the default.

distinct – eliminates duplicate values before **sum** is applied. **distinct** is optional.

expression – is a column name, constant, function, any combination of column names, constants, and functions connected by arithmetic or bitwise operators, or a subquery. With aggregates, an expression is usually a column name. (See “Expressions” in Appendix A, “Expressions, Identifiers, and Wildcard Characters,” for more information.)

Examples

1.
```
select avg(advance), sum(total_sales)
from titles
where type = "business"
```

   Calculates the average advance and the sum of total sales for all business books. Each of these aggregate functions produces a single summary value for all of the retrieved rows.

2.
```
select type, avg(advance), sum(total_sales)
from titles
group by type
```

   Used with a **group by** clause, the aggregate functions produce single values for each group, rather than for the whole table. This statement produces summary values for each type of book.

3.
```
select pub_id, sum(advance), avg(price)
from titles
group by pub_id
having sum(advance) > $25000 and avg(price) > $15
```

   Groups the *titles* table by publishers, and includes only those groups of publishers who have paid more than $25,000 in total advances and whose books average more than $15 in price.

Comments

- **sum**, an aggregate function, finds the sum of all the values in a column. **sum** can only be used on numeric (integer, floating point, or money) datatypes. Null values are ignored in calculating sums.
- For general information about aggregate functions, refer to “Aggregate Functions” on page 2-6.
- When you sum integer data, Adaptive Server treats the result as an *int* value, even if the datatype of the column is *smallint* or *tinyint*. To avoid overflow errors in DB-Library programs, declare all variables for results of averages or sums as type *int*.
- You cannot use **sum** with the binary datatypes.

Standards and Compliance

Standard	Compliance Level	Comments
SQL92	Transact-SQL extension	By default, **sum** is not compliant. **set ansinull on** for compliant behavior.

Permissions

Any user can execute **sum**.

See Also

Commands	**compute Clause**, **group by and having Clauses**, **select**, **where Clause**
Functions	**count**, **max**, **min**

suser_id

Function

Returns the server user's ID number from the *syslogins* table.

Syntax

```
suser_id([server_user_name])
```

Arguments

server_user_name – is an Adaptive Server login name.

Examples

1.
```
select suser_id()
```
```
------
     1
```

2.
```
select suser_id("margaret")
```
```
------
     5
```

Comments

- **suser_id**, a system function, returns the server user's ID number from *syslogins*. For general information about system functions, refer to "System Functions" on page 2-23.
- To find the user's ID in a specific database from the *sysusers* table, use the **user_id** system function.
- If no *server_user_name* is supplied, **suser_id** returns the server ID of the current user.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **suser_id**.

See Also

Functions	**suser_name**, **user_id**

suser_name

Function

Returns the name of the current server user or the user whose server ID is specified.

Syntax

```
suser_name([server_user_id])
```

Arguments

server_user_name – is an Adaptive Server user ID.

Examples

1. **select suser_name()**

```
------------------------------
sa
```

2. **select suser_name(4)**

```
------------------------------
margaret
```

Comments

- **suser_name**, a system function, returns the server user's name. Server user IDs are stored in *syslogins*. If no *server_user_id* is supplied, **suser_name** returns the name of the current user.
- For general information about system functions, refer to "System Functions" on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **suser_name**.

See Also

Functions	**suser_id**, **user_name**

tan

Function

Returns the tangent of the specified angle (in radians).

Syntax

```
tan(angle)
```

Arguments

angle – is the size of the angle in radians, expressed as a column name, variable, or expression of type *float*, *real*, *double precision*, or any datatype that can be implicitly converted to one of these types.

Examples

1. **select tan(60)**

```
--------------------
           0.320040
```

Comments

- **tan**, a mathematical function, returns the tangent of the specified angle (measured in radians).
- For general information about mathematical functions, refer to “Mathematical Functions” on page 2-20.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **tan**.

See Also

Functions	**atan**, **atn2**, **degrees**, **radians**

textptr

Function

Returns the 16-byte *binary* pointer to the first page of the specified *text* column.

Syntax

```
textptr(column_name)
```

Arguments

column_name – is the name of a *text* column.

Examples

1.
```
declare @val varbinary(16)
select @val = textptr(copy) from blurbs
where au_id = "486-29-1786"
readtext blurbs.copy @val 1 5
```

   This example uses the **textptr** function to locate the *text* column, *copy*, associated with *au_id* 486-29-1786 in the author's *blurbs* table. The text pointer is put into a local variable *@val* and supplied as a parameter to the **readtext** command, which returns 5 bytes, starting at the second byte (offset of 1).

2.
```
select au_id, textptr(copy) from blurbs
```

   Selects the *title_id* column and the 16-byte text pointer of the *blurb* column from the *texttest* table.

Comments

- **textptr**, a text and image function, 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.
- If a *text* or an *image* column has not been initialized by a non-null **insert** or by any **update** statement, **textptr** returns a NULL pointer. Use **textvalid** to check whether a text pointer exists. You cannot use **writetext** or **readtext** without a valid text pointer.
- For general information about text and image functions, refer to "Text and Image Functions" on page 2-25.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **textptr**.

See Also

Datatypes	"text and image Datatypes"
Functions	**textvalid**

textvalid

Function

Returns 1 if the pointer to the specified *text* column is valid; 0 if it is not.

Syntax

```
textvalid("table_name.column_name", textpointer)
```

Arguments

"*table_name.column_name*" – is the name of a table and its *text* column.

textpointer – is a text pointer value.

Examples

1.
   ```
   select textvalid ("texttest.blurb",
   textptr(blurb)) from texttest
   ```

   Reports whether a valid text pointer exists for each value in the *blurb* column of the *texttest* table.

Comments

- **textvalid**, a text and image function, checks that a given text pointer is valid. Returns 1 if the pointer is valid or 0 if it is not.
- The identifier for a *text* or an *image* column must include the table name.
- For general information about text and image functions, refer to "Text and Image Functions" on page 2-25.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **textvalid**.

See Also

Datatypes	"text and image Datatypes"
Functions	**textptr**

tsequal

Function

Compares *timestamp* values to prevent update on a row that has been modified since it was selected for browsing.

Syntax

tsequal(*browsed_row_timestamp*, *stored_row_timestamp*)

Arguments

browsed_row_timestamp – is the *timestamp* column of the browsed row.

stored_row_timestamp – is the *timestamp* column of the stored row.

Examples

1.
```
update publishers
set city == "Springfield"
where pub_id = "0736"
and tsequal(timestamp, 0x0001000000002ea8)
```

   Retrieves the *timestamp* column from the current version of the *publishers* table and compares it to the value in the *timestamp* column that has been saved. If the values in the two *timestamp* columns are equal, updates the row. If the values are not equal, returns an error message.

Comments

- **tsequal**, a system function, compares the *timestamp* column values to prevent an update on a row that has been modified since it was selected for browsing. For general information about system functions, refer to "System Functions" on page 2-23.
- **tsequal** allows you to use browse mode without calling the **dbqual** function in DB-Library. Browse mode supports the ability to perform updates while viewing data. It is used in front-end applications using Open Client and a host programming language. A table can be browsed if its rows have been timestamped.
- 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

- The **tsequal** function should not be used in the **where** clause of a select statement, only in the where clause of **insert** and **update** statements where the rest of the **where** clause matches a single unique row.

  If a *timestamp* column is used as a search clause, it should be compared like a regular *varbinary* column; that is, *timestamp1 = timestamp2*.

Timestamping a New Table for Browsing

- When creating a new table for browsing, include a column named *timestamp* in the table definition. The column is automatically assigned a datatype of *timestamp*; 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
  ```

  adds a *timestamp* column with a NULL value to each existing row. To generate a timestamp, update each existing row without specifying new column values. For example:

  ```
  update oldtable
  set col1 = col1
  ```

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **tsequal**.

See Also

Datatypes	"timestamp Datatype"

upper

Function

Returns the uppercase equivalent of the specified string.

Syntax

```
upper(char_expr)
```

Arguments

char_expr – is a character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type.

Examples

1. **select upper("abcd")**

```
----
ABCD
```

Comments

- **upper**, a string function, converts lowercase to uppercase, returning a character value.
- If *char_expr* is NULL, returns NULL.
- For general information about string functions, refer to "String Functions" on page 2-22.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **upper**.

See Also

Functions	lower

used_pgs

Function

Returns the number of pages used by the specified table and its clustered index, or the number of pages in a nonclustered index.

Syntax

used_pgs(*object_id, doampg, ioampg*)

Arguments

object_id – is the object ID of a table or the object ID of a table to which the index belongs.

doampg – is the page number for the object allocation map of a table or clustered index, stored in the *doampg* column of *sysindexes*.

ioampg – is the page number for the allocation map of a nonclustered index, stored in the *ioampg* column of *sysindexes*.

Examples

1.
```
select name, id, indid, doampg, ioampg
from sysindexes where id = object_id("titles")
```

```
name          id          indid  doampg   ioampg
------------- ----------- ------ -------- -------
titleidind      208003772      1      560     552
titleind        208003772      2        0     456
```

```
select used_pgs(208003772, 560, 552)
```

```
-----------
          6
```

Returns the number of pages used by the data and clustered index of the *titles* table.

2. **select name, id, indid, doampg, ioampg**
   from sysindexes where id = object_id("stores")

```
name          id         indid  doampg   ioampg
------------ ---------- ------ -------- -------
stores          240003886      0     464       0
```

   select used_pgs(240003886, 464, 0)

```
-----------
          2
```

   Returns the number of pages used by the *stores* table, which has no index.

Comments

- **used_pgs**, a system function, returns the total number of pages used by a table and its clustered index, or the number of pages in a nonclustered index.
- In the examples, *indid* 0 indicates a table; *indid* 1 indicates a clustered index; an *indid* of 2–250 is a nonclustered index; and an *indid* of 255 is *text* or *image* data.
- **used_pgs** only works on objects in the current database.
- Each table and each index on a table has an object allocation map (OAM), which contains information about the number of pages allocated to and used by an object. This information is updated by most Adaptive Server processes when pages are allocated or deallocated. The **sp_spaceused** system procedure reads these values to provide quick space estimates. Some **dbcc** commands update these values while they perform consistency checks.
- For general information about system functions, refer to “System Functions” on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **used_pgs**.

See Also

Functions	**data_pgs**, **object_id**

user

Function

Returns the name of the current user.

Syntax

```
user
```

Arguments

None.

Examples

1.
```
select user
------
dbo
```

Comments

- **user**, a system function, returns the user’s name.
- If the **sa_role** is active, you are automatically the Database Owner in any database you are using. Inside a database, the user name of the Database Owner is always “dbo”.
- For general information about system functions, refer to “System Functions” on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Entry level compliant

Permissions

Any user can execute **user**.

See Also

Functions	user_name

user_id

Function

Returns the ID number of the specified user or of the current user in the database.

Syntax

```
user_id([user_name])
```

Arguments

user_name – is the name of the user.

Examples

1. `select user_id()`

```
------
     1
```

2. `select user_id("margaret")`

```
------
     4
```

Comments

- **user_id**, a system function, returns the user's ID number. For general information about system functions, refer to "System Functions" on page 2-23.
- **user_id** reports the number from *sysusers* in the current database. If no *user_name* is supplied, **user_id** returns the ID of the current user. To find the server user ID, which is the same number in every database on Adaptive Server, use **suser_id**.
- Inside a database, the "guest" user ID is always 2.
- Inside a database, the **user_id** of the Database Owner is always 1. If you have the **sa_role** active, you are automatically the Database Owner in any database you are using. To return to your actual user ID, use **set sa_role off** before executing **user_id**. If you are not a valid user in the database, Adaptive Server returns an error when you use **set sa_role off**.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

You must System Administrator or System Security Officer to use this function on a *user_name* other than your own.

See Also

Commands	**setuser**
Functions	**suser_id**, **user_name**

user_name

Function

Returns the name within the database of the specified user or of the current user.

Syntax

```
user_name([user_id])
```

Arguments

user_id – is the ID of a user.

Examples

1. **select user_name()**

```
------------------------------
dbo
```

2. **select user_name(4)**

```
------------------------------
margaret
```

Comments

- **user_name**, a system function, returns the user's name, based on the user's ID in the current database. For general information about system functions, refer to "System Functions" on page 2-23.
- If no *user_id* is supplied, **user_name** returns the name of the current user.
- If the **sa_role** is active, you are automatically the Database Owner in any database you are using. Inside a database, the **user_name** of the Database Owner is always "dbo".

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

You must be a System Administrator or System Security Officer to use this function on a *user_id* other than your own.

See Also

Functions	**suser_name**, **user_id**

valid_name

Function

Returns 0 if the specified string is not a valid identifier or a number other than 0 if the string is a valid identifier.

Syntax

valid_name(*character_expression*)

Arguments

character_expression – is a character-type column name, variable, or constant expression of *char*, *varchar*, *nchar* or *nvarchar* type. Constant expressions must be enclosed in quotation marks.

Examples

1.
```
create procedure chkname
@name varchar(30)
as
     if valid_name(@name) = 0
     print "name not valid"
```

Creates a procedure to verify that identifiers are valid.

Comments

- **valid_name**, a system function, returns 0 if the *character_ expression* is not a valid identifier (illegal characters, more than 30 bytes long, or a reserved word), or a number other than 0 if it is a valid identifier.
- Adaptive Server identifiers can be a maximum of 30 bytes in length, whether single-byte or multibyte characters are used. The first character of an identifier must be either an alphabetic character, as defined in the current character set, or the underscore (_) character. Temporary table names, which begin with the pound sign (#), and local variable names, which begin with the at sign (@), are exceptions to this rule. **valid_name** returns 0 for identifiers that begin with the pound sign (#) and the at sign (@).
- For general information about system functions, refer to “System Functions” on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

Any user can execute **valid_name**.

See Also

System Procedures	**sp_checkreswords**

valid_user

Function

Returns 1 if the specified ID is a valid user or alias in at least one database on this Adaptive Server.

Syntax

```
valid_user(server_user_id)
```

Arguments

server_user_id – is a server user ID. Server user IDs are stored in the *suid* column of *syslogins*.

Examples

1. **select valid_user(4)**

```
   ---------------
                 1
```

Comments

- **valid_user**, a system function, returns 1 if the specified ID is a valid user or alias in at least one database on this Adaptive Server.
- For general information about system functions, refer to "System Functions" on page 2-23.

Standards and Compliance

Standard	Compliance Level
SQL92	Transact-SQL extension

Permissions

You must be a System Administrator or a System Security Officer to use this function on a *server_user_id* other than your own.

See Also

System Procedures	**sp_addlogin**, **sp_adduser**